Джереми Уатт • Реза Борхани • Аггелос Катсаггелос

Машинное обучение: основы, алгоритмы и практика применения



Материалы на www.bhv.ru

Подробное руководство

Machine Learning Refined

Foundations, Algorithms, and Applications

JEREMY WATT
Northwestern University, Illinois

REZA BORHANI
Northwestern University, Illinois

AGGELOS K. KATSAGGELOS
Northwestern University, Illinois

ДЖЕРЕМИ УАТТ • РЕЗА БОРХАНИ • АГГЕЛОС КАТСАГГЕЛОС

МАШИННОЕ ОБУЧЕНИЕ: основы, алгоритмы и практика применения

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.85
ББК 32.813
У15

Уатт, Дж.

У15 Машинное обучение: основы, алгоритмы и практика применения: Пер. с англ. / Дж. Уатт, Р. Борхани, А. Катсаггелос. — СПб.: БХВ-Петербург, 2022. — 640 с.: ил.

ISBN 978-5-9775-6763-3

Представлены фундаментальные знания и практические инструменты в области машинного обучения, в том числе более 100 углубленных упражнений на языке Python. Дано введение в машинное обучение и математическую оптимизацию, включая методы первого и второго порядков, градиентного спуска и Ньютона. Приведено полное описание обучения с учителем, включая линейную регрессию, двухклассовую и многоклассовую классификацию, а также обучение без учителя и фундаментальные методы генерации признаков. Дано введение в нелинейное обучение с учителем и без. Обсуждается тема автоматизированного отбора подходящих нелинейных моделей, включая перекрестную валидацию, бустирование, регуляризацию и ансамблирование. Рассмотрены фиксированно-контурные ядра, нейронные сети, деревья и другие универсальные аппроксиматоры. Отдельно дана полная трактовка продвинутых методов оптимизации. Электронный архив на сайте издательства содержит коды всех примеров и более 300 цветных иллюстраций.

Для разработчиков систем машинного обучения

УДК 004.89
ББК 32.813

Научные редакторы:

Архитектор решений,
руководитель группы разработки компании КРОК — *Дмитрий Бардин*
Ведущий разработчик компании КРОК — *Мария Водолазкая*

Группа подготовки издания:

Руководитель проекта — *Евгений Рыбаков*
Зав. редакцией — *Людмила Гауль*
Перевод с английского — *Андрея Логунова*
Компьютерная верстка — *Ольги Сергиенко*
Оформление обложки — *Зои Канторович*

© Cambridge University Press 2020
This translation of ***Machine Learning Refined*** is published by arrangement with Cambridge University Press.
Перевод издания ***Machine Learning Refined*** опубликован по соглашению с Cambridge University Press.

Подписано в печать 11.10.21.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 51,6.
Тираж 1300 экз. Заказ №2588.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-108-48072-7 (англ.)
ISBN 978-5-9775-6763-3 (рус.)

© Cambridge University Press, 2020
© Перевод на русский язык, оформление.
ООО "БХВ-Петербург", ООО "БХВ", 2021

Краткое оглавление

О книге 1

Предисловие 2
Обзор книги 3
К читателям: как пользоваться этой книгой 5
К преподавателям: как пользоваться этой книгой 6

Благодарности 12

Глава 1. Введение в машинное обучение 13
1.1. Введение 13
1.2. Как отличить кошку от собаки: подход на основе машинного обучения 13
1.3. Базовая таксономия задач машинного обучения 18
1.4. Математическая оптимизация 29
1.5. Заключение 31

ЧАСТЬ I. МАТЕМАТИЧЕСКАЯ ОПТИМИЗАЦИЯ 33

Глава 2. Прямые методы оптимизации (нулевого порядка) 35
2.1. Введение 35
2.2. Условие оптимальности нулевого порядка 37
2.3. Методы глобальной оптимизации 38
2.4. Методы локальной оптимизации 41
2.5. Случайный поиск 45
2.6. Координатный поиск и спуск 53
2.7. Заключение 55
2.8. Упражнения 56

Глава 3. Приемы оптимизации первого порядка 60
3.1. Введение 60
3.2. Условие оптимальности первого порядка 60
3.3. Геометрия ряда Тейлора первого порядка 67
3.4. Эффективное вычисление градиентов 70
3.5. Градиентный спуск 72
3.6. Два естественных недостатка градиентного спуска 80
3.7. Заключение 86
3.8. Упражнения 87

Глава 4. Приемы оптимизации второго порядка 91
4.1. Условие оптимальности второго порядка 91
4.2. Геометрия ряда Тейлора второго порядка 94
4.3. Метод Ньютона 97
4.4. Два естественных недостатка метода Ньютона 107
4.5. Заключение 108
4.6. Упражнения 109

ЧАСТЬ II. ЛИНЕЙНОЕ ОБУЧЕНИЕ С УЧИТЕЛЕМ 113

Глава 5. Линейная регрессия 115
5.1. Введение 115
5.2. Линейная регрессия на основе наименьших квадратов 115
5.3. Наименьшие абсолютные отклонения 125
5.4. Метрики качества регрессии 127
5.5. Взвешенная регрессия 130
5.6. Многовыходная регрессия 133
5.7. Заключение 137
5.8. Упражнения 137
5.9. Обобщения 140

Глава 6. Линейная двухклассовая классификация 142
6.1. Введение 142
6.2. Логистическая регрессия и стоимость перекрестной энтропии 142
6.3. Логистическая регрессия и функция softmax (функция мягкого максимума) 152
6.4. Перцептрон 158
6.5. Метод опорных векторов 169
6.6. Какой подход дает наилучшие результаты? 175
6.7. Категориальная стоимость перекрестной энтропии 176
6.8. Метрики качества классификации 179
6.9. Взвешенная двухклассовая классификация 187
6.10. Заключение 190
6.11. Упражнения 191

Глава 7. Линейная многоклассовая классификация 194
7.1. Введение 194
7.2. Многоклассовая классификация "один против всех" 194
7.3. Многоклассовая классификация и перцептрон 204
7.4. Какой подход дает наилучшие результаты? 212
7.5. Стоимостная функция категориальной перекрестной энтропии 213
7.6. Метрики качества классификации 219
7.7. Взвешенная многоклассовая классификация 223
7.8. Стохастическое и мини-пакетное обучение 223
7.9. Заключение 226
7.10. Упражнения 226

Глава 8. Линейное обучение без учителя 229
8.1. Введение 229
8.2. Фиксированные охватывающие множества, ортонормальность и проекции 229
8.3. Линейный автокодировщик и анализ главных компонент 235

8.4. Рекомендательные системы ... 241
8.5. Кластеризация методом *K* средних ... 243
8.6. Общая методика матричной факторизации ... 249
8.7. Заключение ... 253
8.8. Упражнения ... 253
8.9. Обобщения ... 255

Глава 9. Инженерия и отбор признаков ... 260
9.1. Введение ... 260
9.2. Гистограммные признаки ... 262
9.3. Шкалирование признаков посредством стандартной нормализации ... 274
9.4. Вменение значений, отсутствующих в наборе данных ... 278
9.5. Шкалирование признаков посредством декоррелирующего преобразования ... 280
9.6. Отбор признаков посредством бустинга ... 283
9.7. Отбор признаков посредством регуляризации ... 288
9.8. Заключение ... 294
9.9. Упражнения ... 294

ЧАСТЬ III. НЕЛИНЕЙНОЕ ОБУЧЕНИЕ ... 299

Глава 10. Принципы инженерии нелинейных признаков ... 301
10.1. Введение ... 301
10.2. Нелинейная регрессия ... 301
10.3. Нелинейная многовыходная регрессия ... 308
10.4. Нелинейная двухклассовая классификация ... 312
10.5. Нелинейная многоклассовая классификация ... 316
10.6. Нелинейное обучение без учителя ... 320
10.7. Заключение ... 324
10.8. Упражнения ... 324

Глава 11. Принципы усвоения признаков ... 330
11.1. Введение ... 330
11.2. Универсальные аппроксиматоры ... 333
11.3. Универсальная аппроксимация реальных данных ... 351
11.4. Наивная перекрестная валидация ... 362
11.5. Эффективная перекрестная валидация посредством бустинга ... 368
11.6. Эффективная перекрестная валидация посредством регуляризации ... 379
11.7. Тестирование данных ... 391
11.8. Какой универсальный аппроксиматор лучше всего работает на практике? ... 395
11.9. Бутстрап-агрегирование перекрестно-валидированных моделей ... 396
11.10. *K*-блочная перекрестная валидация ... 404
11.11. Когда усвоение признаков не срабатывает ... 409
11.12. Заключение ... 410
11.13. Упражнения ... 411

Глава 12. Ядерные методы ... 414
12.1. Введение ... 414
12.2. Фиксированно-контурные универсальные аппроксиматоры ... 414
12.3. Трюк с ядром ... 417
12.4. Ядра как меры сходства ... 427
12.5. Оптимизация ядрированных моделей ... 428

12.6. Перекрестная валидация ядрированных учеников 429
12.7. Заключение 430
12.8. Упражнения 431

Глава 13. Полносвязные нейронные сети 434
13.1. Введение 434
13.2. Полносвязные нейронные сети 434
13.3. Функции активации 455
13.4. Алгоритм обратного распространения 458
13.5. Оптимизация нейросетевых моделей 459
13.6. Пакетная нормализация 462
13.7. Перекрестная валидация посредством ранней остановки 470
13.8. Заключение 472
13.9. Упражнения 472

Глава 14. Деревья решений 474
14.1. Введение 474
14.2. От пней до глубоких деревьев 474
14.3. Регрессионные деревья 477
14.4. Классификационные деревья 484
14.5. Бустирование градиента 490
14.6. Случайные леса 494
14.7. Приемы перекрестной валидации для рекурсивно определенных деревьев 496
14.8. Заключение 499
14.9. Упражнения 500

ЧАСТЬ IV. ПРИЛОЖЕНИЯ 503

Приложение 1. Продвинутые методы оптимизации первого и второго порядков 505
П1.1. Введение 505
П1.2. Стохастический градиентный спуск с импульсом 505
П1.3. Нормализованный градиентный спуск 511
П1.4. Продвинутые градиентные методы 517
П1.5. Мини-пакетная оптимизация 520
П1.6. Консервативные правила длины шага 523
П1.7. Метод Ньютона, регуляризация и невыпуклые функции 532
П1.8. Безгессианов методы 535

Приложение 2. Производные и автоматическое дифференцирование 544
П2.1. Введение 544
П2.2. Производная 544
П2.3. Правила производных для элементарных функций и операций 547
П2.4. Градиент 549
П2.5. Вычислительный граф 550
П2.6. Прямой режим автоматического дифференцирования 553
П2.7. Обратный режим автоматического дифференцирования 559
П2.8. Производные более высокого порядка 563
П2.9. Ряд Тейлора 565
П2.10. Использование библиотеки autograd 571

Приложение 3. Линейная алгебра 580
П3.1. Введение 580
П3.2. Векторы и векторные операции 580
П3.3. Матрицы и матричные операции 587
П3.4. Собственные значения и собственные векторы 590
П3.5. Векторные и матричные нормы 593

Список литературы 598

Предметный указатель 634

Оглавление

О книге **1**

Предисловие **2**
Обзор книги 3
Часть I: математическая оптимизация (главы 2–4) 3
Часть II: линейное обучение (главы 5–9) 3
Часть III: нелинейное обучение (главы 10–14) 4
Часть IV: приложения 4
К читателям: как пользоваться этой книгой 5
К преподавателям: как пользоваться этой книгой 6

Благодарности **12**

Глава 1. Введение в машинное обучение **13**
1.1. Введение 13
1.2. Как отличить кошку от собаки: подход на основе машинного обучения 13
1.3. Базовая таксономия задач машинного обучения 18
1.3.1. Обучение с учителем 19
Регрессия 19
Пример 1.1. Рост задолженности по студенческим кредитам в Соединенных Штатах 20
Пример 1.2. Закон Клейбера 21
Пример 1.3. Предсказание кассового успеха 21
Пример 1.4. Применение в бизнесе и промышленности 22
Классификация 22
Пример 1.5. Обнаружение объектов 23
Пример 1.6. Анализ настроения 24
Пример 1.7. Компьютерная диагностика патологических состояний здоровья 24
Пример 1.8. Обнаружение спама 25
Пример 1.9. Финансовые применения 26
Пример 1.10. Задачи распознавания 26
1.3.2. Обучение без учителя 27
Снижение размерности 27
Кластеризация 28
1.4. Математическая оптимизация 29
1.5. Заключение 31

ЧАСТЬ I. МАТЕМАТИЧЕСКАЯ ОПТИМИЗАЦИЯ 33

Глава 2. Прямые методы оптимизации (нулевого порядка) 35
2.1. Введение 35
2.1.1. Визуализация минимумов и максимумов 35
Пример 2.1. Визуальный осмотр одновходных функций на предмет минимумов и максимумов 36
2.2. Условие оптимальности нулевого порядка 37
2.3. Методы глобальной оптимизации 38
Пример 2.2. Минимизация квадратичной функции 39
2.3.1. Проклятие размерности 40
2.4. Методы локальной оптимизации 41
2.4.1. Картина в целом 42
2.4.2. Общий каркас 43
2.4.3. Параметр длины шага 44
2.5. Случайный поиск 45
2.5.1. Картина в целом 46
2.5.2. Контроль за длиной каждого шага 47
Пример 2.3. Минимизирование простой квадратичной функции с помощью случайного поиска 47
Пример 2.4. Минимизирование функции с многочисленными локальными минимумами 49
2.5.3. Разведывание фундаментальных правил длины шага 49
Пример 2.5. Неспособность к схождению 50
2.5.4. Правила убывающей длины шага 51
2.5.5. Случайный поиск и проклятие размерности 52
2.6. Координатный поиск и спуск 53
2.6.1. Координатный поиск 53
2.6.2. Координатный спуск 54
Пример 2.6. Координатный поиск против координатного спуска 55
2.7. Заключение 55
2.8. Упражнения 56
2.1. Минимизирование квадратичной функции и проклятие размерности 56
2.2. Имплементирование случайного поиска на Python 57
2.3. Использование случайного поиска для минимизирования невыпуклой функции 57
2.4. Случайный поиск с убывающей длиной шага 57
2.5. Вероятности случайного спуска 57
2.6. Проклятие размерности повторно 58
2.7. Псевдокод для алгоритма координатного поиска 58
2.8. Координатный поиск применительно к минимизированию простой квадратичной функции 58
2.9. Координатный поиск с убывающей длиной шага 58
2.10. Координатный поиск против координатного спуска 59

Глава 3. Приемы оптимизации первого порядка 60
3.1. Введение 60
3.2. Условие оптимальности первого порядка 60
Пример 3.1. Визуальный осмотр одновходных функций для стационарных точек 62
3.2.1. Особые случаи, когда система первого порядка может быть решена вручную 63
Пример 3.2. Отыскание стационарных точек одновходных функций посредством условия первого порядка 63
Пример 3.3. Простая на вид функция 64
Пример 3.4. Стационарные точки многовходных квадратичных функций 65

3.2.2. Координатный спуск и условие оптимальности первого порядка ... 65
Пример 3.5. Минимизирование выпуклых квадратичных функций посредством координатного спуска ... 66
Пример 3.6. Решение симметричных линейных систем уравнений ... 67
3.3. Геометрия ряда Тейлора первого порядка ... 67
3.3.1. Анатомия гиперплоскостей ... 67
3.3.2. Направления самого крутого подъема и спуска ... 68
3.3.3. Градиент и направление самого крутого подъема/спуска ... 70
3.4. Эффективное вычисление градиентов ... 70
3.5. Градиентный спуск ... 72
Пример 3.7. Минимизирование невыпуклой функции с помощью градиентного спуска ... 73
Пример 3.8. Минимизирование выпуклой многовходной функции с использованием градиентного спуска ... 73
3.5.1. Базовые варианты длины шага для градиентного спуска ... 74
Пример 3.9. Выбор фиксированной длины шага для градиентного спуска ... 75
Пример 3.10. Сравнение фиксированной и убывающей длины шага ... 76
3.5.2. Осцилляция в графике истории стоимостной функции: это не всегда плохо ... 77
Пример 3.11. Осцилляторный убывающий график истории стоимостной функции против монотонного ... 77
3.5.3. Критерии схождения ... 78
3.5.4. Имплементация на Python ... 79
3.6. Два естественных недостатка градиентного спуска ... 80
3.6.1. Откуда берутся слабые стороны направления (отрицательного) градиента? ... 81
3.6.2. Направление (отрицательного) градиента ... 81
Пример 3.12. Направление отрицательного градиента ... 82
3.6.3. Зигзагообразное поведение градиентного спуска ... 83
Пример 3.13. Зигзагообразное поведение градиентного спуска ... 83
3.6.4. Медленно ползающее поведение градиентного спуска ... 85
Пример 3.14. Медленно ползущее поведение градиентного спуска ... 86
3.7. Заключение ... 86
3.8. Упражнения ... 87
3.1. Условие оптимальности первого порядка ... 87
3.2. Стационарные точки простой квадратичной функции ... 88
3.3. Стационарные точки отношения Рэлея ... 88
3.4. Координатный спуск первого порядка как схема локальной оптимизации ... 88
3.5. Опробовать градиентный спуск ... 88
3.6. Сравнить фиксированное и убывающее значения длины шага ... 88
3.7. Осцилляция на графике истории стоимостной функции ... 89
3.8. Настроить фиксированную длину шага для градиентного спуска ... 89
3.9. Закодировать импульсно-ускоренный градиентный спуск ... 89
3.10. Медленно ползающее поведение градиентного спуска ... 89
3.11. Сравнение схем нормализованного градиентного спуска ... 89
3.12. Альтернативное формальное определение липшицева градиента ... 89
3.13. Композиция функций с липшицевым градиентом ... 90

Глава 4. Приемы оптимизации второго порядка ... 91
4.1. Условие оптимальности второго порядка ... 91
Пример 4.1. Выпуклость одновходных функций ... 92
Пример 4.2. Выпуклость многовходных квадратичных функций ... 92

4.2. Геометрия ряда Тейлора второго порядка 94
4.2.1. Общий контур одновходных квадратичных функций 94
4.2.2. Общий контур многовходных квадратичных функций 95
4.2.3. Локальная кривизна и ряд Тейлора второго порядка 96
Пример 4.3. Локальная выпуклость/вогнутость и ряд Тейлора второго порядка 97
4.3. Метод Ньютона 97
4.3.1. Направление спуска 98
Пример 4.4. Стационарные точки аппроксимирования квадратичных функций 99
4.3.2. Алгоритм 100
Пример 4.5. Минимизация выпуклой функции с использованием метода Ньютона 101
Пример 4.6. Сравнение с градиентным спуском 102
4.3.3. Обеспечение числовой стабильности 104
4.3.4. Варианты длины шага 105
4.3.5. Метод Ньютона как алгоритм отыскания нуля 105
4.3.6. Имплементация на Python 106
4.4. Два естественных недостатка метода Ньютона 107
4.4.1. Минимизация невыпуклых функций 108
4.4.2. Ограничения по масштабированию 108
4.5. Заключение 108
4.6. Упражнения 109
4.1. Определение собственных значений симметричной матрицы 109
4.2. Матрицы внешнего произведения имеют все неотрицательные собственные значения 110
4.3. Альтернативный способ проверки определения выпуклости второго порядка 110
4.4. Метод Ньютона, вариант I 110
4.5. Метод Ньютона, вариант II 111
4.6. Отыскание квадратных корней 111
4.7. Невыпуклая минимизация с использованием метода Ньютона 111
4.8. Ньютонов спуск 112
4.9. Метод Ньютона как самокорректирующийся метод градиентного спуска 112
4.10. Метод Бройдена — Флетчера — Гольдфарба — Шанно (BFGS) 112

ЧАСТЬ II. ЛИНЕЙНОЕ ОБУЧЕНИЕ С УЧИТЕЛЕМ 113

Глава 5. Линейная регрессия 115
5.1. Введение 115
5.2. Линейная регрессия на основе наименьших квадратов 115
5.2.1. Нотация и моделирование 115
5.2.2. Стоимостная функция наименьших квадратов 117
5.2.3. Минимизация стоимостной функции наименьших квадратов 119
Пример 5.1. Верификация выпуклости путем визуального обследования 119
Пример 5.2. Использование градиентного спуска 120
5.2.4. Имплементация на Python 121
5.3. Наименьшие абсолютные отклонения 125
5.3.1. Подверженность наименьших квадратов к выбросам 125
Пример 5.3. Метод наименьших квадратов излишне прилегает к выбросам 125
5.3.2. Замена квадратической ошибки абсолютной ошибкой 126
Пример 5.4. Наименьшие квадраты против наименьших абсолютных отклонений 127
5.4. Метрики качества регрессии 127
5.4.1. Выполнение предсказаний с использованием натренированной модели 128

5.4.2. Суждение о качестве натренированной модели ... 128
Пример 5.5. Предсказание цен на жилье ... 129
Пример 5.6. Предсказание автомобильного пробега в милях на галлон израсходованного топлива ... 130
5.5. Взвешенная регрессия ... 130
5.5.1. Работа с дубликатами ... 130
Пример 5.7. Квантизация входных признаков может создавать дубликаты точек ... 130
5.5.2. Взвешивание точек по уверенности ... 132
Пример 5.8. Корректирование веса одной точки данных для отражения уверенности ... 132
5.6. Многовыходная регрессия ... 133
5.6.1. Нотация и моделирование ... 133
5.6.2. Стоимостные функции ... 134
Пример 5.9. Подгонка линейной модели к многовыходному регрессионному набору данных ... 135
5.6.3. Имплементация на Python ... 136
5.7. Заключение ... 137
5.8. Упражнения ... 137
5.1. Подгонка регрессионной прямой к данным о студенческой кредитной задолженности ... 137
5.2. Закон Клейбера и линейная регрессия ... 138
5.3. Стоимостная функция наименьших квадратов и один ньютонов шаг ... 138
5.4. Решение нормальных уравнений ... 139
5.5. Константа Липшица для стоимости наименьших квадратов ... 139
5.6. Сравнить стоимости наименьших квадратов и наименьших абсолютных отклонений ... 139
5.7. Эмпирически подтвердить выпуклость для игрушечного набора данных ... 139
5.8. Стоимость наименьших абсолютных отклонений является выпуклой ... 139
5.9. Предсказание цен на жилье и автомобильного пробега в милях на галлон израсходованного топлива ... 140
5.10. Ненадлежащая настройка и взвешенная регрессия ... 140
5.11. Многовыходная регрессия ... 140
5.9. Обобщения ... 140
5.9.1. Доказательство того, что стоимостная функция наименьших квадратов всегда является выпуклой ... 140

Глава 6. Линейная двухклассовая классификация ... 142
6.1. Введение ... 142
6.2. Логистическая регрессия и стоимость перекрестной энтропии ... 142
6.2.1. Нотация и моделирование ... 143
6.2.2. Подгонка прерывистой шаговой функции ... 144
Пример 6.1. Подгонка линейного регрессора к классификационным данным ... 144
Пример 6.2. Визуальный осмотр классификационных стоимостных функций ... 146
6.2.3. Логистическая сигмоидная функция ... 147
6.2.4. Логистическая регрессия с использованием стоимости наименьших квадратов ... 147
Пример 6.3. Использование нормализованного градиентного спуска ... 148
6.2.5. Логистическая регрессия с использованием стоимости перекрестной энтропии ... 149
6.2.6. Минимизирование стоимости перекрестной энтропии ... 150
Пример 6.4. Минимизирование перекрестно-энтропийной логистической регрессии ... 150
6.2.7. Имплементация на Python ... 151

6.3. Логистическая регрессия и функция softmax (функция мягкого максимума) 152
6.3.1. Разные метки, одинаковая история 153
Пример 6.5. Минимизирование мягкомаксимумной логистической регрессии с помощью стандартного градиентного спуска 155
6.3.2. Имплементация на Python 156
6.3.3. Шумные классификационные данные 157
6.4. Перцептрон 158
6.4.1. Стоимостная функция перцептрона 158
6.4.2. Минимизирование стоимости перцептрона 160
6.4.3. Softmax-аппроксимация перцептрона 161
6.4.4. Функция softmax и линейно разделяемые наборы данных 162
Пример 6.6. Полностью разделяемые данные и стоимость мягкого максимума 163
6.4.5. Нормализация признаковых весов 164
6.4.6. Регуляризирование двухклассовой классификации 167
Пример 6.7. Регуляризованная стоимость мягкого максимума 168
6.5. Метод опорных векторов 169
6.5.1. Маржевой перцептрон 169
6.5.2. Связь с функцией мягкого максимума 171
6.5.3. Границы решения с максимальной маржой 172
6.5.4. Задачи о жесткомаржевых и мягкомаржевых SVM-машинах 174
Пример 6.8. Граница решения на основе SVM 175
6.5.5. SVM-машины и шумные данные 175
6.6. Какой подход дает наилучшие результаты? 175
6.7. Категориальная стоимость перекрестной энтропии 176
6.7.1. Категориальные метки, кодированные с одним активным состоянием 177
6.7.2. Выбор нелинейности 177
6.7.3. Выбор стоимостной функции 178
6.8. Метрики качества классификации 179
6.8.1. Выполнение предсказаний, используя натренированную модель 179
6.8.2. Оценивание уверенности 180
6.8.3. Суждение о качестве натренированной модели с помощью метрики точности 181
Пример 6.9. Сравнение историй стоимостной функции и неправильных классифицирований 182
6.8.4. Суждение о качестве натренированной модели с помощью метрики сбалансированной точности 183
6.8.5. Матрица путаницы и дополнительные метрики качества 185
Пример 6.10. Обнаружение спама 186
Пример 6.11. Кредитная проверка 186
6.9. Взвешенная двухклассовая классификация 187
6.9.1. Взвешенная двухклассовая классификация 187
6.9.2. Взвешивание точек по уровню уверенности 188
6.9.3. Устранение классовых дисбалансов 188
Пример 6.12. Классовый дисбаланс и взвешенная классификация 189
6.10. Заключение 190
6.11. Упражнения 191
6.1. Имплементирование сигмоидальной функции наименьших квадратов 191
6.2. Показать эквивалентность логарифмической ошибки и поточечной стоимости перекрестной энтропии 191
6.3. Реализация функции перекрестной энтропии 191
6.4. Вычислить липшицеву константу стоимости перекрестной энтропии 191
6.5. Подтвердить расчеты градиента и гессиана 191

6.6. Показать эквивалентность логарифмической ошибки и поточечной функции мягкого максимума ... 191
6.7. Имплементирование стоимости мягкого максимума ... 192
6.8. Реализация функции softmax с помощью логарифмической ошибки ... 192
6.9. Использование градиентного спуска для минимизирования стоимости персептрона ... 192
6.10. Стоимость персептрона является выпуклой ... 192
6.11. Стоимость мягкого максимума является выпуклой ... 192
6.12. Регуляризированная функция мягкого максимума ... 192
6.13. Сравнение эффективности двухклассовых стоимостных функций (вариант I) ... 193
6.14. Сравнение эффективности двухклассовых стоимостных функций (вариант II) ... 193
6.15. Кредитная проверка ... 193
6.16. Взвешенная классификация и сбалансированная точность ... 193

Глава 7. Линейная многоклассовая классификация ... 194
7.1. Введение ... 194
7.2. Многоклассовая классификация "один против всех" ... 194
7.2.1. Нотация и моделирование ... 194
7.2.2. Тренировка C классификаторов "один против всех" ... 195
7.2.3. Случай 1. Назначение меток точкам на положительной стороне одного классификатора ... 197
7.2.4. Случай 2. Назначение меток точкам на положительной стороне более чем одного классификатора ... 198
7.2.5. Случай 3. Назначение меток точкам на положительной стороне ни одного из классификаторов ... 200
7.2.6. Все случаи вместе ... 201
7.2.7. Алгоритм "один против всех" ... 202
Пример 7.1. Классификация набора данных с $C = 4$ классами с помощью алгоритма OvA ... 202
Пример 7.2. Регрессионный взгляд на правило слияния ... 203
7.3. Многоклассовая классификация и перцептрон ... 204
7.3.1. Стоимостная функция многоклассового перцептрона ... 204
7.3.2. Минимизирование стоимости многоклассового перцептрона ... 205
Пример 7.3. Минимизирование многоклассового перцептрона ... 205
7.3.3. Альтернативные формулировки стоимости многоклассового перцептрона ... 206
7.3.4. Регуляризирование многоклассового перцептрона ... 206
7.3.5. Стоимостная функция многоклассового мягкого максимума ... 207
7.3.6. Минимизирование функции многоклассового мягкого максимума ... 208
Пример 7.4. Метод Ньютона, применяемый для минимизирования стоимости многоклассового мягкого максимума ... 208
7.3.7. Альтернативные формулировки многоклассового мягкого максимума ... 209
7.3.8. Регуляризация и многоклассовый мягкий максимум ... 210
7.3.9. Имплементация на Python ... 210
7.4. Какой подход дает наилучшие результаты? ... 212
Пример 7.5. Сравнение OvA и многоклассового перцептрона ... 212
7.5. Стоимостная функция категориальной перекрестной энтропии ... 213
7.5.1. Дискретные распределения вероятностей ... 213
7.5.2. Экспоненциальная нормализация ... 214
7.5.3. Экспоненциально нормализованные знаковые расстояния ... 216
Пример 7.6. Знаковые расстояния как распределение вероятностей ... 216
7.5.4. Категориальная классификация и стоимость категориальной перекрестной энтропии ... 217

7.6. Метрики качества классификации 219
7.6.1. Выполнение предсказаний с использованием натренированной модели 219
7.6.2. Оценивание уверенности 220
7.6.3. Суждение о качестве натренированной модели с помощью метрики точности 220
Пример 7.7. Сравнение стоимостной функции и подсчет стоимостных значений 221
7.6.4. Продвинутые метрики качества для работы с несбалансированными классами 222
Пример 7.8. Матрица путаницы для игрушечного набора данных 222
Пример 7.9. Набор данных Iris 223
7.7. Взвешенная многоклассовая классификация 223
7.8. Стохастическое и мини-пакетное обучение 223
7.8.1. Мини-пакетная оптимизация и онлайновое (потоковое) обучение 224
Пример 7.10. Распознавание рукописных цифр 225
7.9. Заключение 226
7.10. Упражнения 226
7.1. Псевдокод классификации "один против всех" 227
7.2. Классификация "один против всех" 227
7.3. Многоклассовый перцептрон 227
7.4. Многоклассовый и двухклассовый перцептроны 227
7.5. Многоклассовый мягкий максимум 227
7.6. Показать, что мультиклассовый softmax сводится к двухклассовому softmax при $C = 2$ 227
7.7. Ручные расчеты со стоимостью многоклассового мягкого максимума 227
7.8. Стоимости многоклассового перцептрона и мягкого максимума являются выпуклыми 228
7.9. Сбалансированная точность в многоклассовой окружающей обстановке 228
7.10. Взвешенный многоклассовый мягкий максимум 228
7.11. Распознавание рукописных цифр 228

Глава 8. Линейное обучение без учителя 229
8.1. Введение 229
8.2. Фиксированные охватывающие множества, ортонормальность и проекции 229
8.2.1. Нотация 229
8.2.2. Совершенное представление данных с использованием фиксированных охватывающих множеств 231
Пример 8.1. Кодировка данных 232
8.2.3. Совершенное представление данных с использованием фиксированных ортонормальных охватывающих множеств 232
8.2.4. Несовершенное представление данных с использованием фиксированных охватывающих множеств 233
8.3. Линейный автокодировщик и анализ главных компонент 235
8.3.1. Усвоение надлежащих охватывающих множеств 235
Пример 8.2. Усвоение надлежащего охватывающего множества посредством градиентного спуска 235
8.3.2. Линейный автокодировщик 236
Пример 8.3. Усвоение линейного автокодировщика с использованием градиентного спуска 237
8.3.3. Анализ главных компонент 237
Пример 8.4. Главные компоненты 238
Пример 8.5. Предупреждающий пример! 239
8.3.4. Имплементация на Python 240

8.4. Рекомендательные системы 241
8.4.1. Мотивация 241
8.4.2. Нотация и моделирование 242
8.5. Кластеризация методом *K* средних 243
8.5.1. Представление набора данных посредством кластеров 243
8.5.2. Усвоение кластеров для представления данных 245
Пример 8.6. Влияние инициализации 246
Пример 8.7. Выбор идеального числа кластеров *K* 248
8.6. Общая методика матричной факторизации 249
8.6.1. Обучение без учителя и задачи матричной факторизации 249
8.6.2. Дальнейшие вариации 252
8.7. Заключение 253
8.8. Упражнения 253
8.1. Стандартный базис 253
8.2. Кодировка данных 254
8.3. Ортогональные матрицы и собственные значения 254
8.4. Невыпуклость линейного автокодировщика 254
8.5. Минимизирование линейного автокодировщика над игрушечным набором данных 254
8.6. Получение базиса PCA 254
8.7. Предупреждающий пример 255
8.8. Выполнение алгоритма *K* средних 255
8.9. Построение графика крутого склона 255
8.10. Попеременная минимизации 255
8.9. Обобщения 255
8.9.1. Все минимумы автокодировщика являются ортогональными матрицами 255
8.9.2. Формальная процедура выведения главных компонент 256

Глава 9. Инженерия и отбор признаков 260
9.1. Введение 260
9.2. Гистограммные признаки 262
9.2.1. Гистограммные признаки для категориальных данных 262
9.2.2. Гистограммные признаки для текстовых данных 263
Пример 9.1. Сентиментный анализ 265
Пример 9.2. Обнаружение спама 266
9.2.3. Гистограммные признаки для фотоснимковых данных 267
Пример 9.3. Распознавание рукописных цифр 272
9.2.4. Гистограммные признаки для аудиоданных 272
9.3. Шкалирование признаков посредством стандартной нормализации 274
9.3.1. Стандартная нормализация 274
Пример 9.4. Линейная регрессия со стандартно нормализованными данными 276
Пример 9.5. Линейная двухклассовая классификация со стандартно нормализованными данными 278
9.3.2. Стандартно нормализованная модель 278
9.4. Вменение значений, отсутствующих в наборе данных 278
9.4.1. Вменение среднего значения 279
9.5. Шкалирование признаков посредством декоррелирующего преобразования 280
9.5.1. Декоррелирующее преобразование: картина в целом 280
9.5.2. Декоррелирующее преобразование: технические детали 282
9.5.3. "Отбеленная" модель 283

9.6. Отбор признаков посредством бустинга 283
9.6.1. Отбор признаков на основе бустинга 284
9.6.2. Отбор правильного числа признаков посредством бустинга 285
Пример 9.6. Разведывание предсказателей цен на жилье посредством бустинга 286
Пример 9.7. Разведывание предсказателей кредитного риска посредством бустинга 287
9.6.3. Об эффективности бустинга как жадного алгоритма 287
9.6.4. Взгляд на бустинг регрессии с точки зрения остатков 288
9.7. Отбор признаков посредством регуляризации 288
9.7.1. Регуляризация с использованием норм весовых векторов 289
Пример 9.8. Регуляризация с использованием ℓ_0-нормы 289
Пример 9.9. Регуляризация с использованием ℓ_1-нормы 290
Пример 9.10. Регуляризация с использованием ℓ_2-нормы 290
9.7.2. Отбор признаков посредством ℓ_1-регуляризации 290
9.7.3. Отбор верного регуляризационного параметра 291
Пример 9.11. Разведывание предсказателей цен на жилье посредством регуляризации 292
Пример 9.12. Разведывание предсказателей кредитного риска посредством регуляризации 292
9.7.4. Сравнение регуляризации и бустинга 293
9.8. Заключение 294
9.9. Упражнения 294
9.1. Спамная электронная почта 294
9.2. Классификация набора данных MNIST: пикселы в сравнении с реберными признаками 294
9.3. Студенческая кредитная задолженность 294
9.4. Наименьшие квадраты и совершенно округлые контуры: часть 1 295
9.5. Набор данных по раку молочной железы 295
9.6. Декоррелирующее преобразование и стоимость наименьших квадратов для линейной регрессии 295
9.7. Сравнение стандартной нормализации с декоррелирующим преобразованием на наборе данных MNIST 296
9.8. Наименьшие квадраты и совершенно округлые контуры: часть 2 296
9.9. Разведывание предсказателей цен на жилье 296
9.10. Предсказание автомобильного пробега в милях на галлон израсходованного топлива 297
9.11. Изучение важных предсказателей кредитного риска 297
9.12. Разведывание предсказателей цен на жилье 297
9.13. Изучение важных предсказателей кредитного риска 297

ЧАСТЬ III. НЕЛИНЕЙНОЕ ОБУЧЕНИЕ 299

Глава 10. Принципы инженерии нелинейных признаков 301
10.1. Введение 301
10.2. Нелинейная регрессия 301
10.2.1. Принципы моделирования 301
10.2.2. Инженерия признаков 304
Пример 10.1. Моделирование волны 304
Пример 10.2. Галилей и гравитация 305
10.2.3. Имплементация на Python 307
10.3. Нелинейная многовыходная регрессия 308
10.3.1. Принципы моделирования 308
10.3.2. Инженерия признаков 311
Пример 10.3. Моделирование волн 311
10.3.3. Имплементация на Python 311

10.4. Нелинейная двухклассовая классификация 312
10.4.1. Принципы моделирования 312
10.4.2. Инженерия признаков 313
Пример 10.4. Когда граница решения состоит всего из двух отдельных точек 313
Пример 10.5. Эллиптическая граница решения 315
10.4.3. Имплементация на Python 316
10.5. Нелинейная многоклассовая классификация 316
10.5.1. Принципы моделирования 317
10.5.2. Инженерия признаков 318
Пример 10.6. Многоклассовые данные с эллиптическими границами 319
10.5.3. Имплементация на Python 320
10.6. Нелинейное обучение без учителя 320
10.6.1. Принципы моделирования 320
10.6.2. Инженерия признаков 322
Пример 10.7. Округлое топологическое многообразие 322
10.7. Заключение 324
10.8. Упражнения 324
10.1. Моделирование волны 324
10.2. Моделирование роста численности популяции 324
10.3. Эксперимент Галилея 325
10.4. Закон Мура 325
10.5. Закон Ома 326
10.6. Моделирование волн 326
10.7. Эллиптическая граница решения 327
10.8. Конструирование признаков для двухклассового классификационного набора данных 327
10.9. Округлое топологическое многообразие 327
10.10. Еще одно нелинейное расширение метода главных компонент (PCA) 327
10.11. Нелинейное расширение *K* средних 328

Глава 11. Принципы усвоения признаков 330
11.1. Введение 330
11.1.1. Пределы конструирования нелинейных признаков 330
11.1.2. Краткий обзор главы 332
11.1.3. Метафора со сложностным наборным диском для усвоения признаков 332
11.2. Универсальные аппроксиматоры 333
11.2.1. Совершенные данные 334
11.2.2. Аналогия с охватывающим множеством для универсальной аппроксимации 336
Линейные комбинации векторов и функций 337
Емкость охватывающих множеств 338
Универсальная аппроксимация 341
11.2.3. Популярные универсальные аппроксиматоры 343
Пример 11.1. Семейство фиксированно-контурных универсальных аппроксиматоров 343
Пример 11.2. Нейросетевое семейство универсальных аппроксиматоров 344
Пример 11.3. Древесное семейство универсальных аппроксиматоров 345
11.2.4. Емкостный и оптимизационный наборные диски 347
Пример 11.4. Универсальная аппроксимация почти совершенных регрессионных данных 349
Пример 11.5. Универсальная аппроксимация почти совершенных классификационных данных ... 350

11.3. Универсальная аппроксимация реальных данных 351
11.3.1. Прототипические примеры 351
Пример 11.6. Универсальная аппроксимация реальных регрессионных данных 351
Пример 11.7. Универсальная аппроксимация реальных классификационных данных 354
11.3.2. Емкостный и оптимизационный наборные диски повторно 357
11.3.3. Мотивирование нового измерительного инструмента 359
11.3.4. Валидационная ошибка 361
11.4. Наивная перекрестная валидация 362
11.4.1. Картина в целом 363
Пример 11.8. Наивная перекрестная валидация и регрессия 365
Пример 11.9. Наивная перекрестная валидация и классификация 366
11.4.2. Проблемы с наивной перекрестной валидацией 368
11.5. Эффективная перекрестная валидация посредством бустинга 368
11.5.1. Картина в целом 368
11.5.2. Технические детали 371
Раунд 0 бустинга 372
Раунд 1 бустинга 372
Раунд $m > 1$ бустинга 373
11.5.3. Ранняя остановка 374
11.5.4. Недорогое, но эффективное улучшение 374
Пример 11.10. Бустинг регрессии с использованием древесных единиц 376
Пример 11.11. Бустинг классификации с использованием нейросетевых единиц 376
11.5.5. Сходство с отбором признаков 377
11.5.6. Взгляд на регрессию с точки зрения остатков 378
Пример 11.12. Бустинг с точки зрения подгонки к остатку 378
11.6. Эффективная перекрестная валидация посредством регуляризации 379
11.6.1. Картина в целом 380
11.6.2. Регуляризация на основе ранней остановки 381
11.6.3. Методы на основе регуляризаторов 386
Пример 11.13. Настройка λ для задачи двухклассовой классификации 390
11.6.4. Сходство с регуляризацией для отбора признаков 390
11.7. Тестирование данных 391
11.7.1. Переподгонка к валидационным данным 392
11.7.2. Тестовые данные и тестовая ошибка 393
11.8. Какой универсальный аппроксиматор лучше всего работает на практике? 395
11.9. Бутстрап-агрегирование перекрестно-валидированных моделей 396
11.9.1. Бутстрап-агрегирование регрессионных моделей 397
Пример 11.14. Бутстрап-агрегирование перекрестно-валидированных регрессионных моделей 397
11.9.2. Бутстрап-агрегирование классификационных моделей 399
Пример 11.15. Бутстрап-агрегирование перекрестно-валидированных двухклассовых классификационных моделей 400
Пример 11.16. Бутстрап-агрегирование перекрестно-валидированных многоклассовых классификационных моделей 402
11.9.3. Сколько моделей мы должны агрегировать на практике? 403
11.9.4. Ансамблирование: бутстрап-агрегирование против бустинга 403
11.10. *K*-блочная перекрестная валидация 404
11.10.1. Процедура *K*-блочной перекрестной валидации 404
Пример 11.17. Гравитационный эксперимент Галилея 406

11.10.2. *K*-блочная перекрестная валидация и высокоразмерное линейное моделирование 407
Пример 11.18. Полногеномные исследования ассоциаций 408
Пример 11.19. Исследования фМРТ 409
11.11. Когда усвоение признаков не срабатывает 409
11.12. Заключение 410
11.13. Упражнения 411
11.1. Наивная перекрестная валидация I 411
11.2. Наивная перекрестная валидация II 411
11.3. Перекрестная валидация на основе бустинга I 412
11.4. Перекрестная валидация на основе бустинга II 412
11.5. Перекрестная валидация на основе регуляризации 412
11.6. Бутстрап-агрегирование регрессионных моделей 412
11.7. Бутстрап-агрегирование двухклассовых классификационных моделей 412
11.8. Бутстрап-агрегирование многоклассовых классификационных моделей 412
11.9. *K*-блочная перекрестная валидация 412
11.10. Классификация сахарного диабета 413

Глава 12. Ядерные методы 414
12.1. Введение 414
12.2. Фиксированно-контурные универсальные аппроксиматоры 414
12.2.1. Тригонометрические универсальные аппроксиматоры 414
Пример 12.1. Синусоидальные аппроксиматоры 414
Пример 12.2. Базис Фурье 415
12.2.2. Трудности масштабирования фиксированно-контурных аппроксиматоров с большой входной размерностью 416
Пример 12.3. Число единиц в многочленном аппроксиматоре степени *D* 416
Пример 12.4. Число единиц в Фурье-аппроксиматоре степени *D* 417
12.3. Трюк с ядром 417
12.3.1. Полезный факт из фундаментальной теоремы линейной алгебры 418
12.3.2. Ядрирование стоимостных функций машинного обучения 419
Пример 12.5. Ядрирование регрессии посредством стоимости наименьших квадратов 419
Пример 12.6. Ядрирование двухклассовой классификации посредством стоимости мягкого максимума 420
12.3.3. Популярные ядра в машинном обучении 422
Пример 12.7. Многочленное ядро 422
Пример 12.8. Ядро Фурье 423
Пример 12.9. Ядро радиальной базисной функции 424
12.3.4. Выполнение предсказаний с помощью ядрированных моделей 426
12.4. Ядра как меры сходства 427
12.5. Оптимизация ядрированных моделей 428
12.6. Перекрестная валидация ядрированных учеников 429
Пример 12.10. Классификация рака молочной железы с использованием ядра RBF 429
12.7. Заключение 430
12.8. Упражнения 431
12.1. Комплексное представление Фурье 431
12.2. Комбинаторный взрыв в одночленах 431
12.3. Регрессия с многочленным ядром 431
12.4. Ядрировать ℓ_2-регуляризованную стоимость наименьших квадратов 432
12.5. Ядрировать стоимость многоклассового мягкого максимума 432

12.6. Регрессия с ядром RBF 432
12.7. Двухклассовая классификация с ядром RBF 432
12.8. Многоклассовая классификация с ядром RBF 432
12.9. Многочленные ядра для произвольной степени и входной размерности 432
12.10. Преобразование бесконечномерных признаков 432
12.11. Ядро Фурье для векторнозначного входа 432
12.12. Ядра и набор данных о раке 433

Глава 13. Полносвязные нейронные сети 434
13.1. Введение 434
13.2. Полносвязные нейронные сети 434
13.2.1. Структурные единицы с одним скрытым слоем 434
Пример 13.1. Иллюстрирование емкости однослойных структурных единиц 435
Компактное представление однослойных нейронных сетей 438
13.2.2. Структурные единицы с двумя скрытыми слоями 439
Пример 13.2. Иллюстрирование емкости двухслойных структурных единиц 440
Компактное представление двухслойных нейронных сетей 441
13.2.3. Общие структурные единицы с многочисленными скрытыми слоями 444
Пример 13.3. Иллюстрирование емкости трехслойных структурных единиц 444
Компактное представление многослойных нейронных сетей 445
13.2.4. Отбор правильной сетевой архитектуры 448
13.2.5. Нейронные сети: биологическая точка зрения 448
13.2.6. Имплементация на Python 450
Пример 13.4. Нелинейная классификация с использованием многослойных нейронных сетей 452
Пример 13.5. Случайные автокодировочные топологические многообразия 453
Пример 13.6. Нелинейный автокодировщик с использованием многослойных нейронных сетей 454
13.3. Функции активации 455
Пример 13.7. Шаговая и сигмоидная активации 456
Пример 13.8. Функция активации ReLU 456
Пример 13.9. Активация maxout 458
13.4. Алгоритм обратного распространения 458
13.5. Оптимизация нейросетевых моделей 459
13.5.1. Невыпуклость 459
Пример 13.10. Сравнение первопорядковых оптимизаторов на многослойной нейросетевой модели 461
13.6. Пакетная нормализация 462
13.6.1. Пакетная нормализация структурных единиц с одним скрытым слоем 463
Пример 13.11. Визуализирование внутреннего ковариатного сдвига в однослойной сети 464
13.6.2. Пакетная нормализация структурных единиц с многочисленными скрытыми слоями 466
Пример 13.12. Визуализирование внутреннего ковариатного сдвига в многослойной сети 467
Пример 13.13. Стандартная нормализация против пакетной на наборе данных MNIST 469
13.6.3. Оценивание новых точек данных в пакетно-нормализованных сетях 469
13.7. Перекрестная валидация посредством ранней остановки 470
Пример 13.14. Ранняя остановка и регрессия 470
Пример 13.15. Ранняя остановка и классификация рукописных цифр 470
13.8. Заключение 472
13.9. Упражнения 472
13.1. Двухклассовая классификация с использованием нейронных сетей 472
13.2. Многоклассовая классификация с использованием нейронных сетей 472

13.3. Число весов для усвоения в нейронной сети 472
13.4. Нелинейный автокодировщик с использованием нейронных сетей 473
13.5. Функция активации maxout 473
13.6. Сравнение продвинутых первопорядковых оптимизаторов I 473
13.7. Сравнение продвинутых первопорядковых оптимизаторов II 473
13.8. Пакетная нормализация 473
13.9. Перекрестная валидация на основе ранней остановки 473
13.10. Распознавание рукописных цифр с использованием нейронных сетей 473

Глава 14. Деревья решений 474
14.1. Введение 474
14.2. От пней до глубоких деревьев 474
14.2.1. Пень 474
14.2.2. Создание глубоких деревьев посредством рекурсии 475
14.2.3. Создание глубоких деревьев путем сложения 477
14.2.4. Интерпретируемость человеком 477
14.3. Регрессионные деревья 477
14.3.1. Определение оптимальной точки ветвления при фиксированных листовых значениях 478
14.3.2. Определение оптимальных листовых значений при фиксированной точке ветвления 478
14.3.3. Оптимизация регрессионных пней 481
Пример 14.1. Подгонка параметров простого регрессионного дерева 481
14.3.4. Более глубокие регрессионные деревья 482
Пример 14.2. Выращивание регрессионного дерева максимальной глубины 483
14.4. Классификационные деревья 484
14.4.1. Определение оптимальной точки ветвления при фиксированных листовых значениях 484
14.4.2. Определение оптимальных листовых значений при фиксированной точке ветвления 484
14.4.3. Оптимизация классификационных пней 488
Пример 14.3. Подгонка параметров простого классификационного дерева 489
14.4.4. Более глубокие классификационные деревья 489
Пример 14.4. Выращивание двух классификационных деревьев максимальной глубины 489
14.5. Бустирование градиента 490
14.5.1. Правило мелких деревьев 491
14.5.2. Бустирование с деревьями решений 491
Пример 14.5. Регрессия посредством бустирования градиента 492
Пример 14.6. Обнаружение спама посредством бустирования градиента 493
14.6. Случайные леса 494
Пример 14.7. Классификация на основе случайных лесов 495
14.7. Приемы перекрестной валидации для рекурсивно определенных деревьев 496
14.7.1. Ранняя остановка 496
Пример 14.8. Ранняя остановка по глубине и росту листьев 497
14.7.2. Подрезка 498
14.8. Заключение 499
14.9. Упражнения 500
14.1. Выращивание глубоких деревьев путем сложения 500
14.2. Подгонка параметров простого регрессионного дерева 500

14.3. Имплементировать регрессионное дерево в коде 500
14.4. Имплементировать двухклассовое классификационное дерево в коде 500
14.5. Имплементировать многоклассовое классификационное дерево в коде 500
14.6. Бустирование градиента для регрессии 501
14.7. Бустирование градиента для классификации 501
14.8. Случайные леса 501
14.9. Ограничение деревьев вне их тренировочного диапазона 501
14.10. Наивная перекрестная валидация 501

ЧАСТЬ IV. ПРИЛОЖЕНИЯ 503

Приложение 1. Продвинутые методы оптимизации первого и второго порядков 505
П1.1. Введение 505
П1.2. Стохастический градиентный спуск с импульсом 505
П1.2.1. Экспоненциальное сглаживание 505
П1.2.2. Смягчение зигзагообразного поведения градиентного спуска 508
Пример П1.1. Ускорение градиентного спуска в простой квадратичной функции 509
П1.3. Нормализованный градиентный спуск 511
П1.3.1. Нормализация модуля полного градиента 511
Пример П1.2. Смягчение медленного ползания вблизи минимумов и седловых точек 512
Пример П1.3. Компромисс при использовании нормализованного градиентного спуска 513
П1.3.2. Нормализация модуля покомпонентно 514
Пример П1.4. Полный и покомпонентно-нормализованный градиентные спуски 516
П1.4. Продвинутые градиентные методы 517
Пример П1.5. Адаптивное оценивание импульса (Adam) 518
Пример П1.6. Распространение ошибки корня из среднеквадратической ошибки (RMSProp) 519
П1.5. Мини-пакетная оптимизация 520
П1.5.1. Простая идея с мощными следствиями 520
П1.5.2. Спуск с более крупными размерами мини-пакета 521
П1.5.3. Общая результативность 522
П1.6. Консервативные правила длины шага 523
П1.6.1. Градиентный спуск и простые квадратичные функции-заменители 523
П1.6.2. Линейный поиск с отслеживанием в обратном порядке 525
П1.6.3. Точный линейный поиск 527
П1.6.4. Консервативно оптимальные фиксированные значения длины шага 528
Пример П1.7. Вычисление липшицевой константы одновыходной синусоиды 529
Пример П1.8. Вычисление липшицевой константы многовходной квадратичной функции 530
П1.6.5. Доказательства сходимости 530
Градиентный спуск с фиксированной длиной шага по Липшицу 530
Градиентный спуск с использованием линейного поиска с отслеживанием в обратном порядке 531
П1.7. Метод Ньютона, регуляризация и невыпуклые функции 532
П1.7.1. Поднятие ε 532
Пример П1.9. Эффект регуляризации 533
Пример П1.10. Минимизация невыпуклой функции 534
П1.8. Безгессиановы методы 535
П1.8.1. Взятие подвыборок из гессиана 536
П1.8.2. Методы секущей 537

П1.8.3. Квазиньютоновы методы 539
Пример П1.11. Разностная матрица ранга 1 540
Пример П1.12. Метод Дэвидона — Флетчера — Пауэлла (Davidon — Fletcher — Powell, DFP) 541
П1.8.4. Квазиньютоновы методы с низким потреблением памяти 542

Приложение 2. Производные и автоматическое дифференцирование 544
П2.1. Введение 544
П2.2. Производная 544
П2.2.1. Секущие и касательные 544
П2.2.2. Численное дифференцирование 546
П2.3. Правила производных для элементарных функций и операций 547
Пример П2.1. Производная от общих одночленов 548
Пример П2.2. Правило произведения 549
П2.4. Градиент 549
П2.5. Вычислительный граф 550
Пример П2.3. Вычислительный граф одновходной функции 551
Пример П2.4. Вычислительный граф многовходной функции 552
П2.6. Прямой режим автоматического дифференцирования 553
Пример П2.5. Дифференцирование одновходной функции в прямом режиме 554
Пример П2.6. Дифференцирование многовходной функции в прямом режиме 557
П2.7. Обратный режим автоматического дифференцирования 559
Пример П2.7. Обратное дфференцирование многовходной функции 560
П2.8. Производные более высокого порядка 563
П2.8.1. Производные более высокого порядка одновходных функций 563
Пример П2.8. Производные более высокого порядка 563
П2.8.2. Производные более высокого порядка многовходных функций 564
П2.9. Ряд Тейлора 565
П2.9.1. Линейная аппроксимация — это только начало 565
П2.9.2. От касательной линии к касательной квадратичной функции 567
П2.9.3. Построение более качественных локальных аппроксимаций 568
П2.9.4. Многовходной ряд Тейлора 569
П2.10. Использование библиотеки autograd 571
П2.10.1. Установка библиотеки autograd 571
П2.10.2. Использование autograd 571
Пример П2.9. Вычисление производных одновходных функций 571
Пример П2.10. Вычисление функции и градиента 573
Пример П2.11. Вычисление аппроксимаций с использованием рядов Тейлора 573
Пример П2.12. Вычисление индивидуальных частных производных 575
Пример П2.13. Вычисление нескольких производных или полного градиента 576
П2.10.3. Выравнивание математических функций с помощью autograd 577
Пример П2.14. Выравнивание многовходной функции 577

Приложение 3. Линейная алгебра 580
П3.1. Введение 580
П3.2. Векторы и векторные операции 580
П3.2.1. Вектор 580
П3.2.2. Сложение векторов 581
П3.2.3. Умножение векторов 582
Умножение вектора на скаляр 583
Поэлементное произведение двух векторов 583

Внутреннее произведение двух векторов ... 583
Длина или норма вектора ... 583
Геометрическая интерпретация внутреннего произведения ... 584
Внешнее произведение двух векторов ... 585
П3.2.4. Линейная комбинация векторов ... 585
П3.3. Матрицы и матричные операции ... 587
П3.3.1. Матрица ... 587
П3.3.2. Сложение матриц ... 588
П3.3.3. Умножение матриц ... 588
Умножение матрицы на скаляр ... 588
Умножение матрицы на вектор ... 588
Поэлементное умножение двух матриц ... 589
Общее умножение двух матриц ... 589
П3.4. Собственные значения и собственные векторы ... 590
П3.4.1. Линейные функции и матричное умножение ... 590
П3.4.2. Линейные функции и квадратные матрицы ... 590
П3.4.3. Собственные числа и собственные векторы ... 591
П3.4.4. Частный случай симметричной матрицы ... 592
П3.5. Векторные и матричные нормы ... 593
П3.5.1. Векторные нормы ... 593
ℓ_2-норма ... 593
ℓ_1-норма ... 594
ℓ_∞-норма ... 594
П3.5.2. Общие свойства векторных норм ... 594
Шары ℓ_p-нормы ... 595
ℓ_0-норма ... 595
П3.5.3. Матричные нормы ... 596
Норма Фробениуса ... 596
Спектральные и ядерные нормы ... 596

Список литературы ... 598

Предметный указатель ... 634

Посвящается нашим семьям:

Дебе, Роберту и Терри
Сойле, Али и Марьям
Ειρηνη, Ζωη, Σοφια и Ειρηνη

О книге

Благодаря интуитивно понятному, но строгому подходу к машинному обучению эта книга предоставляет учащимся фундаментальные знания и практические инструменты, необходимые для проведения исследований и создания продуктов, основанных на данных. Авторы отдают приоритет геометрической интуиции и алгоритмическому мышлению, а также подробно описывают все основные математические предпосылки, чтобы предложить свежий и доступный способ обучения. Особое внимание уделяется практическим приложениям с примерами из таких дисциплин, как компьютерное зрение, обработка естественного языка, экономика, нейробиология, рекомендательные системы, физика и биология. Книга содержит более 300 цветных иллюстраций[1], которые были тщательно обработаны с целью обеспечения интуитивного понимания технических концепций, а более 100 углубленных упражнений на языке Python обеспечивают реальное понимание важнейших алгоритмов машинного обучения. Набор онлайновых ресурсов, включая примеры кода, наборы данных, интерактивные слайды лекций и руководство по решениям задач, предоставлены онлайн, что делает эту книгу идеальной как для специализированных курсов по машинному обучению, так и для индивидуального справочного и самостоятельного изучения.

Джереми Уотт (Jeremy Watt) получил докторскую степень по электротехнике в Северо-Западном университете, шт. Иллинойс, США, и теперь является консультантом и преподавателем машинного обучения. Он преподает машинное обучение, глубокое обучение, математическую оптимизацию и подкрепляемое обучение в Северо-Западном университете.

Реза Борхани (Reza Borhani) получил докторскую степень по электротехнике в Северо-Западном университете, шт. Иллинойс, США, и теперь является консультантом и преподавателем машинного обучения. Он преподает различные курсы по машинному обучению и глубокому обучению в Северо-Западном университете.

Аггелос К. Кацаггелос (Aggelos K. Katsaggelos) — профессор в Северо-Западном университете, шт. Иллинойс, США, где он возглавляет лабораторию обработки изображений и видео. Он является членом IEEE, SPIE, EURASIP и OSA и лауреатом медали IEEE Third Millennium Medal (2000).

---

[1] Коды всех примеров и более 300 цветных иллюстраций можно скачать по ссылке **ftp://ftp.bhv.ru/9785977567633.zip**, а также со страницы книги на сайте **bhv.ru**.

Предисловие

На протяжении многих эпох мы, люди, искали правила или шаблоны, которые точно описывали бы то, как работают важные системы в окружающем нас мире, будь то сельскохозяйственные, биологические, физические, финансовые и т. д. Мы делаем это потому, что такие правила позволяют нам лучше понять систему, точно предсказать ее будущее поведение и в конечном счете контролировать ее. Однако задача отыскания "верного" правила, которое, по всей видимости, управляет данной системой, исторически была нелегкой. В течение большей части нашей истории данные (т. е. проблески этой системы в действии) были чрезвычайно редким товаром. Более того, наша способность вычислять, пробовать различные правила, чтобы увидеть, что наиболее точно представляет явление, была ограничена тем, что мы могли бы сделать вручную. Оба эти фактора естественным образом ограничивали круг явлений, которые научные первопроходцы прошлого могли исследовать, и неизбежно вынуждали их использовать философские и/или визуальные подходы к формированию правил. Однако сегодня мы живем в мире, изобилующем данными, и у нас под рукой колоссальные вычислительные мощности. Благодаря этому мы, счастливые потомки великих первопроходцев, можем решать гораздо более широкий круг задач и применять гораздо более эмпирический подход к отысканию правил, чем наши предшественники. Термин "машинное обучение"[1], тема этой книги, используется для описания широкого набора алгоритмов для поиска закономерностей, предназначенных для правильного эмпирического выявления системных правил как эмпирически, так и путем задействования нашего доступа к потенциально безмерным объемам данных и вычислительной мощности.

За последнее десятилетие пользовательская база машинного обучения значительно выросла. Из относительно небольшого круга специалистов в области компьютерных наук, инженерии и математики пользователями машинного обучения теперь являются студенты и исследователи из всех уголков академической Вселенной, а также представители промышленности, исследователи данных, предприниматели и энтузиасты машинного обучения. Эта книга является результатом разбиения стандартной учебной программы машинного обучения на ее наиболее фундаментальные компоненты и тщательного выбора тех, которые, по нашему мнению, принесут

---

[1] Термины "обучение" и "усвоение" в настоящем переводе используются взаимозаменяемо. Соответствующий английский термин learning означает усвоение учеником знаний, заучивание, обучение, а применительно к машинному обучению (machine learning) — автоматическое усвоение компьютерной моделью регулярностей/шаблонов/паттернов в данных. — *Прим. пер.*

наибольшую пользу этой расширяющейся аудитории учащихся. Она содержит обновленные и интуитивно понятные, но точные описания самых фундаментальных понятий, необходимых для проведения исследований, создания продуктов и работы с ними.

Обзор книги

Второе издание этой книги представляет собой полную переработку нашей первой попытки, причем практически каждая глава оригинала переписана с нуля, и было добавлено восемь новых глав материала, что удваивает размер первого издания. Темы из первого издания, начиная с изложений о градиентном спуске и заканчивая классификацией "один против всех" и анализом главных компонент, были переработаны и отшлифованы. Был добавлен ряд новых тем, от беспроизводной оптимизации до взвешенного обучения с учителем, отбора признаков, инженерии нелинейных признаков, перекрестной валидации на основе бустирования и многого другого.

Несмотря на то что наша первоначальная попытка была масштабнее, ее цель осталась неизменной: объяснить машинное обучение, от первых принципов до практической имплементации, в самых простых терминах. Далее приведена общая разбивка текста второго издания.

Часть I: математическая оптимизация (главы 2–4)

Математическая оптимизация — это рабочая лошадка машинного обучения, обеспечивающая не только настройку конкретных моделей машинного обучения (представленных в *части II*), но и каркас, посредством которого мы сами определяем соответствующие модели с помощью перекрестной валидации (обсуждаемой в *части III*).

В этой части книги мы даем полное введение в математическую оптимизацию, от базовых методов нулевого порядка, подробно описанных в *главе 2*, до фундаментальных и продвинутых методов первого и второго порядков соответственно в *главах 3* и *4*. В частности, эта часть книги содержит полные описания методов локальной оптимизации, случайного поиска, градиентного спуска и метода Ньютона.

Часть II: линейное обучение (главы 5–9)

В этой части книги мы опишем фундаментальные компоненты машинного обучения на основе функции потерь с акцентом на линейные модели.

Сюда входит полное описание обучения с учителем в *главах 5–7*, включая линейную регрессию, двухклассовую и многоклассовую классификации. В каждой из этих глав мы опишем ряд точек зрения и популярных вариантов решений, которые используются при таком способе машинного обучения.

В *главе 8* мы аналогично описываем обучение без учителя, а в *главе 9* содержится введение в фундаментальные методы генерации признаков, включая популярные

гистограммные признаки, а также различные варианты нормализации в отношении входных данных, и правила отбора признаков.

Часть III: нелинейное обучение (главы 10–14)

В заключительной части книги мы расширим фундаментальные понятия, введенные в *части II*, на общую нелинейную составляющую.

Мы сделаем это осторожно, начиная с базового введения в нелинейное обучение с учителем и без учителя в *главе 10*, в которой введем мотивацию, общую терминологию и нотацию нелинейного самообучения, используемые в остальной части книги.

В *главе 11* мы обсудим тему автоматизированного отбора подходящих нелинейных моделей, начиная с введения в универсальную аппроксимацию. Это, естественно, приводит к подробным описаниям перекрестной валидации, а также бустирования, регуляризации, ансамблирования и K-fold перекрестной валидации.

Имея на руках эти фундаментальные идеи, *главы 12–14* мы посвятим популярным универсальным аппроксиматорам, используемым в машинном обучении: фиксированно-контурным ядрам, нейронным сетям и деревьям, где мы обсудим сильные и слабые стороны, технические особенности и применение каждого популярного универсального аппроксиматора.

Для того чтобы извлечь максимальную пользу из этой части книги, мы настоятельно рекомендуем изучить и понять *главу 11* и содержащиеся в ней фундаментальные идеи, прежде чем переходить к *главам 12–14*.

Часть IV: приложения

В этой более короткой коллекции дополнительных материалов дается полная трактовка продвинутых методов оптимизации, а также подробное введение в ряд тем, которые читатели должны будут понять, чтобы в полной мере использовать книгу.

Приложение 1 продолжает наше обсуждение, начатое в *главах 3* и *4*, и описывает передовые приемы оптимизации первого и второго порядков. Сюда входит обсуждение популярных расширений градиентного спуска, включая мини-пакетную оптимизацию, оптимизатор импульса, нормализацию градиента и результат совмещения этих улучшений различными способами (например, алгоритмы первого порядка RMSProp и Adam), и метод Ньютона, включая регуляризационные схемы и безгессиановы методы.

Приложение 2 содержит экскурс в вычислительное исчисление, включающий введение в производную/градиент, производные более высокого порядка, матрицу Гессе, численное дифференцирование, прямое и обратное (на основе обратного распространения) автоматическое дифференцирование и аппроксимации с использованием ряда Тейлора.

Приложение 3 содержит соответствующую справку по линейной и матричной алгебре, включая векторную/матричную арифметику, понятия охватывающих множеств и ортогональности, а также собственные значения и собственные векторы.

К читателям: как пользоваться этой книгой

Эта книга была написана с учетом учащихся, которые только начинают изучать данный предмет, а также для более знающих читателей, которые жаждут более интуитивной и полезной трактовки, чем то, что имеется сегодня. Для того чтобы использовать книгу в полной мере, требуется только базовое понимание векторной алгебры (математических функций, векторной арифметики и т. д.) и компьютерного программирования (например, базового владения языком с динамической типизацией, таким как Python). Мы предоставляем полные вводные трактовки других обязательных тем, включая линейную алгебру, векторное исчисление и автоматическое дифференцирование в приложениях к книге. Примерные "дорожные карты", показанные на рис. B1–B4, предоставляют предлагаемые пути для навигации по тексту книги на основе большого разнообразия исходов обучения и университетских курсов (от курса по основным элементам машинного обучения до специальных тем, как описано в *разд. "К преподавателям: как пользоваться этой книгой" далее*).

Мы считаем, что интуитивные скачки осознания предшествуют интеллектуальным, и с этой целью уступаем использование вероятностных и статистических взглядов на машинное обучение в пользу новой и последовательной геометрической точки зрения во всей книге. Мы полагаем, что эта точка зрения не только позволяет интуитивнее усваивать отдельные понятия из книги, но и помогает устанавливать явные связи между идеями, часто рассматриваемыми как принципиально разные (например, логистическо-регрессионные классификаторы и машины опорных векторов, ядра и полносвязные нейронные сети и т. д.). Мы также очень подчеркиваем важность математической оптимизации в нашем подходе к машинному обучению. Как подробно описано в *разд. "Обзор книги" ранее*, оптимизация служит рабочей лошадкой машинного обучения и является фундаментальной на многих уровнях — от тонкой настройки отдельных моделей до общего отбора надлежащих нелинейностей посредством перекрестной валидации. Именно поэтому необходимо глубокое понимание математической оптимизации, если имеется желание досконально изучить машинное обучение и иметь возможность реализовывать фундаментальные алгоритмы.

С этой целью мы уделяем значительное внимание архитектурному дизайну и реализации алгоритмов во всей книге с использованием реализаций фундаментальных алгоритмов, приводимых на языке Python. Эти основополагающие примеры в дальнейшем можно использовать в качестве строительных блоков, чтобы помочь читателю выполнить упражнения по программированию данной книги, предоставляя ему возможность "выполнить черновую работу" и "научиться на практике", воплощая в жизнь идеи, высказанные в основной части книги. Хотя для выполнения

упражнений по кодированию в данной книге в принципе можно использовать любой язык программирования, мы настоятельно рекомендуем взять Python ввиду его простоты применения и поддержки со стороны сообщества. Мы рекомендуем использовать библиотеки Python с открытым исходным кодом NumPy, autograd и matplotlib, а также редактор Jupyter, чтобы упростить написание и тестирование кода. Полный набор инструкций по инсталляции библиотек, скачиванию наборов данных, а также использованию стартовых блокнотов для многих упражнений можно найти на веб-сайте

https://github.com/jermwatt/machine_learning_refined

К преподавателям: как пользоваться этой книгой

Визуализации к главам, связанные с этой книгой, наборы данных, а также большой массив учебных интерактивных виджетов Python, иллюстрирующих различные концепции на протяжении всей книги, можно найти в репозитории github, сопровождающем эту книгу, по адресу:

https://github.com/jermwatt/machine_learning_refined

Этот веб-сайт содержит инструкции по инсталляции языка Python, а также ряд других бесплатных программных библиотек, которые студенты найдут полезными при выполнении упражнений книги.

Эта книга легла в основу ряда курсов машинного обучения в Северо-Западном университете, шт. Иллинойс, США, начиная с вводных курсов, подходящих для студентов старших курсов, и заканчивая более продвинутыми курсами по специальным темам, посвященным оптимизации и глубокому обучению для аспирантов. С учетом изложения основ, приложений и алгоритмов эта книга может быть использована в качестве первичного ресурса или фундаментального компонента для таких курсов, как перечисленные ниже.

Основы машинного обучения: введение в основные элементы машинного обучения идеально подходит для бакалавриата и университетов, где предмет изучается в течение семестра или где глубокое погружение в содержание книги невозможно из-за нехватки времени. Темы для такого курса могут включать градиентный спуск, логистическую регрессию, метод опорных векторов, логистическую регрессию "один-все" и многоклассовую логистическую регрессию, метод главных компонент, кластеризацию на основе K средних, основные элементы инженерии и отбора признаков, перекрестную валидацию, регуляризацию, ансамблирование, бутстрап-агрегирование (бэггинг), ядерные методы, полносвязные нейронные сети и деревья. Рекомендуемая дорожная карта для такого курса, включая соответствующие главы, разделы и темы, показана на рис. B1.

Полная трактовка машинного обучения: стандартный курс по машинному обучению, основанный на этой книге, расширяет описанный выше курс основных элементов как с точки зрения широты, так и глубины. В дополнение к темам, упомянутым в курсе основных элементов, преподаватели могут выбрать метод Ньютона,

наименьшие абсолютные отклонения, многовыходную регрессию, взвешенную регрессию, персептрон, категориальную функцию потерь, взвешенную двухклассовую и многоклассовую классификации, динамическое обучение, рекомендательные системы, методы матричной факторизации, отбор признаков на основе бустирования, универсальную аппроксимацию, бустирование градиента, случайные леса, а также более углубленную обработку полносвязных нейронных сетей, включающую такие темы, как пакетная нормализация и регуляризация на основе ранней остановки. Рекомендуемая дорожная карта для такого курса, включая соответствующие главы, разделы и темы, проиллюстрирована на рис. B2.

Математическая оптимизация в машинном обучении и глубоком обучении: такой курс влечет за собой всестороннее описание приемов оптимизации нулевого, первого и второго порядков из *части I* книги (а также *приложения 1*), включая координатный спуск, градиентный спуск, метод Ньютона, квазиньютоновы методы, стохастическую оптимизацию, оптимизатор импульса, правила с фиксированным и адаптивным размерами шага, а также продвинутые схемы с нормализованным градиентным спуском (например, Adam и RMSProp). За ними может последовать подробное описание процессов конструирования признаков (в особенности стандартной нормализации и PCA-сферизации), которые ускоряют оптимизационные алгоритмы (в особенности алгоритмы первого порядка). Все студенты в общем, и те, кто проходит курс по оптимизации для машинного обучения, в частности должны оценить фундаментальную роль оптимизации в выявлении "правильной" нелинейности посредством процессов перекрестной валидации на основе бустирования и регуляризации, принципы которой рассматриваются в *главе 11*. Выборочные темы из *главы 13* и *приложения 2*, в том числе обратное распространение, пакетная нормализация и прямой/обратный режим автоматического дифференцирования, также могут быть охвачены. Рекомендуемая дорожная карта для такого курса, включая соответствующие главы, разделы и темы, приведена на рис. B3.

Вводная часть курса по глубокому обучению: такой курс лучше всего подходит для студентов, которые ранее сталкивались с фундаментальными понятиями машинного обучения, и может начинаться с обсуждения соответствующих методов оптимизации первого порядка с акцентом на стохастическую и мини-пакетную оптимизацию, оптимизатор импульса и схемы нормализованного градиента, такие как Adam и RMSProp. В зависимости от аудитории может потребоваться краткий обзор основных элементов машинного обучения с использованием выборочных порций *части II* книги. Законченное обсуждение полносвязных сетей, включая обсуждение обратного распространения и прямого/обратного режима автоматического дифференцирования, а также специальные темы, такие как пакетная нормализация, перекрестная валидация на основе ранней остановки, можно затем выполнить с помощью *глав 11, 13* и *приложений 1, 2*. Рекомендуемая дорожная карта для такого курса, включая соответствующие главы, разделы и темы, показана на рис. B4.

Дополнительные рекомендуемые ресурсы по темам для завершения стандартного курса по глубокому обучению — например, сверточные и рекуррентные сети — можно найти, посетив репозиторий книги на github.

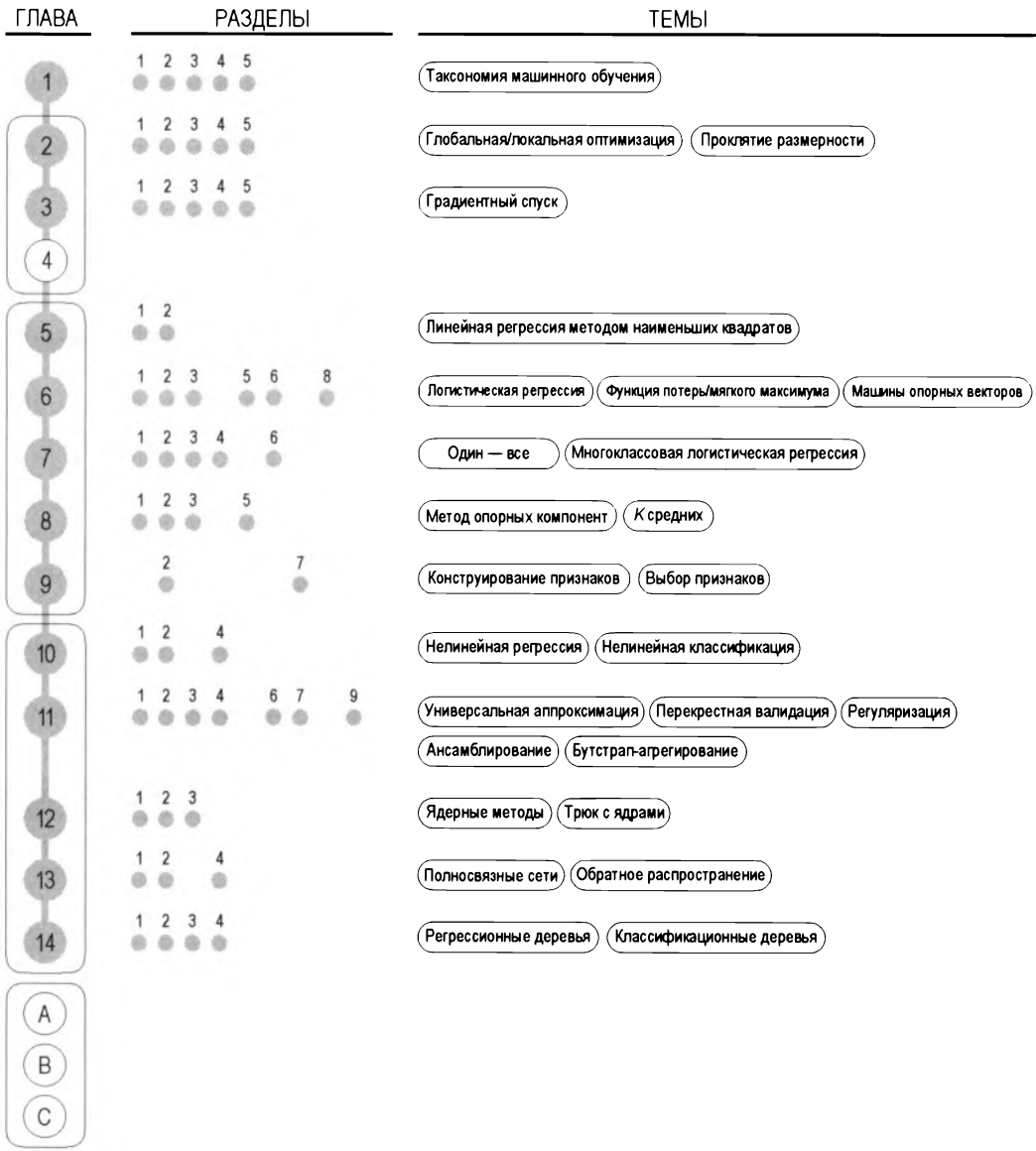


Рис. В1. Рекомендуемая учебная дорожная карта для курса по основным элементам машинного обучения, включающая необходимые главы (*слева*), разделы (*в центре*) и соответствующие темы (*справа*). Этот план основных элементов подходит для ограниченных по времени курсов (в семестровых программах и университетах), или для самостоятельного изучения, или там, где машинное обучение не является единственным фокусом внимания, но ключевым компонентом какого-то более широкого курса обучения. Обратите внимание, что главы сгруппированы визуально на основе разбивки текста книги, подробно описанной в *разд. "Обзор книги"* в *предисловии*

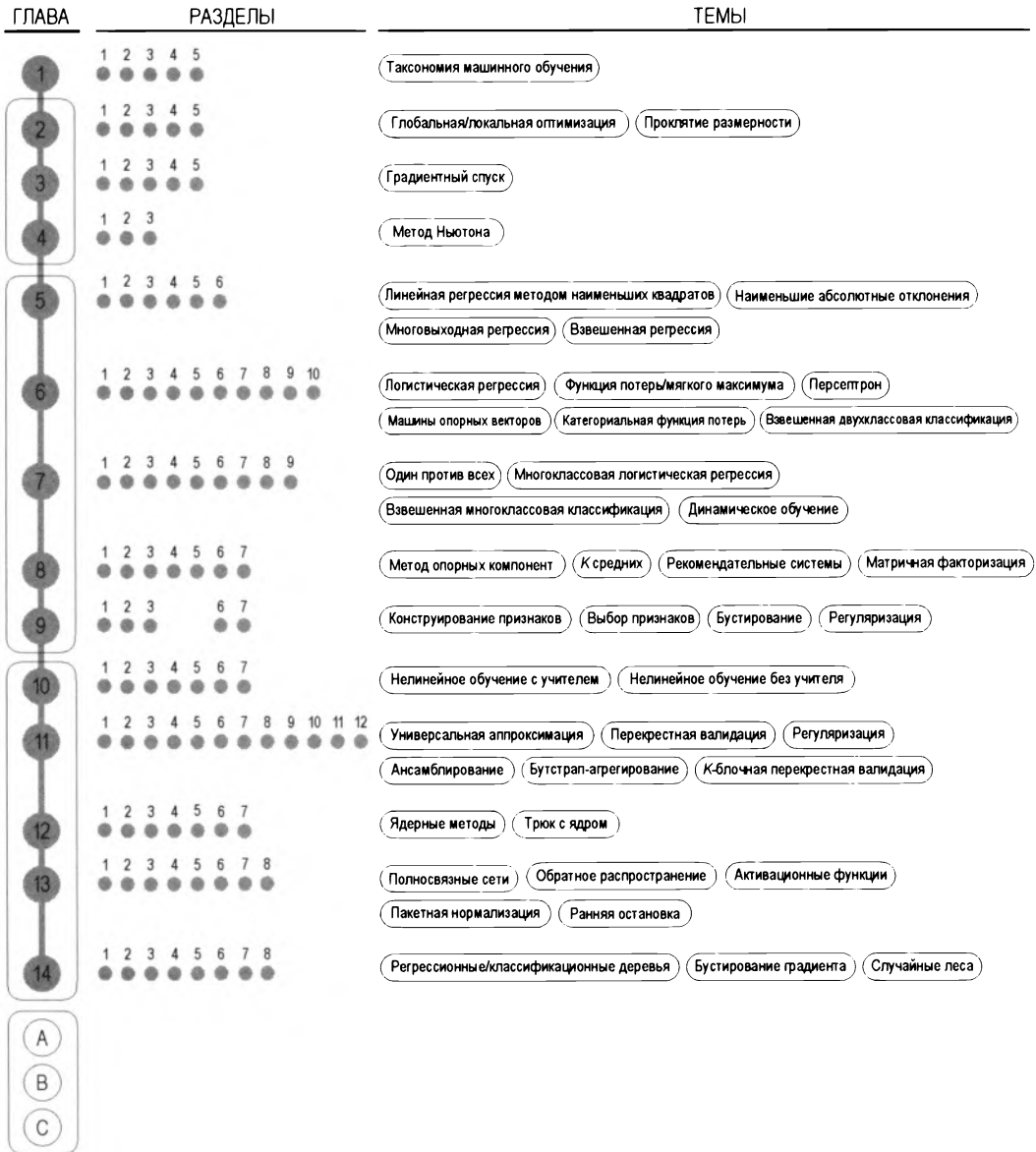


Рис. B2. Рекомендуемая учебная дорожная карта для полной трактовки стандартных предметов машинного обучения, включая главы, разделы, а также соответствующие темы для изучения. Этот план предполагает более глубокое освещение тем машинного обучения по сравнению с дорожной картой основных элементов, приведенной на рис. B1, и лучше всего подходит для студентов старших курсов/начинающих аспирантов в семестровых программах и увлеченных независимых читателей.

Подробности *см. в разд. "К преподавателям: как пользоваться этой книгой"* в *предисловии*

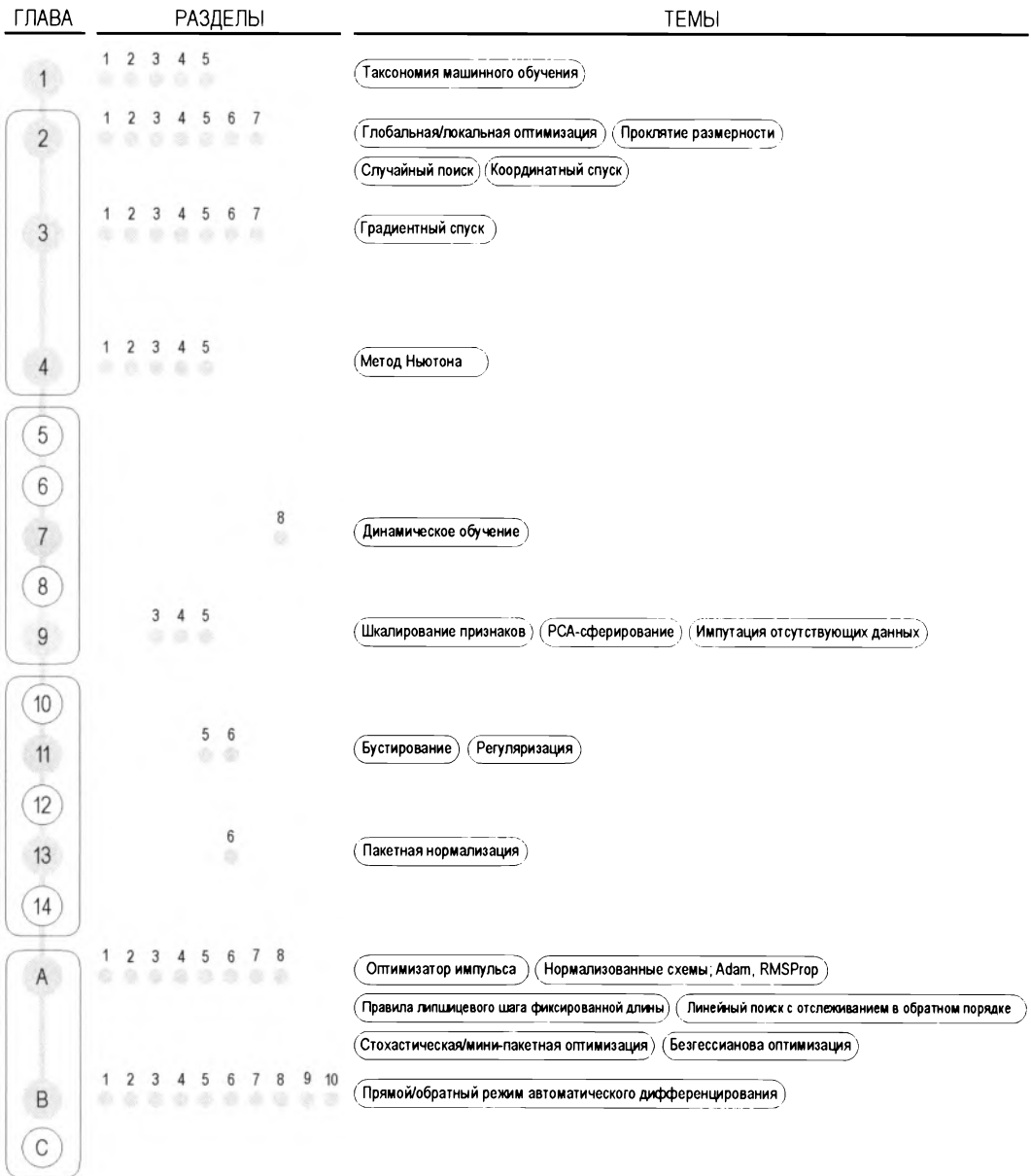


Рис. В3. Рекомендуемая учебная дорожная карта для курса по математической оптимизации для машинного обучения и глубокого обучения, включающая главы, разделы, а также темы для изучения. Подробности *см. в разд. "К преподавателям: как пользоваться этой книгой"* в *предисловии*

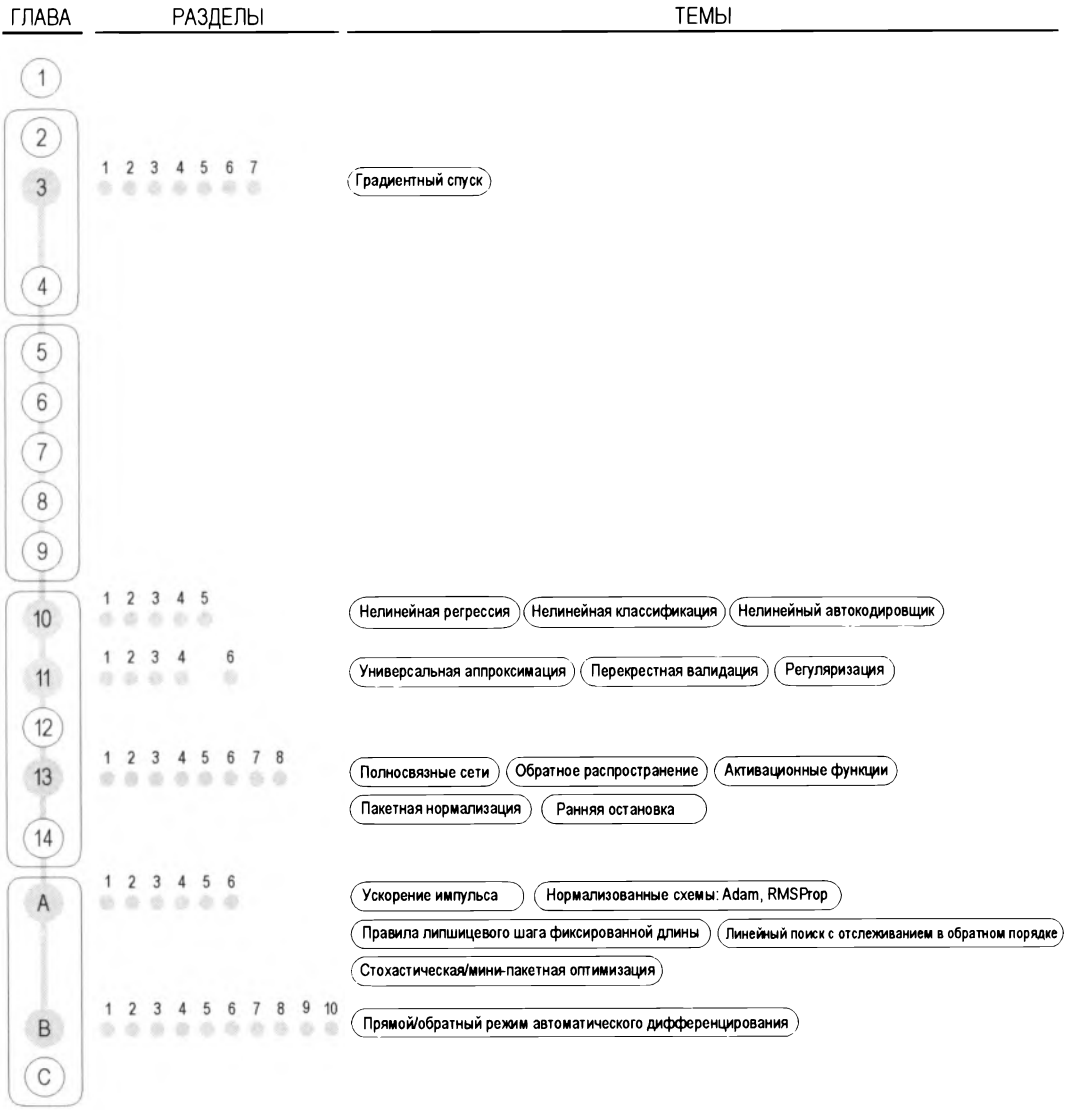


Рис. B4. Рекомендуемая учебная дорожная карта для вводной части курса по глубокому обучению, включающая главы, разделы, а также темы для изучения. Подробности *см. в разд. "К преподавателям: как пользоваться этой книгой"* в *предисловии*

Благодарности

Эта книга не была бы написана даже близко в той форме, которую она имеет сегодня, без огромной работы бесчисленных гениев-ангелов в сообществе Python-поддержки открытого исходного кода, в особенности авторов и соавторов библиотек NumPy, Jupyter и matplotlib. Мы особенно благодарны авторам и соавторам библиотеки autograd, включая Дугала Маклорина (Dougal Maclaurin), Дэвида Дювено (David Duvenaud), Мэтта Джонсона (Matt Johnson) и Джейми Таунсенда (Jamie Townsend), поскольку autograd позволил нам экспериментировать и неоднократно тестировать массу новых идей, включенных во второе издание этой книги, что значительно улучшило его, а также, мы надеемся, учебный опыт для ее читателей.

Мы также очень благодарны многим студентам, которые за прошедшие годы предоставили содержательные отзывы об этой книге, и в особенности благодарны Боуэну Тяню (Bowen Tian), который предоставил большое количество глубоких отзывов в отношении ранних черновиков книги.

Наконец, большое спасибо Марку Макнессу Розенгрену (Mark McNess Rosengren) и всей команде сети кофейн Standing Passengers за то, что они помогали нам оставаться накофеиненными во время написания этой книги.

1 Введение в машинное обучение

1.1. Введение

Машинное обучение — это единый алгоритмический каркас, предназначенный для выявления вычислительных моделей, которые точно описывают эмпирические данные и явления, лежащие в их основе, практически без участия человека. Будучи еще молодой дисциплиной с гораздо бóльшим числом предстоящих открытий, чем известно в настоящее время, сегодня машинное обучение можно использовать для обучения компьютеров выполнять широкий спектр полезных задач, включая автоматическое обнаружение объектов на снимках (важнейший компонент в системах распознавания водителя за рулем автомобиля), распознавание речи (которое приводит в действие технологию голосовых команд), новые знания в области медицины (используемые улучшения диагностики сложных заболеваний) и предсказательную аналитику (используемую для продаж и экономического прогнозирования), и это лишь некоторые из них.

В этой главе мы даем высокоуровневое введение в область машинного обучения, а также содержание этой книги.

1.2. Как отличить кошку от собаки: подход на основе машинного обучения

Для того чтобы получить общее представление о том, как работает машинное обучение, мы начнем с обсуждения модельной задачи: научим компьютер отличать снимки кошек от снимков собак. Это позволит нам неформально описать терминологию и процедуры, связанные с решением типичной задачи машинного обучения.

Помните ли вы, как вы впервые узнали о разнице между кошками и собаками, и о том, что они являются разными животными? Ответ, вероятно, будет отрицательным, поскольку большинство людей учатся выполнять простые когнитивные задачи, подобные этой, в очень раннем возрасте. Однако ясно одно: маленькие дети не нуждаются в какой-то формальной научной подготовке или в зоологической лекции о видах *felis catus* (домашних кошек) и *canis familiaris* (домашних собак), чтобы уметь различать кошек и собак. Вместо этого они учатся на примере. Им в естественных условиях показывают многочисленные изображения тех, кто по словам их наставника (родителя, воспитателя и т. д.) является либо кошками, либо

собаками, до тех пор, пока дети не поймут эти два понятия. Как узнать, когда ребенок уже способен успешно различать кошек и собак? Интуитивно это можно сделать, когда он сталкивается с новыми (изображениями) кошек и собак и может правильно идентифицировать каждый новый пример или, другими словами, когда он может *обобщить* то, что узнал, на новые, ранее не встречавшиеся примеры.

Аналогично людям, компьютеры можно научить тому, как выполнять подобного рода задачи. Такого рода задача, когда мы стремимся научить компьютер различать разные типы или классы вещей (здесь кошек и собак), на жаргоне машинного обучения называется *задачей классификации* и выполняется с помощью ряда шагов, которые мы подробно рассмотрим ниже.

1. **Сбор данных.** Подобно человеку, компьютер должен быть натренирован распознавать разницу между этими двумя родами животных, учась на серии примеров, обычно именуемом *тренировочным набором* данных. На рис. 1.1 показан такой тренировочный набор, состоящий из нескольких снимков разных кошек и собак. Интуитивно понятно, что чем больше и разнообразнее тренировочный набор, тем лучше компьютер (или человек) способен выполнять учебную задачу, поскольку, находясь под воздействием более широкого спектра примеров, ученик получает больше опыта.



Рис. 1.1. Тренировочный набор, состоящий из шести снимков кошек (*сверху*) и шести снимков собак (*снизу*). Этот набор используется для тренировки модели машинного обучения, которая может различать будущие снимки кошек и собак. Снимки на этом рисунке были взяты из [1]

2. **Конструирование признаков.** Задумайтесь на мгновение о том, как мы (люди) отличаем снимки, содержащие кошек, от снимков, содержащих собак. Для того чтобы их различать, мы используем цвет, размер, форму ушей или носа и/или некоторое сочетание этих признаков. Другими словами, мы не просто рассматриваем снимок как коллекцию из большого числа маленьких квадратных пикселов. Мы выбираем из снимков более грубые детали или признаки, подобные этим, чтобы идентифицировать то, на что мы смотрим. То же справедливо и для компьютеров. Для того чтобы успешно натренировать компьютер выполнять эту задачу (и любую задачу машинного обучения в целом), мы должны обеспечить его правильно сконструированными признаками или в идеале заставить его отыскивать, вырабатывать или усваивать такие признаки самостоятельно.

Задача конструирования качественных признаков, как правило, не является тривиальной, поскольку она может очень зависеть от приложения. Например, такой признак, как цвет, был бы менее полезным для различения кошек и собак (поскольку многие кошки и собаки имеют одинаковый окрас), чем для различения медведей гризли и белых медведей! Задача извлечения признаков из тренировочного набора данных также может быть трудной. Например, если бы некоторые из наших тренировочных снимков были размыты или взяты с ракурса, в котором мы не могли видеть животное должным образом, то признаки, которые мы сконструировали бы, возможно, не были бы извлечены надлежащим образом.

Однако ради простоты нашей игровой задачи предположим, что мы можем легко извлечь следующие два признака из каждого снимка в тренировочном наборе: *размер носа* относительно размера головы, варьирующийся от малого до крупного, и *форму ушей*, изменяющуюся от круглой до заостренной.

Рассматривая тренировочные снимки, показанные на рис. 1.1, мы видим, что у всех кошек маленькие носы и заостренные уши, в то время как у собак обычно крупные носы и круглые уши. Обратите внимание, что с текущей подборкой признаков каждый снимок теперь может быть представлен только двумя числами: числом, выражающим относительный размер носа, и еще одним числом, отражающим остроту или округлость ушей. Другими словами, мы можем представить каждый снимок нашего тренировочного набора в двумерном *пространстве признаков*, где размер носа и форма ушей являются соответственно горизонтальной и вертикальной координатными осями (рис. 1.2).

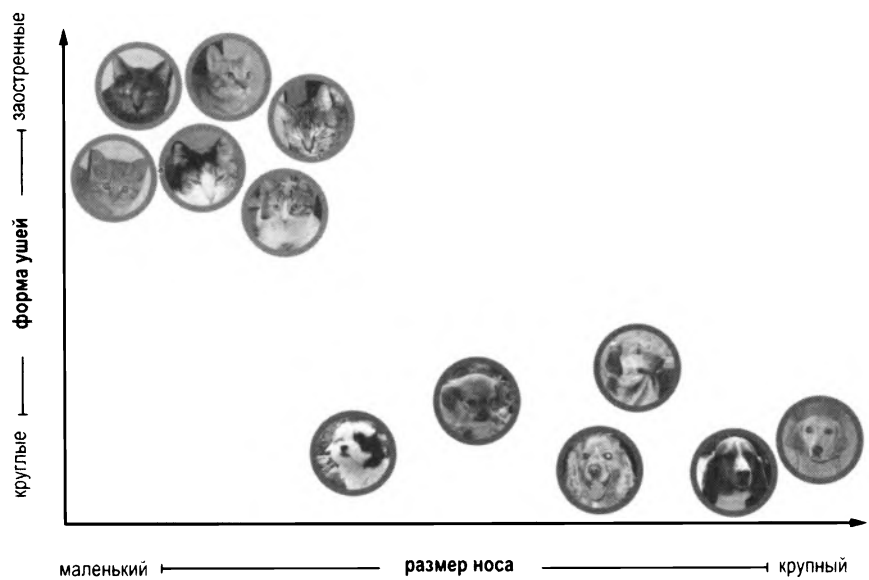


Рис. 1.2. Представление тренировочного набора, показанного на рис. 1.1, в форме пространства признаков, где горизонтальная и вертикальная оси представляют признаки соответственно размера носа и формы ушей. Тот факт, что изображения кошек и собак из нашего тренировочного набора расположены в разных участках пространства признаков, означает, что у нас признаки выбраны удачно

3. **Тренировка модели.** С нашим признаковым представлением тренировочных данных задача машинного обучения по различению кошек и собак теперь становится простой геометрической задачей: машина должна отыскать прямую или кривую, которая отделяет кошек от собак в нашем тщательно сформулированном пространстве признаков. Ради простоты предположив, что мы используем прямую, мы должны найти правильные значения для двух ее параметров — наклона и вертикального пересечения, которые определяют ориентацию прямой в пространстве признаков. Процесс определения надлежащих параметров опирается на набор инструментов, именуемый *математической оптимизацией*, подробно описанный в *главах 2–4*, и тонкая настройка такого набора параметров под тренировочный набор называется *тренировкой модели*.

На рис. 1.3 показана натренированная линейная модель (наклонная прямая линия), которая делит пространство признаков на участки кошек и собак. Эта линейная модель обеспечивает простое вычислительное правило для различения кошек и собак: когда признаковое представление будущего снимка лежит выше прямой (в левом верхнем углу рисунка), машина будет считать его кошкой, и точно так же любое представление, которое оказывается ниже прямой (в правом нижнем углу рисунка), будет считаться собакой.

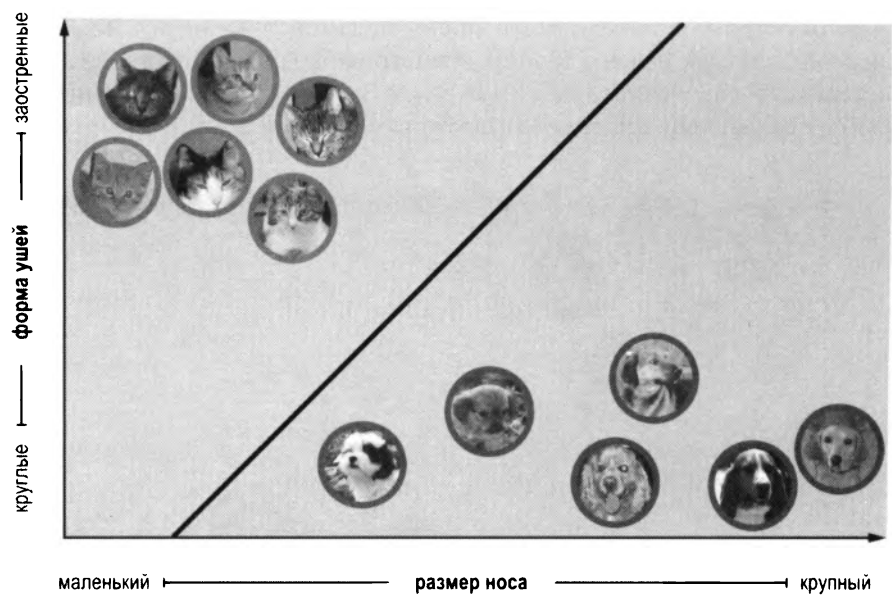


Рис. 1.3. Натренированная линейная модель (наклонная прямая линия) обеспечивает вычислительное правило для различения кошек и собак. Если по признакам любой новый снимок попадает в пространство выше прямой, то это кошка, а если ниже — собака

4. **Валидация модели.** Для валидации эффективности нашего натренированного ученика мы теперь показываем компьютеру серию ранее не встречавшихся снимков кошек и собак, которая обычно называется *валидационным набором* данных, и смотрим, насколько хорошо он может идентифицировать животное на каждом снимке. На рис. 1.4 показан примерный валидационный набор для



Рис. 1.4. Валидационный набор снимков кошек и собак (также взятых из [1]). Снимки в этом наборе не обведены (как в случае с тренировочным набором на рис.1.1), а это указывает на то, что истинная идентичность каждого снимка ученику не раскрывается. Обратите внимание, что одна из собак (бостонский терьер в правом нижнем углу) имеет маленький нос и заостренные уши. Из-за выбранного нами признакового представления компьютер подумает, что это кошка!

решаемой задачи, состоящий из трех новых снимков кошек и собак. Для этого мы берем каждый новый снимок, извлекаем наши сконструированные признаки (например, размер носа и форму уха) и проверяем, на какую сторону от нашей прямой (или классификатора) попадает признаковое представление. В этом случае, как видно на рис. 1.5, все новые кошки и все собаки, кроме одной, из валидационного набора были идентифицированы нашей натренированной моделью правильно.

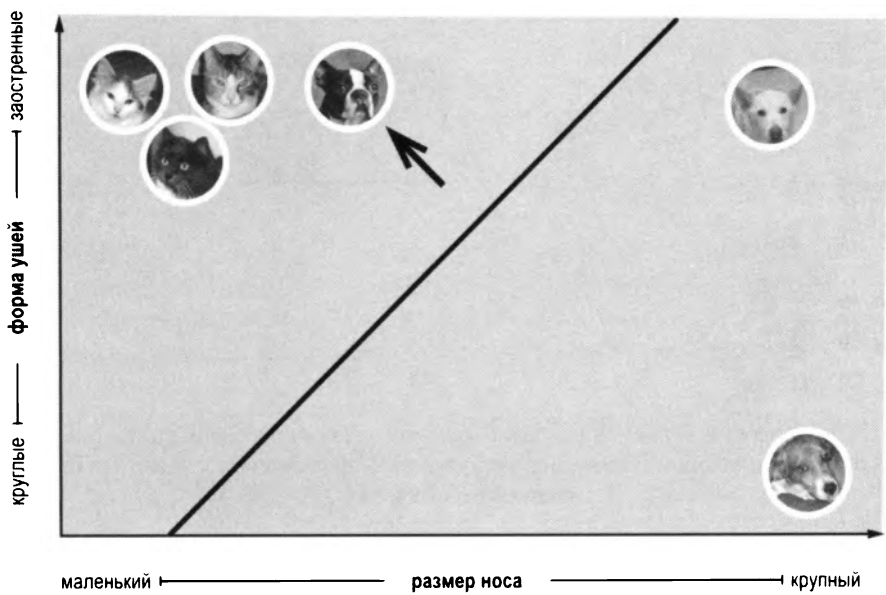


Рис. 1.5. Идентификация (признакового представления) валидационных снимков с использованием нашей натренированной линейной модели. Бостонский терьер (на которого указывает стрелка) ошибочно классифицируется как кошка, поскольку у него заостренные уши и маленький нос, как и у кошек в нашем тренировочном наборе

Неправильное идентифицирование одной собаки (бостонского терьера) в значительной степени является результатом нашего выбора признаков, сконструированных нами на основе тренировочного набора из рис. 1.1 и в некоторой степени на нашем решении использовать линейную модель (вместо нелинейной). Эта собака была неправильно идентифицирована просто потому, что ее признаки (маленький нос и заостренные уши) совпадают с признаками кошек из нашего тренировочного набора. Таким образом, хотя сначала казалось, что сочетание размера носа и формы ушей действительно поможет отличать кошек от собак, теперь благодаря валидации мы видим, что наш тренировочный набор был, возможно, слишком мал и недостаточно разнообразен, чтобы этот набор признаков был действительно эффективным.

Мы можем предпринять ряд шагов, чтобы улучшить нашего ученика. Во-первых, мы должны собрать больше данных, сформировав более крупный и разнообразный тренировочный набор. Во-вторых, мы можем подумать о конструировании/включении большего числа различительных признаков (возможно, цвета глаз, формы хвоста и т. д.), которые в дальнейшем помогут отличать кошек от собак с помощью линейной модели. Наконец, мы также можем опробовать (т. е. натренировать и валидировать) массив нелинейных моделей в надежде, что более сложное правило, возможно, будет лучше различать кошек и собак. На рис. 1.6 кратко подытожены четыре шага, необходимые для решения модельной задачи классификации "кошка — собака".

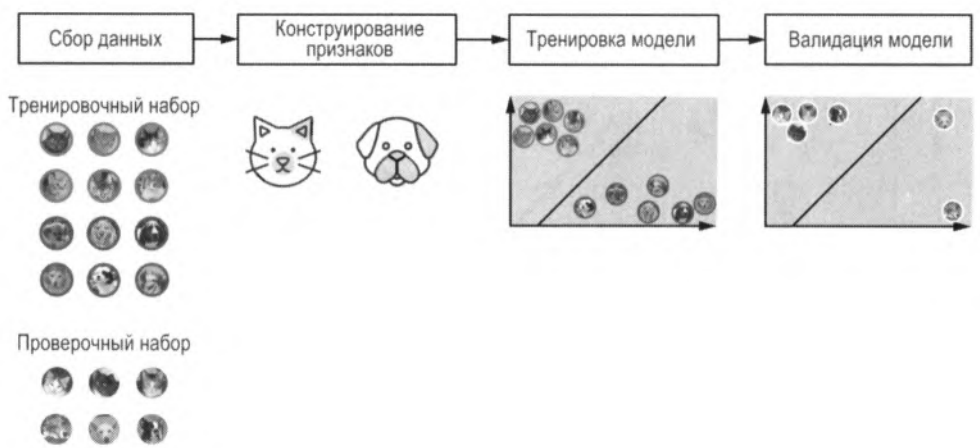


Рис. 1.6. Схематический конвейер нашей модельной задачи классификации кошек и собак. Один и тот же общий конвейер используется практически для всех задач машинного обучения

1.3. Базовая таксономия задач машинного обучения

Вычислительные правила, которые мы можем усваивать с помощью машинного обучения, обычно делятся на две главные категории, именуемые *обучение с учителем* и *обучение без учителя*, которые мы обсудим далее.

1.3.1. Обучение с учителем

Задачи обучения с учителем (например, прототип, описанный в *разд. 1.2*) относятся к автоматическому усвоению вычислительных правил, включающих связи между вводом и выводом. Применительно к широкому спектру ситуаций и типов данных этот тип задачи имеет две формы, именуемые *регрессией* и *классификацией*, в зависимости от общей числовой формы вывода.

Регрессия

Предположим, что мы хотим предсказать цену пакета акций компании, которая вот-вот станет публичной. Следуя конвейеру, описанному в *разд. 1.2*, мы сначала собираем тренировочный набор данных, состоящий из нескольких корпораций (предпочтительно действующих в одной и той же отрасли) с известными ценами пакетов. Далее нам нужно сформулировать признак (признаки), который по замыслу имеет отношение к поставленной задаче. Выручка компании является одним из таких потенциальных признаков, поскольку мы можем ожидать, что чем выручка выше, тем дороже должен быть пакет акций. Для того чтобы связать цену пакета (выход) с выручкой (вход), мы можем натренировать простую линейную модель, или *регрессионную прямую*, используя наши тренировочные данные.

Два верхних графика на рис. 1.7 показывают набор данных, содержащий информацию о соотношении цены пакета акций и выручки для десяти компаний, а также линейную модель, подогнанную к этим данным. После того как модель натренирована, цена пакета акций новой компании может быть предсказана на основе ее вы-

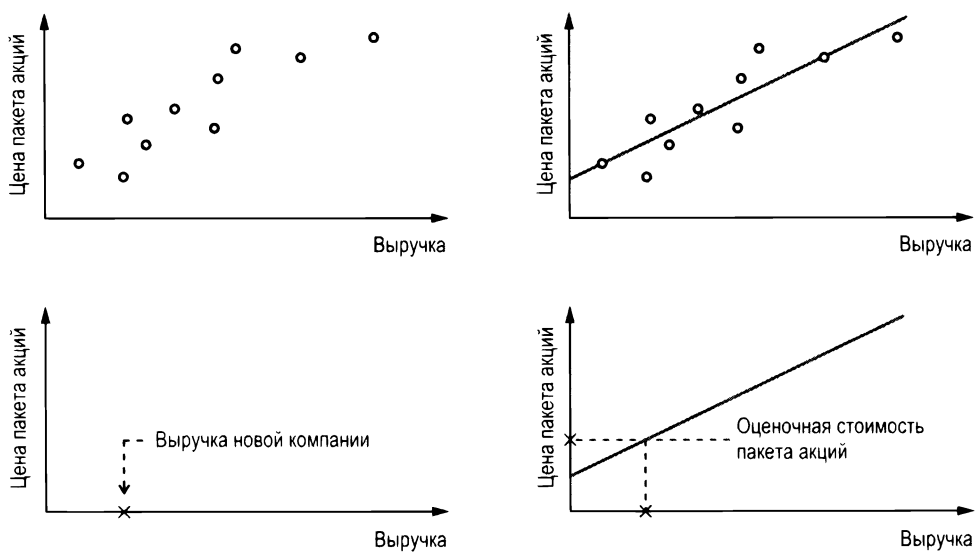


Рис. 1.7. Модельный тренировочный набор данных, состоящий из цен пакетов акции и выручек десяти корпораций (*слева сверху*). Линейная модель вписывается в данные (подгоняется к данным; *справа сверху*). Эта линия тренда моделирует совокупную траекторию точек и может быть использована для предсказания в будущем (*снизу*)

ручки (см. два нижних графика на рис. 1.7). Наконец, сравнив предсказанную цену с фактической ценой для валидационного набора данных, мы можем протестировать результативность нашей линейно-регрессионной модели и внести изменения по мере необходимости, скажем, сконструировав новые признаки (например, суммарные активы, суммарный капитал, число сотрудников, годы активной деятельности и т. д.) и/или попробовать более сложные нелинейные модели.

Такого рода задача, т. е. подгонка модели к набору тренировочных данных с тем, чтобы можно было делать предсказания о *непрерывных выходных значениях* (здесь — цена пакета акций), называется *регрессией*. Мы начнем наше подробное обсуждение регрессии в *главе 5* с линейного случая и перейдем к нелинейным моделям сперва в *главе 10*, а затем в *главах 11–14*. Далее мы опишем несколько дополнительных примеров регрессии, чтобы укрепить понимание этой концепции.

Пример 1.1. Рост задолженности по студенческим кредитам в Соединенных Штатах

На рис. 1.8 (данные взяты из [2]) показана суммарная задолженность по студенческим кредитам (т. е. деньгам, взятым студентами для оплаты обучения в колледже, проживания, питания и т. д.) граждан США с 2006 по 2014 год, измеряемым ежеквартально. За 8 лет, отраженных на этом графике, студенческая кредитная задолженность выросла почти втрое, составив к концу 2014 года более 1 трлн долларов. Регрессионная прямая хорошо вписывается в этот набор данных и своим резким положительным наклоном подчеркивает тот факт, что студенческая кредитная задолженность растет угрожающе быстро. Более того, если этот тренд сохранится, то мы можем использовать регрессионную прямую для того, чтобы предсказать, что к 2026 году суммарная задолженность студентов превысит 2 трлн долларов (мы вернемся к этой задаче позже в упражнении 5.1).

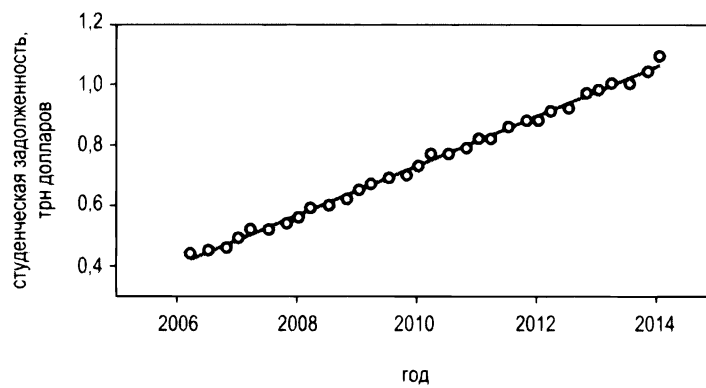


Рис. 1.8. График связан с примером 1.1, иллюстрирующим суммарную задолженность по студенческим кредитам в Соединенных Штатах, измеряемую ежеквартально с 2006 по 2014 год. Быстрый темп роста задолженности, измеряемый наклоном трендовой прямой, подогнанной к данным, подтверждает, что студенческая кредитная задолженность растет очень быстро

Пример 1.2. Закон Клейбера

Многие законы природы в естественных науках принимают форму регрессионных моделей. Например, собрав значительный объем данных, сравнивающих массу тела и скорость метаболизма (показатель расхода энергии в состоянии покоя) различных животных, биолог начала XX века Макс Клейбер обнаружил, что логарифмы этих двух величин связаны *линейно*. Эту линейную зависимость можно увидеть визуально, рассмотрев набор данных на рис. 1.9. Рассматривая аналогичный набор данных, Клейбер обнаружил, что наклон регрессионной прямой составляет около 3/4 или, другими словами, слова, что скорость метаболизма примерно равна массе в степени 3/4 (или $\sim m^{3/4}$).

Эта сублинейная зависимость означает, что по сравнению с более мелкими видами (такими как птицы) более крупные виды (такие как моржи) имеют более низкую скорость метаболизма, что согласуется с более низкой частотой сердечных сокращений и большей продолжительностью жизни (мы вернемся к этой задаче позже в упражнении 5.2).

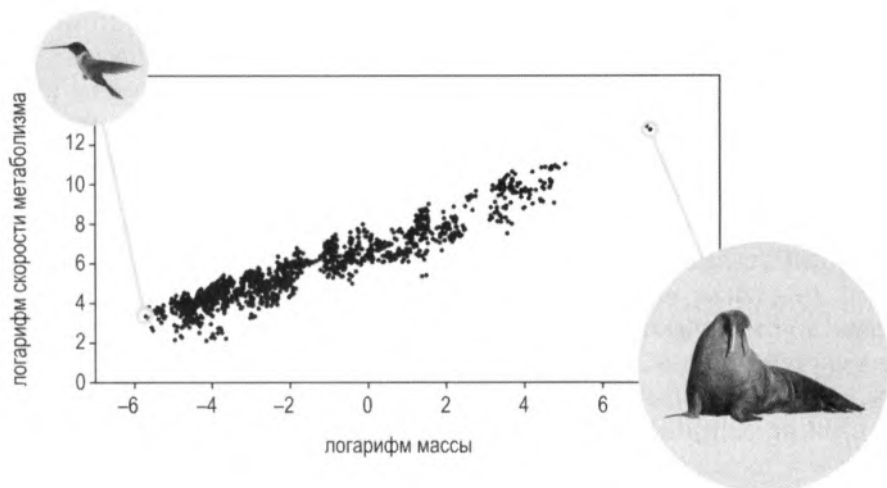


Рис. 1.9. Иллюстрация к примеру 1.2. Крупный набор данных о соотношении массы тела и скорости метаболизма, преобразованных путем взятия логарифма каждого значения, для различных животных в широком диапазоне разных масс

Пример 1.3. Предсказание кассового успеха

В 1983 году лауреат премии "Оскар" сценарист Уильям Голдман придумал фразу "никто ничего не знает" в своей книге "Приключения на экране", ссылаясь на свое убеждение, что в то время никто не мог предсказать успех или неуспех голливудских фильмов. Хотя это может быть правдой в эпоху Интернета — используя такие данные, как число поисковых запросов трейлера фильма в Сети, а также число дискуссий о фильме в социальных сетях (см., например, [3, 4]), машинное обучение может точно предсказать кассовые сборы при выходе определенных фильмов.

Предсказание продаж по целому ряду товаров/услуг, включая кассовые продажи, часто выполняется с использованием регрессии, поскольку предсказываемый выводимый результат по природе является непрерывным (достаточным).

Пример 1.4. Применение в бизнесе и промышленности

Примеры регрессии многочисленны в бизнесе и промышленности. Например, использование регрессии для точного предсказания цен на потребительские товары (от электроники до автомобилей и домов) — это чрезвычайно ценные мероприятия сами по себе (как будет разведано подробнее в примере 5.5). Регрессия также обычно используется в промышленных приложениях для лучшего понимания заданной системы, например, как конфигурация автомобиля влияет на его производительность (см. пример 5.6), в результате чего такие процессы могут быть оптимизированы.

Классификация

Классификационная задача машинного обучения в принципе аналогична задаче регрессии с ключевым различием между ними, заключающимся в том, что вместо предсказания непрерывных выходных данных в случае классификации результат, который мы стремимся предсказать, принимает *дискретные значения* или *классы*. Задачи классификации возникают во множестве форм. Так, распознавание, когда разные объекты из набора снимков отличаются друг от друга (например, рукописные цифры для автоматической сортировки почты или уличные знаки для автомобилей с водителем и без водителя), является очень популярной задачей классификации. Модельная задача, в которой надо отличить кошку от собаки, обсуждавшаяся в *разд. 1.2*, также попадает в эту корзину. Другие распространенные задачи классификации включают распознавание речи (распознавание разных специальных слов для систем распознавания голоса), определение общей предрасположенности, или сентимента, социальной сети, такой как Twitter, к определенному продукту или услуге, а также определение того, какой жест рукой человек делает из конечного набора возможностей (например, при управлении компьютером без мыши).

С геометрической точки зрения задача классификации в двух размерностях сводится к общепринятому методу поиска разделительной линии (или в более общем смысле разделительной кривой), которая точно разделяет два вида данных[1]. Это именно та точка зрения на классификацию, которую мы приняли при описании примера в *разд. 1.2*, где мы использовали прямую для разделения признаков, извлеченных из снимков кошек и собак. Новые данные из валидационного набора затем автоматически классифицируются, чтобы можно было определить, по какую сторону от прямой лежат данные. На рис. 1.10 проиллюстрирована концепция линейной модели или классификатора, используемого для выполнения классификации на двумерном наборе данных.

[1] В более высоких размерностях мы также стремимся определить разделяющую линейную гиперплоскость (или в более широком понимании нелинейное топологическое многообразие).

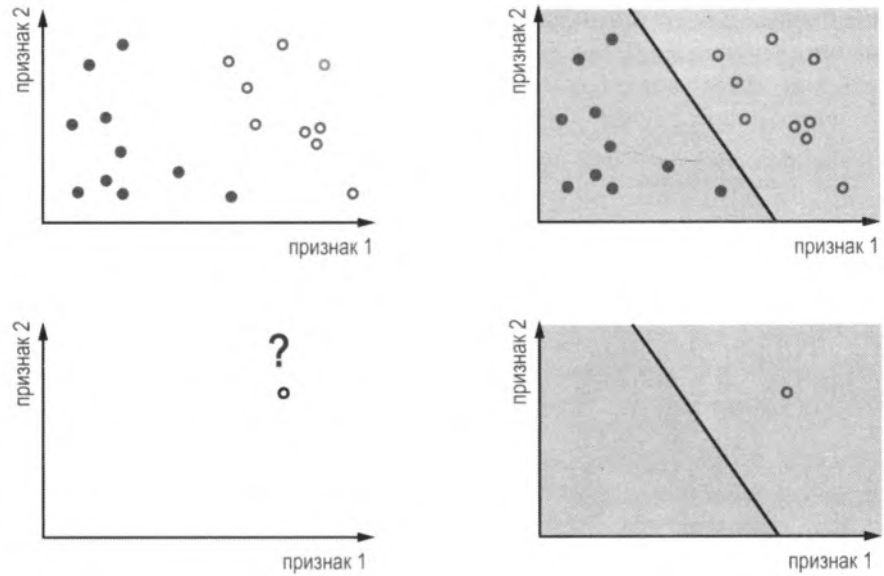


Рис. 1.10. Двумерный тренировочный набор данных, состоящий из двух разных классов: закрашенного и незакрашенного (*слева сверху*). Линейная модель натренирована разделять эти два класса (*справа сверху*). Валидационная точка, класс которой неизвестен (*слева снизу*). Валидационная точка классифицируется как незакрашенная, т. к. она находится на правой стороне натренированного линейного классификатора (*справа снизу*)

Многие задачи классификации (например, обсуждаемое далее распознавание рукописных цифр), естественно, имеют более двух классов. После описания линейной двухклассовой классификации в *главе 6* мы подробно рассмотрим линейную многоклассовую классификацию в *главе 7*. Затем нелинейное расширение обеих задач описывается, начиная с *главы 10* и на протяжении *глав 11–14*. Далее мы кратко опишем еще несколько примеров классификации, которые помогут укрепить понимание этой концепции.

Пример 1.5. Обнаружение объектов

Обнаружение объектов — часто встречающаяся задача классификации (например, [5–7]), она предусматривает автоматическое идентифицирование конкретного объекта в коллекции снимков или видео. Популярные приложения обнаружения объектов включают обнаружение лиц на снимках для организационных целей и фокусировки камеры, обнаружение пешеходов для автономного вождения транспортных средств и выявление неисправных компонентов для автоматизированного контроля качества в производстве электроники. Тот же самый каркас машинного обучения, который мы рассматриваем здесь для случая обнаружения лиц, может использоваться для решения многих аналогичных задач обнаружения.

После тренировки (линейного) классификатора на наборе тренировочных данных, содержащих изображения лиц и других объектов, лица отыскиваются в новом валидационном снимке с помощью перемещения (обычно) квадратного окна по всему снимку. В каждом местоположении скользящего окна внутри него изучается со-

держимое снимка для того, чтобы можно было выяснить, на какой стороне классификатора оно лежит (рис. 1.11). Если содержимое (признаковое представление) лежит на стороне классификатора с меткой "лицо", то содержимое классифицируется как лицо, в противном случае — как не-лицо.

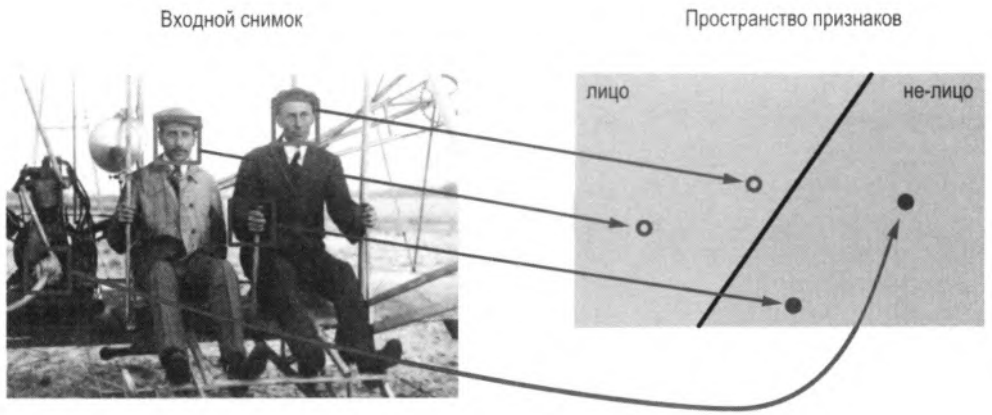


Рис. 1.11. Иллюстрация примера 1.5. Для того чтобы определить, присутствуют ли какие-либо лица на входном снимке (в данном случае снимке братьев Райт, изобретателей самолета, сидящих вместе в одном из своих первых моторизованных летательных аппаратов в 1908 г.), сканируется небольшое окно во всей его поверхности. Содержимое внутри скользящего окна в каждом экземпляре определяется как лицо путем проверки того, на какой стороне обучившегося классификатора лежит признаковое представление содержимого. На показанной здесь фигуральной иллюстрации область левее и правее усвоенного классификатора является соответственно лицевой и нелицевой сторонами классификатора

Пример 1.6. Анализ настроения

Появление социальных медиа значительно усилило голос потребителей, предоставив им массу хорошо организованных каналов для комментариев, обсуждения и ранжирования товаров и услуг (см., например, [8]). Это побудило многие фирмы выискивать интенсивные методы определения настроения своих клиентов от недавно выпущенных продуктов, рекламных кампаний и т. д. Определение агрегированных чувств большой клиентской базы с помощью текстового контента, такого как отзывы о товарах, твиты и комментарии, обычно называется *сентимент-анализом*, или *анализом настроений*. Классификационные модели часто используются для проведения сентимент-анализа, позволяющего идентифицировать потребительские данные на предмет положительных или отрицательных чувств. Мы обсудим эту задачу подробнее в примере 9.1.

Пример 1.7. Компьютерная диагностика патологических состояний здоровья

Мириады задач двухклассовой классификации естественным образом возникают в медицинских учреждениях, когда медицинский работник пытается определить, страдает ли пациент от конкретного заболевания, и когда медики проводят иссле-

дование того, какие признаки отличают больных и здоровых (в надежде, что на основе этих признаков затем может быть разработано лекарство для облегчения протекания болезни). Модальность такого рода экспериментов широко варьируется, начиная от статистических измерений пораженных областей (например, контура и площади биопсированной опухолевой ткани; см. упражнение 6.13) до биохимических маркёров, информации, выводимой из данных рентгенологического снимка, и до самих генов (подробнее в примере 11.18).

Например, в паре с алгоритмом классификации функциональная магнитно-резонансная томография (МРТ) головного мозга становится все более полезным методом диагностики неврологических расстройств, таких как аутизм, болезнь Альцгеймера и синдром дефицита внимания и гиперактивности (СДВГ). Для выполнения классификации необходим набор данных, состоящий из статистически обоснованных признаков, извлеченных из функциональных магнитно-резонансных (МР) сканов мозга пациентов, страдающих одним из ранее упомянутых когнитивных расстройств, а также лиц из контрольной группы, которые не страдают этим заболеванием. Эти функциональные МР-сканы мозга улавливают шаблоны нейронной активности, локализованные в разных частях мозга, когда пациенты выполняют простые действия, такие как отслеживание небольшого визуального объекта. На рис. 1.12, взятом из работы [9], иллюстрируется результат применения классификационной модели к задаче диагностики пациентов с СДВГ. Эта тема обсуждается подробнее в примере 11.19.

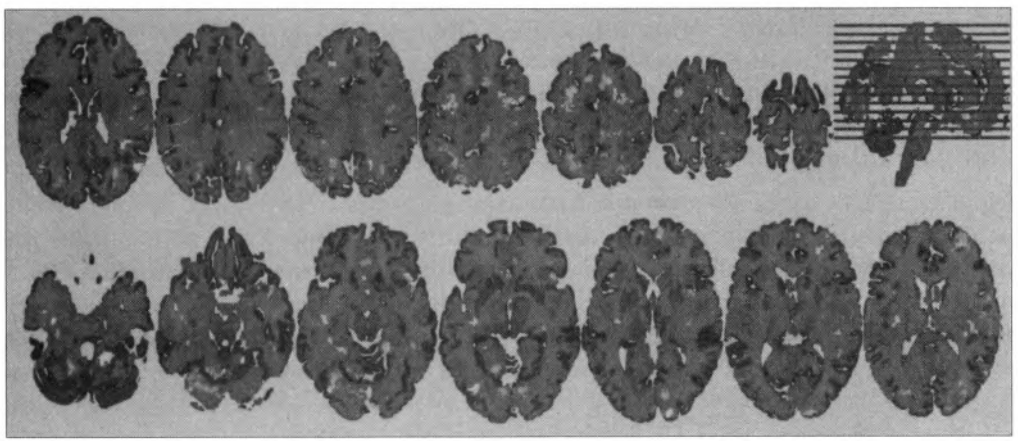


Рис. 1.12. Иллюстрация примера 1.7

Пример 1.8. Обнаружение спама

Обнаружение спама — это стандартная задача текстовой двухклассовой классификации. В большинстве электронных почтовых систем обнаружение спама автоматически выявляет нежелательные сообщения (например, рекламные объявления), именуемые спамом (spam), из писем, которые пользователи хотят видеть (на англ. языке часто именуемые ham). После тренировки спамовый детектор может удалять нежелательные сообщения без ввода данных пользователем, что значительно

улучшает работу пользователя с электронной почтой. Этот случай подробнее рассматривается в примерах 6.10 и 9.2.

Пример 1.9. Финансовые применения

Задачи двухклассовой классификации возникают во всех видах финансовых применений. Они часто используются в коммерческом кредитовании для того, чтобы можно было определить, может ли человек получить коммерческий кредит, кредитную карту и т. д. на основании его исторической финансовой информации. Эта задача стандартной двухклассовой классификации, "одолжить" денежные средства либо "не одолжить", подробнее рассматривается в примерах 6.11 и 9.7, причем последний пример описывается в контексте *отбора признаков*.

Обнаружение мошенничества является еще одной чрезвычайно популярной задачей двухклассовой классификации в финансовом пространстве. Обнаружение мошеннических финансовых (например, кредитно-карточных) операций естественным образом оформляется как задача двухклассовой классификации, где оба класса состоят из законных транзакций и мошеннических. Трудность с такими задачами в типичной ситуации заключается в том, что регистрация мошеннических транзакций во много раз затмевается валидными записями, а это крайне разбалансирует такие наборы данных (*см. разд. 6.8.4* и *6.9*).

Пример 1.10. Задачи распознавания

Задачи распознавания — это популярная форма многоклассовой классификации, цель которой состоит в том, чтобы натренировать классификатор автоматически различать коллекцию явлений, будь то человеческие жесты (распознавание жестов), различные визуальные объекты (распознавание объектов) или произнесенные слова (распознавание речи).

Например, распознавание рукописных цифр является популярной задачей распознавания объектов, обычно встроенной в мобильные банковские приложения банкоматов, чтобы дать пользователям среди прочего возможность автоматически депонировать бумажные чеки. В этом приложении каждый класс данных состоит из (снимков) нескольких рукописных версий одной цифры в диапазоне 0–9, в результате чего в общей сложности получается десять классов (рис. 1.13).

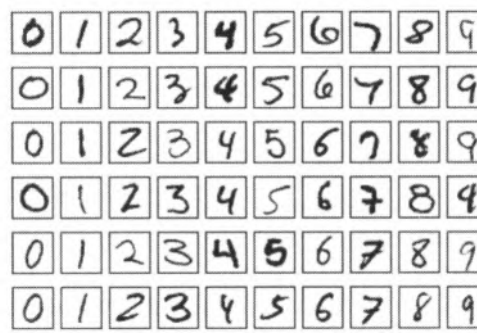


Рис. 1.13. Иллюстрация примера 1.10. Различные рукописные цифры

Мы обсудим распознавание рукописных цифр подробнее в нескольких разных местах (например, в примере 7.10), а также более общие применения распознавания объектов и речи соответственно в *разд. 9.2.4* и *9.2.3*.

1.3.2. Обучение без учителя

Обучение без учителя (в отличие от описанных ранее задач обучения с учителем) имеет дело с автоматическим усвоением вычислительных правил, которые описывают только *входные* данные. Часто такие правила усваиваются в целях упрощения набора данных, чтобы облегчить обучение с учителем, либо для анализа и интерпретации человеком. Две фундаментальные задачи обучения без учителя — *снижение размерности* и *кластеризация* — позволяют упростить набор данных двумя естественными путями: либо путем уменьшения внешней размерности входных данных (в первом случае), либо путем определения малого числа представителей, адекватно описывающих разнообразие более крупного набора данных (во втором случае). Обе подкатегории обучения без учителя впервые будут представлены в *главе 8* (где подробно описана линейная версия каждой из них) и будут обсуждаться далее в *главах 10–14* (где рассматриваются их нелинейные расширения).

Снижение размерности

Размерность современных данных, таких как фотоснимки, видео, текстовые документы и генетическая информация, часто слишком велика для эффективного использования предсказательного моделирования и анализа. Например, даже мегапиксельный фотоснимок, по сегодняшним стандартам фотоснимок средней разрешающей способности, представляет собой миллионноразмерный фрагмент данных. Это справедливо для снимка в градациях серого, который имеет только одну размерность для каждого из миллиона пикселов. Цветной мегапиксельный снимок будет иметь 3 млн размерностей. Поэтому снижение размерности такого рода данных часто имеет решающее значение для эффективного применения многих алгоритмов машинного обучения, что делает редукцию размерности обычным этапом предобработки для предсказательных и аналитических задач.

С геометрической точки зрения, уменьшить размерность набора данных означает сжать или спроецировать его вниз на правильную прямую или кривую меньшей размерности (или в более широком понимании линейную гиперплоскость или нелинейное топологическое многообразие), предпочтительно такую, которая сохраняет как можно больше определяющих характеристик исходных данных.

Эта общая идея проиллюстрирована для двух наборов данных на рис. 1.14, где двумерные (*слева*) и трехмерные (*справа*) данные сжимаются (или проецируются) на соответствующую одномерную прямую и двумерную гиперплоскость соответственно, уменьшая окружающую размерность данных на единицу в каждом случае и сохраняя бóльшую часть контура исходных данных. На практике уменьшение в размерности современных крупных наборов данных может быть намного больше, чем показано на этом рисунке.



Рис. 1.14. Два набора данных, состоящих из двумерных (*слева*) и трехмерных (*справа*) входных данных, показаны в виде полых черных окружностей. Данные в каждом случае проецируются на подпространство с более низкой размерностью, сохраняя при этом значительную часть структуры исходных данных

Кластеризация

Кластеризация призвана выявлять опорную структуру в наборе входных данных путем группировки точек, имеющих общие структурные характеристики, например близость друг к другу в пространстве признаков, что помогает лучше организовывать или подытоживать тренировочный набор данных для анализа человеком или компьютером. Структура данных может сильно варьироваться, причем данные попадают в шаровые скопления — кластеры — или располагаются вдоль нелинейных топологических многообразных контуров (рис. 1.15).

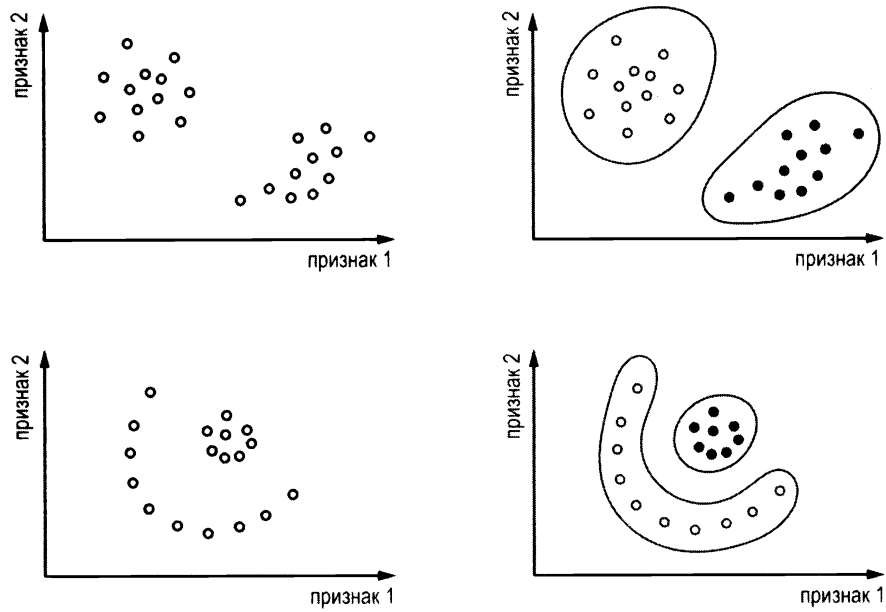


Рис. 1.15. Два примера огромного массива, в котором входные данные могут группироваться в кластеры. Алгоритмы кластеризации предназначены для выявления таких четко различимых структур. В каждом случае отдельные кластеры в исходных данных (*слева*) окрашиваются для целей визуализации (*справа*)

1.4. Математическая оптимизация

Как мы увидим в оставшейся части книги, мы можем формализовать поиск параметров автообучающейся модели посредством четко определенных математических функций. Эти функции, обычно именуемые *функциями потери* или *стоимости*, или *стоимостными функциями*, принимают конкретный набор модельных параметров и возвращают балл, показывающий, насколько хорошо мы выполнили бы заданную задачу обучения, используя этот вариант параметров. Высокое значение указывает на вариант параметров, который дает слабую результативность, тогда как противоположное справедливо для варианта параметров, обеспечивающих низкое значение. Например, вспомните пример предсказания цены пакета акций, приведенный на рис. 1.7, где мы ставили своей целью усвоить регрессионную прямую для предсказания цены пакета акций компании, основываясь на ее выручке. Эта прямая меняет свое положение относительно данных путем оптимальной настройки двух ее параметров: наклона и вертикального пересечения. Геометрически это соответствует отысканию набора параметров, обеспечивающих наименьшее значение (именуемое *минимумом*) двумерной функции потерь (рис. 1.16). Эта концепция играет аналогичную фундаментальную роль и в классификации (да и вообще во всех задачах машинного обучения). На рис. 1.10 мы подробно описали то,

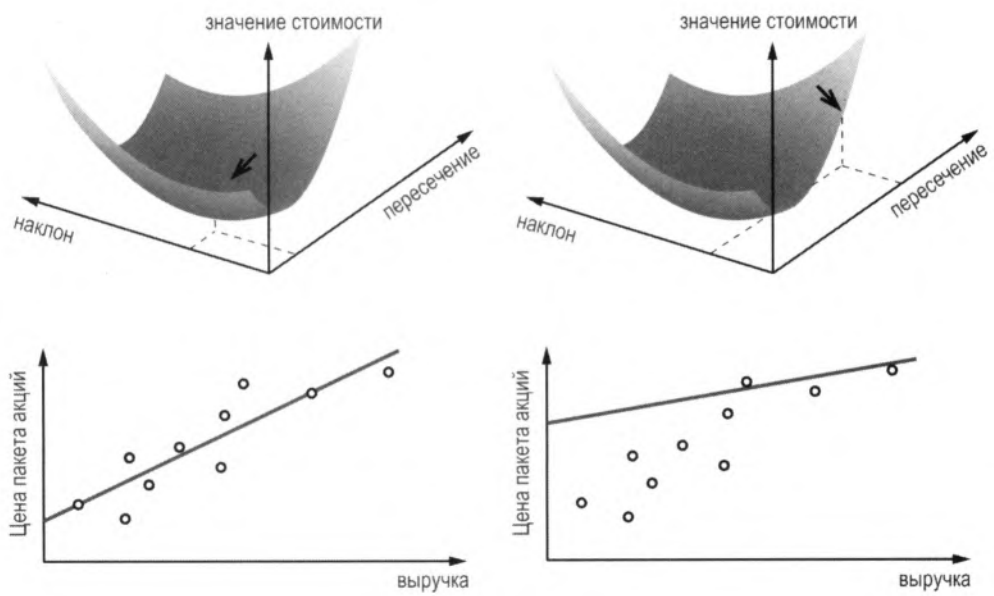


Рис. 1.16. Рисунок (*сверху*) двумерной функции потерь, связанной с обучением параметров наклона и пересечения линейной модели для задачи регрессии цены пакета акций, рассмотренной в предыдущем разделе и показанной на рис. 1.7. Кроме того, здесь показаны два разных набора параметрических значений: один (*слева*) в минимуме функции потерь, а другой (*справа*) в точке с большим значением функции, и линейная модель (*снизу*), соответствующая каждому набору параметров на верхних рисунках. Набор параметров, приводящих к наилучшей подгонке, найден в минимуме поверхности

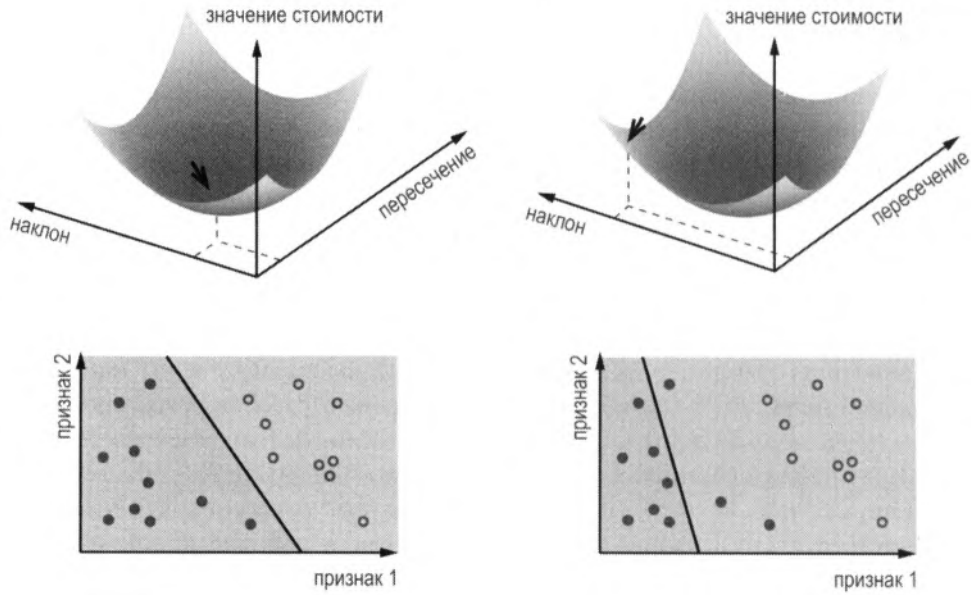


Рис. 1.17. Рисунок (*сверху*) двумерной функции потерь, связанной с автоматическим усвоением параметров наклона и пересечения линейной модели, разделяющей два класса набора данных, впервые показанного на рис. 1.10. Здесь также показаны два разных набора параметрических значений: один (*слева*) соответствует минимуму функции, а другой (*справа*) соответствует точке с большим значением функции, и линейные классификаторы (*снизу*), соответствующие каждому набору параметров в верхних панелях. Оптимальный набор параметров, т. е. тех, которые дают минимальное значение соответствующей функции, позволяет выполнить наилучшее разделение двух классов

как тренируется общий линейный классификатор, причем идеальная конфигурация его параметров снова соответствует минимуму функции потерь (рис. 1.17).

Поскольку низкое значение соответствует высокорезультативной модели, как в случае регрессии, так и в случае классификации (и, как мы увидим, для задач обучения без учителя), мы всегда будем стремиться минимизировать функции потерь, чтобы найти идеальные параметры связанных с ними моделей. В качестве исследования вычислительных методов минимизации формальных математических функций инструменты *математической оптимизации*, таким образом, играют фундаментальную роль во всей книге. Вдобавок, как мы увидим далее, начиная с *главы 11*, оптимизация также играет фундаментальную роль в *перекрестной валидации* или автоматическом обучении правильной *нелинейной* модели для любого набора данных. По причине критически важных ролей, которые математическая оптимизация играет в машинном обучении, мы начинаем наше обсуждение с исчерпывающего описания основополагающих инструментов оптимизации в *главах 2–4*.

1.5. Заключение

В этой главе мы дали широкий обзор машинного обучения с акцентом на критически важные концепции, которые будем встречать неоднократно на протяжении всей книги. В *разд. 1.2* мы начали с описания модельной задачи машинного обучения, а также шагов, обычно предпринимаемых для решения такой задачи (подытоженных на рис. 1.6). Затем в *разд. 1.3* мы представили фундаментальные семейства задач машинного обучения — обучение с учителем и без, подробно описав ряд применений обоих. Наконец, в *разд. 1.4* мы мотивировали необходимость математической оптимизации отысканием идеальных параметров для модели машинного обучения, которая имеет прямое соответствие с геометрической задачей отыскания наименьшего значения связанной с ней функции потерь (наглядно подытоженной на рис. 1.16 и 1.17).

Часть I

Математическая оптимизация

2 Прямые методы оптимизации (нулевого порядка)[1]

2.1. Введение

Задача определения наименьшего (или наибольшего) значения, которое может принимать функция, именуемая ее *глобальным минимумом* (или *глобальным максимумом*), имеет многовековую историю и многочисленные приложения во всех науках и отраслях инженерного дела. В этой главе мы начнем наше исследование математической оптимизации с описания *прямых методов оптимизации* — также именуемых приемами *беспроизводной оптимизации*. Несмотря на то что *прямые методы оптимизации* не всегда самые мощные инструменты оптимизации в нашем инструментарии, они концептуально являются самыми простыми и доступными нам инструментами, требующими наименьшего числа интеллектуальных механизмов и терминологического жаргона для описания. По этой причине обсуждение *прямых методов оптимизации* позволит нам в простой окружающей обстановке обнажить ряд ключевых понятий, которые мы увидим в последующих главах в более сложных обстановках, включая понятия *оптимальности, локальной оптимизации, направлений спуска, длин шага* и т. д.

2.1.1. Визуализация минимумов и максимумов

Когда функция принимает только один или два входных значения, мы можем попытаться визуально определить ее минимумы или максимумы, построив ее график над крупным участком входного пространства. Хотя эта идея не сработает, когда функция принимает три или более входных значения (поскольку мы больше не сможем ее надлежаще визуализировать), мы тем не менее начнем с того, что сначала рассмотрим ряд примеров с низкой размерностью, чтобы интуитивно ощутить возможность эффективно выявлять эти желаемые минимумы или максимумы в целом.

---

[1] Прямыми методами в теории оптимизации традиционно называются методы, требующие вычисления только целевой функции в точках приближения, без взятия производных. В англоязычной традиции такие методы принято называть *методами нулевого порядка* (zero-order) по аналогии с методами оптимизации первого порядка, требующими взятия первой производной (для многомерных функций — градиента) целевой функции, и методами оптимизации второго порядка, требующими взятия второй производной (для многомерных функций — гессиана) целевой функции. — *Прим. ред.*

Пример 2.1. Визуальный осмотр одновходных функций на предмет минимумов и максимумов

На двух верхних графиках рис. 2.1 построены графики квадратичной функции

$$g(w) = w^2 \tag{2.1}$$

по небольшому набору данных (центрированному вокруг нуля) от $w = -3$ до $w = 3$. На этом же рисунке отмечено вычисленное значение глобального минимума функции в $w = 0$ (т. е. точка (0, $g(0)$), где $g(0) = 0$) зеленой точкой. Обратите внимание, что по мере удаления от начала координат (как в отрицательном, так и в положительном направлении) значение функции g становится все больше, из чего вытекает, что ее глобальные максимумы лежат в точках $w = \pm\infty$.

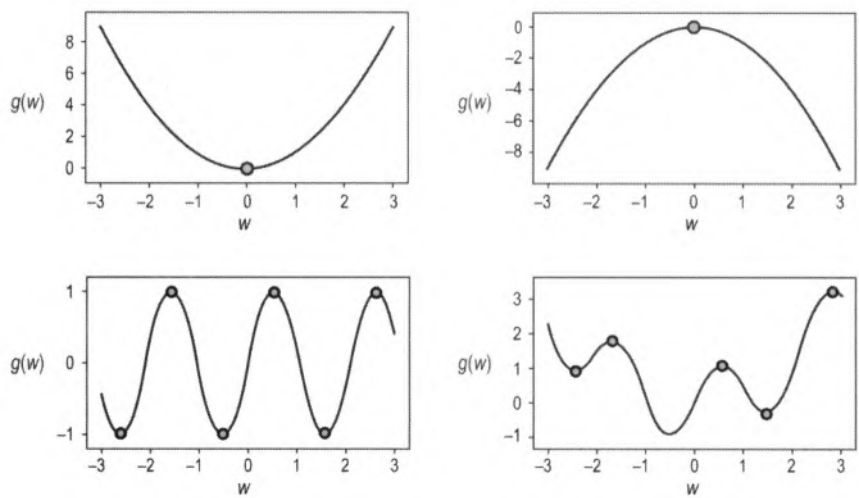


Рис. 2.1. Иллюстрация примера 2.1. Четыре примера функций с вычисленными значениями их минимумов и максимумов, выделенных зелеными точками

На правом верхнем графике рис. 2.1 показан результат умножения предыдущей квадратичной функции на -1, дающий новую квадратичную функцию:

$$g(w) = -w^2. \tag{2.2}$$

Это умножение приводит к тому, что функция переворачивается и ее глобальные минимумы теперь лежат в $w = \pm\infty$, а вход $w = 0$, который до этого обеспечивал *глобальный минимум* g, теперь возвращает свой *глобальный максимум*. Вычисленное значение этого максимума снова отмечено зеленой точкой.

Слева внизу на рис. 2.1 построен график синусоидальной функции

$$g(w) = \sin(3w). \tag{2.3}$$

Здесь мы ясно видим, что (в диапазоне построенного графика функции) существуют три глобальных минимума и три глобальных максимума (вычисленное значение каждого из них отмечено зеленой точкой на синусоиде). Если бы мы построили график этой функции на основе более широкого диапазона ее входных данных, то

увидели бы, что она имеет бесконечно много таких глобальных минимумов и максимумов (существующих при каждом нечетном кратном числе $\pi/6$).

Справа внизу на рис. 2.1 приведен график функции, которая является суммой синусоидальной и квадратичной функций и принимает алгебраическую форму:

$$g(w) = \sin(3w) + 0{,}3w^2 . \tag{2.4}$$

Проанализировав график этой функции, мы видим, что она имеет глобальный минимум около точки $w = -0{,}5$. Указанная функция также имеет другие минимумы и максимумы, которые являются *локальными*, т. е. значения, которые минимальны или максимальны только локально и только по отношению к своим соседним точкам (а не к функции в целом). Например, g имеет локальный максимум около $w = 0{,}6$ и локальный минимум около $w = 1{,}5$. Вычисленные значения как максимумов, так и минимумов над диапазоном входных значений, показанных для этой функции, отмечены зелеными точками на рис. 2.1.

2.2. Условие оптимальности нулевого порядка

С помощью ряда простых примеров, иллюстрирующих минимумы и максимумы, мы теперь можем определить их более формально. Задача определения глобального минимума функции g с N входными переменными $w_1, w_2, ..., w_N$ формально может быть сформулирована как следующая ниже *задача минимизации*

$$\underset{w_1, w_2, ..., w_N}{\text{минимизировать}}\ g\left(w_1, w_2, ..., w_N\right), \tag{2.5}$$

которую можно переписать гораздо компактнее (путем обозначения всех входных данных в виде N-мерного вектора $\mathbf{w}$) как

$$\underset{\mathbf{w}}{\text{минимизировать}}\ g(\mathbf{w}). \tag{2.6}$$

Решая такую задачу минимизации, мы стремимся найти такую точку $\mathbf{w}^\star$, что

$$g(\mathbf{w}^\star) \leq g(\mathbf{w}) \text{ для всех } \mathbf{w}. \tag{2.7}$$

Эта запись представляет собой *определение нулевого порядка глобального минимума*. В общем случае функция может иметь несколько или даже бесконечно много глобально минимальных точек (как синусоидальная функция в уравнении (2.3)).

Мы также можем описать математические точки $\mathbf{w}^\star$, в которых g имеет глобальный максимум. Для таких точек мы можем написать

$$g(\mathbf{w}^\star) \geq g(\mathbf{w}) \text{ для всех } \mathbf{w}. \tag{2.8}$$

Эта запись представляет собой *определение нулевого порядка глобального максимума*. Для того чтобы выразить наше стремление к глобальному максимуму функции, запишем:

$$\underset{\mathbf{w}}{\text{максимизировать}}\ g(\mathbf{w}). \tag{2.9}$$

Обратите внимание, что понятия минимумов и максимумов функции всегда связаны друг с другом через умножение на –1. То есть глобальный минимум функции g всегда является глобальным максимумом функции $-g$, и наоборот. Поэтому мы всегда можем выразить задачу максимизации в уравнении (2.9) в терминах задачи минимизации, как

$$\underset{\mathbf{w}}{\text{максимизировать}} \; -g(\mathbf{w}). \tag{2.10}$$

Подобно определениям нулевого порядка для глобальных минимумов и максимумов в уравнениях (2.7) и (2.8), существуют также определения нулевого порядка для локальных минимумов и максимумов. Например, мы можем сказать, что функция g имеет *локальный* минимум в точке $\mathbf{w}^\star$, если

$$g(\mathbf{w}^\star) \leq g(\mathbf{w}) \text{ для всех } \mathbf{w} \text{ вблизи } \mathbf{w}^\star. \tag{2.11}$$

Утверждение "для всех $\mathbf{w}$ вблизи $\mathbf{w}^\star$" является относительным, просто описывая тот факт, что должны существовать окрестности (как бы малы они ни были) вокруг $\mathbf{w}^\star$, такие, что при вычислении значения функции g в каждой точке этой окрестности функция g достигает своего наименьшего значения в точке $\mathbf{w}^\star$. То же самое формальное определение нулевого порядка можно сделать и для локальных максимумов, заменив знак $\leq$ на $\geq$.

Упакованные вместе, эти определения нулевого порядка для минимумов и максимумов (обычно именуемые *оптимумами*) часто называют *условием для оптимальности нулевого порядка*. Фраза *"нулевой порядок"* в этом контексте означает, что в каждом случае оптимумы функции определяются в терминах самой функции (и ничем больше). В последующих главах мы увидим более *высокопорядковые* определения оптимальных точек, в частности определения *первого порядка*, включающие первую производную функции в *главе 3*, а также определения *второго порядка*, включающие вторую производную функции в *главе 4*.

2.3. Методы глобальной оптимизации

В этом разделе мы опишем первый подход, который можно было бы использовать для приближенной минимизации произвольной функции: вычислить функцию, используя большое число входных точек, и рассматривать вход, который обеспечивает наименьшее значение функции как приближенный глобальный минимум функции. Этот подход называется *глобально-оптимизационным методом*, поскольку он способен аппроксимировать глобальные оптимумы функции (при условии достаточно большого числа вычисляемых значений).

С такого рода схемой оптимизации связан важный вопрос: как выбирать входные данные для их апробирования с помощью универсальной функции? Совершенно очевидно, что мы не можем апробировать их все, т. к. даже для непрерывной функции одной переменной (single-input function) апробированию подлежит бесконечное число точек.

Здесь мы можем использовать два подхода к выбору нашего (конечного) набора входных точек для тестирования: отбирать точки (т. е. угадывать их) равноотстоящих элементов (равномерный отбор) либо подбирать одинаковое число входных точек случайно (случайный отбор). Проиллюстрируем оба варианта в примере 2.2.

Пример 2.2. Минимизация квадратичной функции

Рассмотрим два метода отбора для отыскания глобального минимума квадратичной функции

$$g(w) = w^2 + 0,2\,, \tag{2.12}$$

которая имеет глобальный минимум в точке $w = 0$. Для простоты ограничим интервал, над которым выполним поиск, — $[-1; +1]$. На двух верхних рис. 2.2 приведен результат равномерного и случайного отбора из четырех входов, показанных синим цветом (причем соответствующие вычисленные значения показаны зеленым цветом на самой функции). Путем случайного отбора мы смогли (случайно) достичь более низко расположенной точки по сравнению с отбором входных значений функции, которые отбирались равномерно равноудаленно. Однако, используя достаточное число выборок, мы можем отыскать вход, очень близкий к истинному глобальному минимуму функции, при любом подходе к отбору. На двух нижних графиках показан результат равномерного отбора 20 входов против случайного. Видно, что, увеличивая число выборок с использованием любого подхода из двух, мы можем аппроксимировать глобальный минимум с гораздо большей точностью.

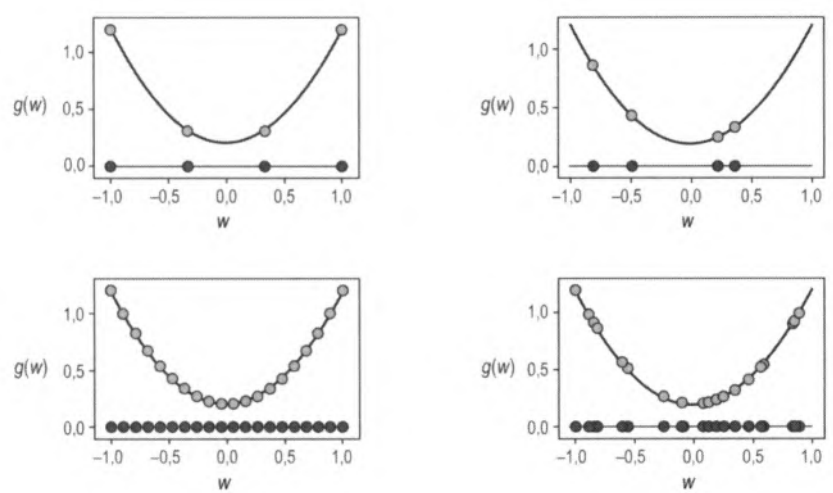


Рис. 2.2. Иллюстрация примера 2.2. Минимизирование простой функции посредством отбора входных значений или "угадывания". *Сверху*: отбор входных значений функции четыре раза равномерно (*слева*) и случайно (*справа*). Здесь отобранные входы показаны синими графическими точками, а их значения, вычисленные функцией, — зелеными точками. *Снизу*: равномерный (*слева*) и случайный (*справа*) отборы 20 раз. Чем больше выборок мы возьмем, тем больше вероятность того, что мы отыщем точку, близкую к глобальному минимуму, используя любой из подходов к отбору

Обратите внимание, что в обоих подходах к глобальной оптимизации, рассмотренных в примере 2.2, мы задействуем лишь условие оптимальности нулевого порядка, т. к. из множества, состоящего из K выбранных входов $\left\{\mathbf{w}^k\right\}_{k=1}^K$, мы выбираем один вход $\mathbf{w}^j$ с наименьшим вычисленным значением функции потерь

$$g\left(\mathbf{w}^j\right) \leq g\left(\mathbf{w}^k\right), \quad k = 1, 2, ..., K, \tag{2.13}$$

что действительно является аппроксимацией условия оптимальности нулевого порядка, рассмотренного в предыдущем разделе.

Несмотря на то что этот естественный каркас нулевого порядка прост в имплементации и совершенно адекватен для функций с низкоразмерными входами, как мы увидим далее, он терпит фиаско, когда входная размерность функции вырастает даже до умеренного размера.

2.3.1. Проклятие размерности

Хотя такого рода глобальная оптимизация, основанная на вычислениях нулевых производных функции, прекрасно работает для низкоразмерных функций, она быстро становится неприемлемой при попытке решать задачи с функциями с более крупным числом входов или, другими словами, функциями, которые принимают N-мерный вход $\mathbf{w}$, где N является большим. Такие методы оптимизации практически непригодны в современном машинном обучении, поскольку функции, с которыми мы часто имеем дело, имеют размерности входа от сотен до сотен тысяч или даже миллионов.

Для того чтобы понять, почему глобальный подход быстро становится неосуществимым, представьте себе, что мы используем равномерный отбор точек по всему входному пространству функции одной переменной, выбирая (в качестве аргументации) три точки, каждая на расстоянии d от предыдущей (рис. 2.3, *слева сверху*). Теперь представьте себе, что размерность входного пространства функции увеличивается на единицу и диапазон каждого входа точно совпадает с диапазоном исходной функции одной переменной (рис. 2.3, *в центре сверху*). Мы по-прежнему стремимся покрыть пространство равномерно и с достаточным числом отбираемых образцов, чтобы каждый вход, который мы вычисляем, снова находился на расстоянии d от своих ближайших соседей в любом направлении. Для того чтобы сделать это в теперь уже двумерном пространстве, нам нужно отобрать $3^2 = 9$ входных точек. Если мы снова увеличим размерность входа так, чтобы выборка равномерно распределилась по всему входному пространству и каждый вход находился на максимальном расстоянии d от своих соседей в каждом входном измерении, то нам понадобится $3^3 = 27$ входных точек (рис. 2.3, *справа сверху*). Если мы мысленно продолжим этот эксперимент, то для общего N-мерного входа нам придется отобрать 3^N точек — огромное число даже для умеренных значений N. Этот простой пример демонстрирует так называемое *проклятие размерности*, описывающее экспоненциальную трудность, с которой приходится сталкиваться при попытке справиться с функциями возрастающей входной размерности.

Проклятие размерности остается проблемой, даже если мы решим выбирать образцы случайно. Для того чтобы понять, почему это так, используя тот же гипотетический сценарий, предположим теперь, что вместо фиксирования расстояния d каждой выборки от ее соседей мы фиксируем суммарное число случайно отбираемых образцов постоянным значением и смотрим, насколько хорошо они склонны распределяться по входному пространству, когда мы увеличиваем его размерность. На рис. 2.3 демонстрируется пример того, как в общей сложности десять точек отбираются случайно соответственно в одномерном ($N = 1$), двумерном ($N = 2$) и трехмерном ($N = 3$) пространствах. И снова нас сдерживает проклятие размерности. По мере увеличения размерности входного пространства среднее число образцов, отбираемых на единичный гиперкуб, экспоненциально падает, оставляя все больше и больше участков пространства без единой выборки или соответствующего вычисленного значения. Для того чтобы противодействовать этой проблеме, нам нужно было бы начать отбирать экспоненциально много точек, что привело бы к той же самой проблеме, с которой мы сталкиваемся при задействовании схемы с равномерным отбором.

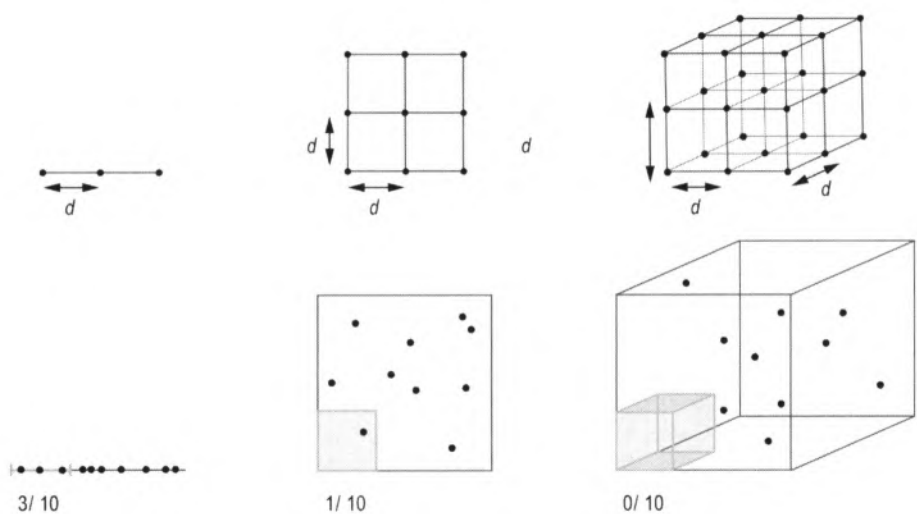


Рис. 2.3. Число входных точек, которые мы должны отбирать равномерно, если хотим, чтобы каждая из них находилась на расстоянии d от своих соседей, растет экспоненциально по мере увеличения входной размерности функции (*сверху*). Если для покрытия одновходного пространства таким образом используются три точки (*слева*), то $3^2 = 9$ точек потребуются в двух размерностях и $3^3 = 27$ точек в трех размерностях (и этот тренд продолжается). Случайный отбор также не решает проблему (*внизу*)

2.4. Методы локальной оптимизации

В отличие от методов глобальной оптимизации, описанных в *разд. 2.3*, где большое число входных точек отбирается одновременно, при этом наименьшее вычисленное значение венчает приближенный глобальный минимум, *методы локальной опти-*

мизации работают, начиная с одной входной точки, а затем последовательно уточняя ее, приводят точку к приближенной минимальной точке. Методы локальной оптимизации — на сегодняшний день наиболее популярные схемы математической оптимизации, используемые в машинном обучении, и именно их мы будем рассматривать далее в этой книге. Хотя в конкретных видах методов локальной оптимизации, которые мы обсудим позднее, имеются существенные различия, все они имеют общий каркас, который мы введем в этом разделе.

2.4.1. Картина в целом

Начиная с входа-образца, обычно именуемого *начальной точкой* и обозначаемого по всему тексту через $\mathbf{w}^0$, методы локальной оптимизации последовательно уточняют эту начальную точку, подтягивая ее по *склону* вниз к точкам, которые находятся все ниже и ниже на графике функции, в конечном итоге достигая минимума (рис. 2.4). Выражаясь конкретнее, от $\mathbf{w}^0$ точка сползает вниз по склону к новой точке $\mathbf{w}^1$, расположенной ниже на графике функции, т. е. где $g\left(\mathbf{w}^0\right) > g\left(\mathbf{w}^1\right)$. Сама точка $\mathbf{w}^1$ затем сползает вниз к новой точке $\mathbf{w}^2$. Повторение этого процесса K раз дает последовательность из K точек (исключая нашу стартовую начальную точку)

$$\mathbf{w}^0, \mathbf{w}^1, ..., \mathbf{w}^K, \tag{2.14}$$

где каждая последующая точка находится (вообще говоря) на все более и более низкой части графика функции, т. е.

$$g\left(\mathbf{w}^0\right) > g\left(\mathbf{w}^1\right) > ... > g\left(\mathbf{w}^K\right). \tag{2.15}$$

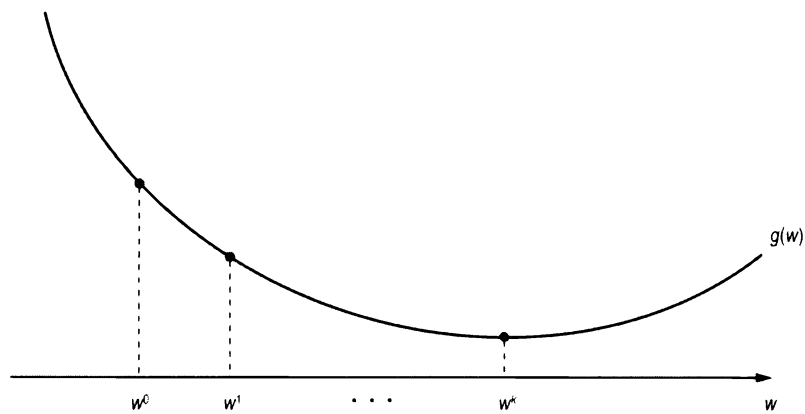


Рис. 2.4. Методы локальной оптимизации работают путем минимизирования целевой функции на протяжении последовательности шагов. Здесь показан универсальный метод локальной оптимизации, применяемый для минимизирования функции одной переменной. Начиная со стартовой точки $\mathbf{w}^0$, мы движемся к более низким точкам функции потерь, словно мяч, скатывающийся по склону

2.4.2. Общий каркас

В общем случае последовательный процесс уточнения, осуществляемый методом локальной оптимизации, работает следующим образом. Для того чтобы сделать первый шаг от начальной точки $\mathbf{w}^0$ до первого обновления $\mathbf{w}^1$, нужно найти то, что называется *направлением спуска* в точке $\mathbf{w}^0$. Это вектор направления $\mathbf{d}^0$ во входном пространстве, начинающийся с точки $\mathbf{w}^0$ и указывающий направление в сторону новой точки $\mathbf{w}^1$ с меньшим значением функции. Когда такое направление найдено, первое обновление $\mathbf{w}^1$ задается суммой

$$\mathbf{w}^1 = \mathbf{w}^0 + \mathbf{d}^0. \tag{2.16}$$

Для того чтобы уточнить точку $\mathbf{w}^1$, мы ищем новое направление спуска $\mathbf{d}^1$, которое позволит нам двигаться вниз по склону из точки $\mathbf{w}^1$. Найдя такое направление, мы сформируем второе обновление $\mathbf{w}^2$ как сумму:

$$\mathbf{w}^2 = \mathbf{w}^1 + \mathbf{d}^1. \tag{2.17}$$

Будем повторять этот процесс, создавая последовательность входных точек:

$$\begin{aligned} &\mathbf{w}^0; \\ &\mathbf{w}^1 = \mathbf{w}^0 + \mathbf{d}^0; \\ &\mathbf{w}^2 = \mathbf{w}^1 + \mathbf{d}^1; \\ &\ldots \\ &\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^{k-1}, \end{aligned} \tag{2.18}$$

где $\mathbf{d}^{k-1}$ — это направление спуска, определяемое на k-м шаге процесса, задающее k-й шаг как $\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^{k-1}$, такой, что в конечном итоге выполняются неравенства в уравнении (2.15). Весь процесс схематично показан с помощью универсальной функции, принимающей два входа (рис. 2.5, *сверху*). Двухвходная функция проиллюстрирована здесь с помощью *контурного графика* распространенного инструмента визуализации, который позволяет проецировать функцию вниз на ее входное пространство. Более темные участки на графике соответствуют точкам с бóльшими вычисленными значениями (выше на графике функции), в то время как более яркие участки соответствуют точкам с меньшими вычисленными значениями (ниже на графике функции).

Направления спуска в уравнении (2.18) можно найти массой способов. В остальных разделах этой главы мы обсудим *подходы нулевого порядка*, которые для этого предназначены, а в следующих главах мы опишем так называемые *подходы первого и второго порядков*, т. е. подходы, которые используют производные первого и второго порядков для определения направлений спуска. Именно методами определения направления спуска отличаются друг от друга главенствующие схемы локальной оптимизации.

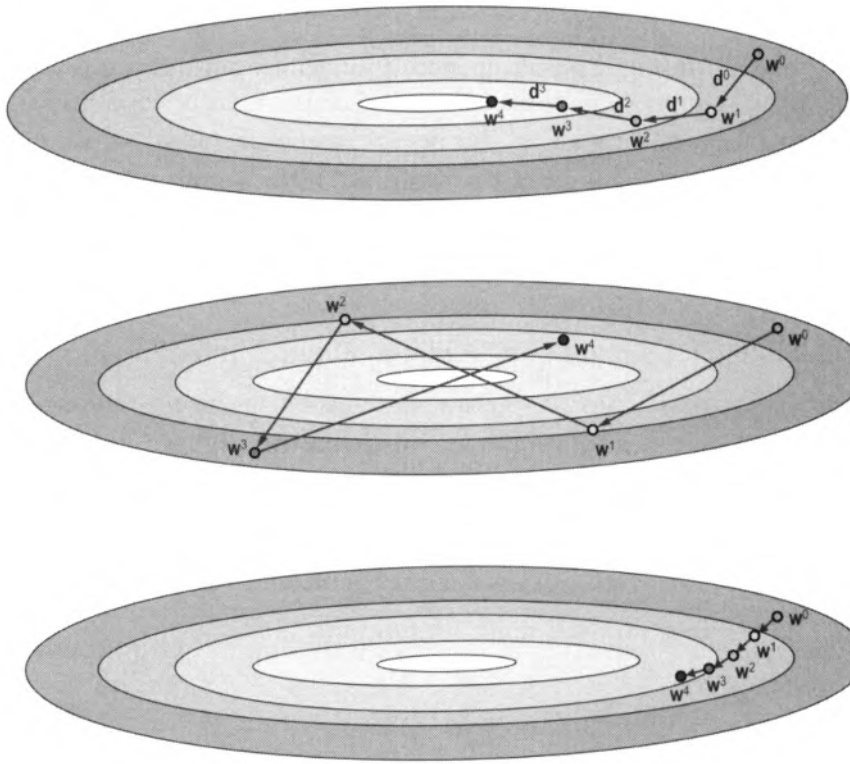


Рис. 2.5. Иллюстрация схемы локальной оптимизации, применяемой для поиска минимума функции, принимающей два входа, при этом направления спуска отмечены стрелками черного цвета (*сверху*). Векторы направления слишком велики, что вызывает очень сильное осцилляторное поведение вокруг минимальной точки (*в центре*). Векторы направления слишком малы, что требует большого числа шагов, чтобы достичь минимума (*снизу*)

2.4.3. Параметр длины шага

Будем вычислять дальность перемещения на каждом шаге метода локальной оптимизации, проверяя общую форму локального шага. Сняв это показание, мы видим, что на k-м шаге, определенном в уравнении (2.18), мы перемещаемся на расстояние, равное длине соответствующего направления спуска

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|\mathbf{d}^{k-1}\right\|_2. \tag{2.19}$$

Длина векторов спуска может вызывать проблемы, даже если они указывают в направлении спуска, вниз по склону. Например, если они являются слишком длинными, как показано на рис. 2.5, *в центре*, то локальный метод может очень сильно осциллировать на каждом шаге обновления, никогда не достигнув приближенного минимума. Если векторы спуска являются слишком короткими, то локальный метод будет продвигать нас настолько медленно, что потребуется слишком много шагов для достижения приближенного минимума (рис. 2.5, *внизу*).

Ввиду этой потенциальной проблемы многие схемы локальной оптимизации оснащены так называемым *параметром длины шага*, именуемым на жаргоне машинного обучения *коэффициентом скорости обучения*[2]. Этот параметр, который позволяет нам контролировать длину каждого шага обновления (отсюда и название параметра "длина шага", или step length), обычно обозначается греческой буквой α. Универсальная запись k-го шага обновления с параметром длины шага выглядит так:

$$\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}. \tag{2.20}$$

Вся последовательность k шагов в этом случае записывается как

$$\begin{aligned} &\mathbf{w}^0; \\ &\mathbf{w}^1 = \mathbf{w}^0 + \alpha \mathbf{d}^0; \\ &\mathbf{w}^2 = \mathbf{w}^1 + \alpha \mathbf{d}^1; \\ &\ldots \\ &\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}. \end{aligned} \tag{2.21}$$

Обратите внимание, что единственное различие между этой формой для k-го шага и оригиналом заключается в том, что теперь мы шкалируем направление спуска $\mathbf{d}^{k-1}$ по параметру длины шага $\alpha > 0$. С добавлением этого параметра расстояние, пройденное на k-м шаге универсальной схемы локальной оптимизации, может быть вычислено как

$$\left\| \mathbf{w}^k - \mathbf{w}^{k-1} \right\|_2 = \left\| \left(\mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1} \right) \right\|_2 = \alpha \left\| \mathbf{d}^{k-1} \right\|_2. \tag{2.22}$$

Другими словами, длина k-го шага теперь *пропорциональна* длине вектора спуска, и мы можем точно настроить дальность, на которую хотим перемещаться в этом направлении, надлежаще установив значение α. Общепринято устанавливать α равным некоторому *фиксированному* малому значению для каждого из K шагов. Однако (как и в случае самих методов локальной оптимизации) задать параметр длины шага можно целым рядом способов, и они станут предметом дальнейшего обсуждения в настоящей и последующих главах.

2.5. Случайный поиск

В данном разделе мы опишем наш первый алгоритм локальной оптимизации — *случайный поиск*. В этом примере общего каркаса локальной оптимизации мы отыскиваем направление спуска на каждом шаге путем обследования некоторого числа случайных направлений, выходящих из нашей текущей точки. Этот способ определения направления спуска, во многом похожий на схему глобальной оптимизации из *разд. 2.3*, плохо масштабируется с размером входных данных, что дисквалифи-

[2] А также темпом обучения, скоростью заучивания (от *англ.* learning rate). — *Прим. пер.*

цирует случайный поиск в качестве инструмента для использования в современных крупномасштабных задачах машинного обучения. Однако этот подход нулевого порядка к локальной оптимизации чрезвычайно полезен в качестве простого примера общего каркаса, введенного ранее, и предоставляет нам возможность продемонстрировать простой, но конкретный алгоритмический пример повсеместно присутствующих идей, таких как *направления спуска*, различные *варианты параметра длины* шага и трудности *схождения*.

2.5.1. Картина в целом

Определяющая характеристика *случайного поиска* (как и в случае любой главенствующей схемы локальной оптимизации) заключается в том, как выбирается направление спуска $\mathbf{d}^{k-1}$ на k-м шаге обновления $\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^{k-1}$ локальной оптимизации.

С помощью случайного поиска мы делаем (возможно) самую "ленивую" вещь, которую можно было бы придумать, чтобы отыскать направление спуска: мы отбираем заданное число случайных направлений, выходящих из $\mathbf{w}^{k-1}$, вычисляем значение каждой кандидатной точки обновления и выбираем ту, которая дает нам наименьшее вычисленное значение (если она действительно ниже на функции, чем наша текущая точка). Другими словами, мы ищем локально вокруг текущей точки, в некотором числе случайных направлений, точку, которая имеет более низкое вычисленное значение, и если мы ее находим, то мы перемещаемся к ней.

Выражаясь точнее, на k-м шаге мы генерируем P случайных направлений $\{\mathbf{d}^p\}_{p=1}^P$ для опробования, каждое из которых выходит из предыдущего шага $\mathbf{w}^{k-1}$ и ведет к кандидатной точке $\mathbf{w}^{k-1} + \mathbf{d}^p$ для $p = 1, 2, ..., P$.

После вычисления значения всех таких P кандидатных точек мы берем ту, которая дает нам наименьшее вычисленное значение, т. е. ту, которая имеет индекс, задаваемый формулой

$$s = \underset{p=1,2,...,P}{\arg\min}\, g\left(\mathbf{w}^{k-1} + \mathbf{d}^p\right). \tag{2.23}$$

Наконец, если найденная наилучшая точка имеет меньшее вычисленное значение, чем текущая точка, т. е. если $g\left(\mathbf{w}^{k-1} + \mathbf{d}^s\right) < g\left(\mathbf{w}^{k-1}\right)$, то мы переходим к новой точке $\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^s$, в противном случае мы либо останавливаем метод, либо пробуем другой пакет P случайных направлений.

Метод случайного поиска проиллюстрирован на рис. 2.6 с использованием квадратичной функции, где для целей визуализации число случайных направлений, которые нужно апробировать, задано относительно малым и равняется $P = 3$.

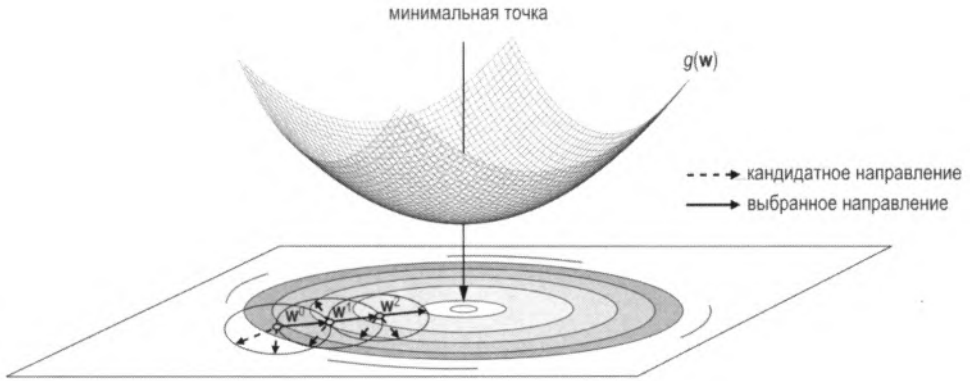


Рис. 2.6. На каждом шаге алгоритм случайного поиска определяет направление спуска, обследуя некоторое число случайных направлений. Направление, ведущее к новой точке с наименьшим вычисленным значением, выбирается в качестве направления спуска, и процесс повторяется до тех пор, пока не будет найдена точка вблизи локального минимума. Здесь показаны три прототипических шага, выполняемых случайным поиском, где на каждом шаге обследуются три случайных направления. На каждом шаге наилучшее найденное направление спуска нарисовано сплошной черной стрелкой, в то время как два других нижних направления показаны пунктирными черными стрелками

2.5.2. Контроль за длиной каждого шага

Для того чтобы лучше контролировать ход случайного поиска, мы можем *нормализовать* каждое случайно выбираемое направление до единичной длины, т. е. $\|\mathbf{d}\|_2 = 1$. Благодаря этому мы можем корректировать каждый шаг так, чтобы он имел любую желаемую длину, введя параметр длины шага α (как описано в *разд. 2.4.3*). Этот более общий шаг $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\mathbf{d}$ теперь имеет длину, в точности равную параметру длины шага α, как

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \|\alpha\mathbf{d}\|_2 = \alpha\|\mathbf{d}\|_2 . \tag{2.24}$$

Пример 2.3. Минимизирование простой квадратичной функции с помощью случайного поиска

В этом примере мы выполняем случайный локальный поиск для $K = 5$ шагов с $\alpha = 1$ для всех шагов, на каждом шаге отыскивая $P = 1000$ случайных направлений, чтобы минимизировать квадратичную функцию:

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2 . \tag{2.25}$$

На рис. 2.7 показана функция в трех размерностях на левой верхней панели вместе с набором шагов, производимых алгоритмом, окрашенным от зеленого (в начале прогона, где мы инициализируем в точке $\mathbf{w}^0 = [3 \quad 4]^T$) до красного цвета (когда алгоритм останавливается). Стрелки иллюстрируют каждое выбранное направление спуска, соединяя каждый шаг с его предшественником, и показаны, чтобы помочь

проиллюстрировать суммарный путь, который принимает алгоритм. На рис. 2.7 справа показана та же самая функция, но рассматриваемая непосредственно сверху, как ее *контурный график*.

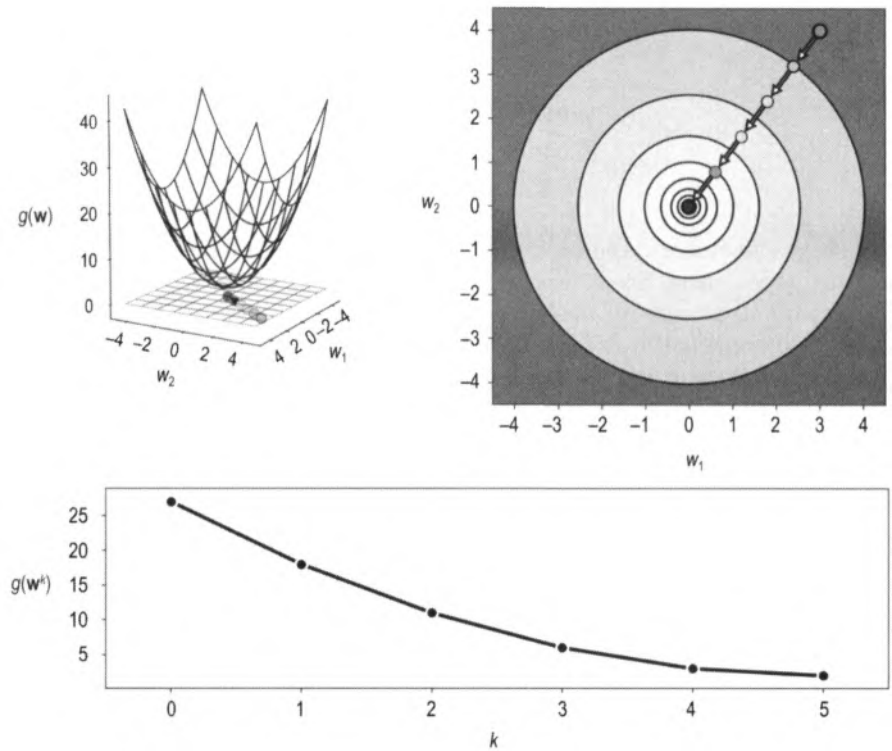


Рис. 2.7. Иллюстрация примера 2.3

Обратите внимание, что, если размерность входа N больше 2, то мы не можем построить график, подобный показанному на рисунке, чтобы понять, насколько хорошо выполняется конкретный прогон любой локальной оптимизации (здесь случайный поиск). Более общий способ визуализации прогрессии, достигнутой локальным методом, заключается в построении графика соответствующей последовательности вычисленных функций по номеру шага, т. е. построении графика пар $\left(k, g(\mathbf{w}^k)\right)$ для $k = 1, 2, ..., K$ (рис. 2.7, *снизу*). Такой график позволяет нам сделать вывод о том (независимо от входной размерности N минимизируемой функции), насколько хорошо алгоритм работает и нужно ли нам корректировать какие-либо его параметры (например, максимальное число шагов K или значение α). Эта визуализация называется *графиком истории функции потерь*. Дополнительной выгодой такого графика является то, что мы можем легче определить точное значение каждого вычисления функции во время прогона локальной оптимизации.

Пример 2.4. Минимизирование функции с многочисленными локальными минимумами

В этом примере мы покажем, что нужно сделать, чтобы отыскать глобальный минимум функции, используя схему локальной оптимизации, такую как случайный поиск. Для визуализации воспользуемся функцией одной переменной

$$g(w) = \sin(3w) + 0{,}3w^2 \tag{2.26}$$

и инициализируем два прогона, один из которых начинается в точке $w^0 = 4{,}5$, а другой — в точке $w^0 = -1{,}5$. Для обоих прогонов зададим длину шага $\alpha = 0{,}1$, фиксированную для всех $K = 10$ шагов или итераций. В зависимости от того, где мы инициализируем процесс, мы можем оказаться вблизи локального (рис. 2.8, *слева*) либо глобального (рис. 2.8, *справа*) минимума. Для данного примера шаги каждого прогона отмечены кружками вдоль входной оси с соответствующими значениями самой функции в виде аналогично окрашенных ×-отметок зеленого цвета в начале прогона до красного, когда прогон останавливается.

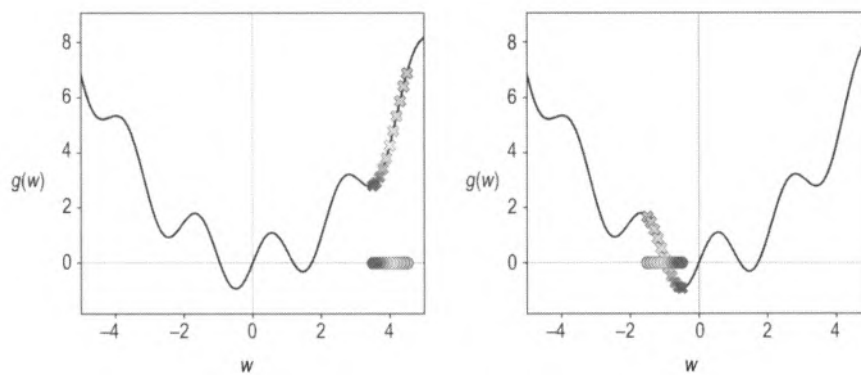


Рис. 2.8. Иллюстрация примера 2.4. Надлежащее минимизирование такой невыпуклой функции, как показанная здесь, требует нескольких прогонов локальной оптимизации с разными инициализациями

2.5.3. Разведывание фундаментальных правил длины шага

В примерах 2.3 и 2.4 мы устанавливали параметр длины шага α фиксированным для всех шагов каждого прогона. Именуемый *правилом фиксированной длины* шага, такой подход является очень распространенным вариантом выбора параметра длины шага для методов локальной оптимизации в целом. Также существует возможность изменять значение α от одного шага к другому с помощью *правила корректируемой длины шага*. Прежде чем разведать очень распространенное правило корректируемой длины шага, используемое в машинном обучении, под названием *правила убывающей длины шага*, сначала покажем важность настройки длины шага на простом примере.

Пример 2.5. Неспособность к схождению

В этом примере мы используем случайный поиск для минимизирования квадратичной функции

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2, \tag{2.27}$$

используя длину шага $\alpha = 1$ (как и в примере 2.3), но с другой инициализацией — в точке $\mathbf{w}^0 = [1{,}5 \quad 2]^T$. Однако при такой инициализации, как показано на контурном графике этой функции на рис. 2.9, *слева*, алгоритм застревает в неоптимальной точке (показана красным цветом) вдали от глобально минимальной точки, расположенной в начале координат, где контурный график здесь показан без цвета в целях более четкой визуализации. Кроме того, на том же графике нарисована синяя единичная окружность с центром в финальной красной точке, охватывающая все возможные точки, к которым алгоритм мог бы нас привести, если бы мы решили сделать еще один шаг и переместиться оттуда, где он останавливается в красной точке. Обратите внимание, как эту синюю окружность охватывает один из контуров квадратичной функции (красным пунктиром), на котором лежит финальная красная точка. Из этого следует, что любое возможное направление обеспечивает подъем, а не спуск, и поэтому алгоритм должен остановиться.

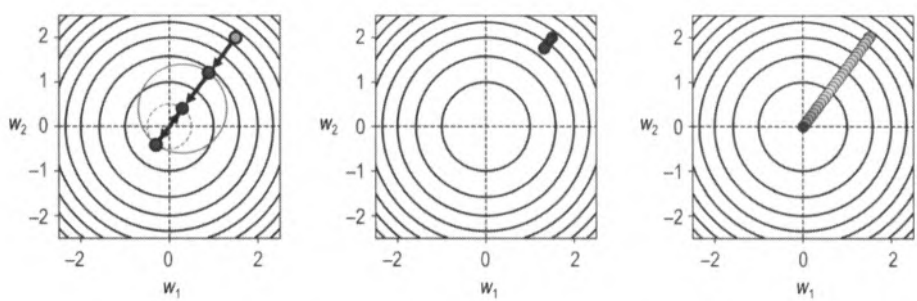


Рис. 2.9. Иллюстрация примера 2.5. Определение надлежащей длины шага имеет решающее значение для оптимальной результативности случайного поиска и, как следствие, многих алгоритмов локальной оптимизации. Здесь выбор слишком большой длины шага приводит к остановке метода в неоптимальной точке (*слева*), установка его слишком малой приводит к очень медленному схождению к минимуму функции (*в центре*), а установка его "в самый раз" способствует идеальному схождению к точке, близкой к минимуму функции (*справа*)

Мы должны проявлять осторожность при выборе значения длины шага с этой простой квадратичной функцией и, следовательно, любой общей функцией. Если, как показано на рис. 2.9, *в центре*, мы повторим тот же самый эксперимент, но уменьшим длину шага до $\alpha = 0{,}01$, то не достигнем точки, близкой к глобальному минимуму, в течение того же самого числа шагов.

Установив параметр длины шага немного больше ($\alpha = 0{,}1$) для всех шагов, мы делаем еще один прогон, зеркально отражающий предыдущий, — его результаты показаны на рис. 2.9, *справа*. Алгоритм теперь сходится к точке, гораздо более близкой к глобальному минимуму функции, в начале координат.

В общем случае сочетание длины шага и максимального числа итераций лучше всего выбирать вместе. Компромисс здесь прост: малая длина шага в сочетании с большим числом шагов может гарантировать схождение к локальному минимуму, но может обходиться вычислительно дорого. И наоборот, большая длина шага и малое максимальное число итераций могут стоить дешевле, но быть менее эффективными при поиске оптимума. Часто на практике такого рода выбор осуществляется путем выполнения нескольких прогонов алгоритма и построения графика соответствующих историй функций потерь для определения оптимальных параметрических настроек.

2.5.4. Правила убывающей длины шага

Часто используемой альтернативой правилам фиксированной длины шага являются так называемые *правила убывающей длины шага*, в которых мы сжимаем или *уменьшаем* размер длины шага на каждом шаге локальной оптимизации. Один из распространенных способов имплементации правила убывающей длины шага — просто устанавливать $\alpha = 1/k$ на k-м шаге процесса. Этим обеспечивается выгода от сжатия расстояния между последующими шагами по мере прогрессии в прогоне алгоритма, т. к. при таком выборе длины шага и векторе направления спуска единичной длины мы имеем

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \alpha\left\|\mathbf{d}^{k-1}\right\|_2 = \frac{1}{k}. \tag{2.28}$$

В то же время если мы просуммируем общее расстояние, пройденное алгоритмом за K шагов (при условии, что мы действительно движемся на каждом шаге), то получим

$$\sum_{k=1}^{K}\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \sum_{k=1}^{K}\frac{1}{k}. \tag{2.29}$$

Красота такого рода уменьшения длины шага заключается в том, что в то время как длина шага $\alpha = 1/k$ стремится к нулю по мере увеличения k, суммарное расстояние, пройденное алгоритмом, уходит в бесконечность[3]. Это означает, что локальный алгоритм, задействующий такого рода правило убывающей длины шага, теоретически может перемещаться на бесконечное расстояние в поисках минимума, все время делая все меньшие и меньшие шаги, что позволяет ему работать в любых малых закоулках и трещинах, которые могут быть у функции там, где лежит любой минимум.

---

[3] Сумма $\sum_{k=1}^{\infty} 1/k$ часто называется *гармоническим рядом*, и один из способов увидеть, что она расходится до бесконечности, состоит в том, чтобы соединить вместе последующие члены как

$$\sum_{k=1}^{\infty} 1/k = 1 + \frac{1}{2} + \left(\frac{1}{3} + \frac{1}{4}\right) + \left(\frac{1}{5} + \frac{1}{6} + \frac{1}{7}\right) + \ldots \geq 1 + \frac{1}{2} + 2\frac{1}{4} + 4\frac{1}{8} + \ldots = 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \ldots \tag{2.30}$$

Другими словами, гармонический ряд ограничен снизу бесконечной суммой из значений 1/2 и, отсюда, расходится до бесконечности.

2.5.5. Случайный поиск и проклятие размерности

Как и в случае глобально-оптимизационного подхода, рассмотренного в *разд. 2.3*, проклятие размерности также представляет собой главенствующее препятствие для практического применения алгоритма случайного поиска по мере увеличения размерности входов функции. Другими словами, для большинства функций становится *экспоненциально* труднее отыскать *произвольное* направление спуска в заданной точке по мере увеличения ее входной размерности.

Возьмем, например, квадратичную функцию одной переменной $g(w) = w^2 + 2$ и предположим, что мы делаем один шаг, используя алгоритм случайного поиска, начинающийся в $w^0 = 1$, с длиной шага $\alpha = 1$. Как показано на рис. 2.10, *сверху*, поскольку входная размерность в этом случае равна $N = 1$, для определения направления спуска нам нужно рассмотреть только два направления: отрицательное и положительное направления от нашей начальной точки. Одно из них обеспечит спуск (здесь отрицательное направление). Другими словами, у нас был бы 50-процентный шанс найти направление спуска, если бы мы выбрали его наугад.

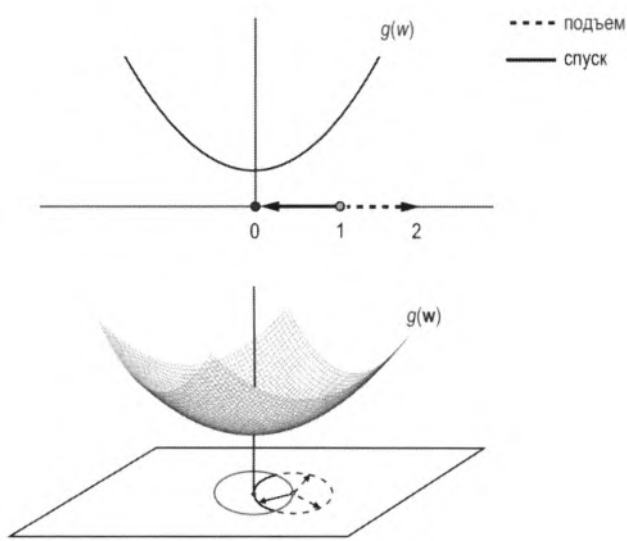


Рис. 2.10. Когда входная размерность равна $N = 1$, мы можем двигаться лишь в двух единичных направлениях, только одно из которых (сплошная стрелка) является направлением спуска (*сверху*). Когда входная размерность равна $N = 2$, существует бесконечно много единичных направлений на выбор, только часть из которых, чья конечная точка лежит внутри единичной окружности (точки на твердой части дуги), является направлениями спуска (*снизу*)

Теперь давайте обследуем тот же вид квадратичной функции, но на этот раз функция принимает в качестве входа двумерный вектор $\mathbf{w}$: $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2$, и представим себе, что мы делаем один шаг случайного поиска, начинающийся в $\mathbf{w}^0 = [1 \quad 0]^T$, двумерном аналоге инициализации, которую мы использовали для

нашей одномерной квадратичной функции. Как показано на рис. 2.10, *снизу*, теперь у нас бесконечно много направлений для выбора, но только часть из них (менее 50%) обеспечивает спуск. Другими словами, в двух размерностях вероятность случайного выбора направления спуска падает относительно аналогичного ему значения в одной размерности. Это уменьшение вероятности случайного выбора направления спуска *экспоненциально* уменьшается по мере увеличения входной размерности N этой квадратичной функции. И действительно, можно вычислить, что для общего N вероятность выбора направления спуска случайна, начиная с

$$\mathbf{w}^0 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{N\times 1}, \tag{2.31}$$

для рассматриваемой квадратичной функции ограничена сверху $\frac{1}{2}\left(\frac{\sqrt{3}}{2}\right)^{N-1}$ (см. упражнение 2.5). Это означает, например, что при $N = 30$ вероятность спуска падает ниже 1%, что делает случайный поиск невероятно неэффективным для минимизирования даже простой квадратичной функции.

2.6. Координатный поиск и спуск

Алгоритмы координатного поиска и спуска являются дополнительными локальными прямыми методами (нулевого порядка), которые обходят присущие им проблемы масштабирования случайного локального поиска, ограничивая набор направлений поиска координатными осями входного пространства. Указанная концепция проста: случайный поиск был разработан для минимизирования функции $g(w_1, w_2, ..., w_N)$ по отношению ко всем ее параметрам *одновременно*. С помощью координатных алгоритмов мы пытаемся минимизировать такую функцию относительно одной координаты или параметра за раз (или в более широком смысле одного подмножества координат или параметров за раз), оставляя все остальные фиксированными.

Хотя это ограничивает разнообразие направлений спуска, которые потенциально могут быть обнаружены, и поэтому часто требуется больше шагов для определения приближенных минимумов, эти алгоритмы гораздо более масштабируемы, чем случайный поиск.

2.6.1. Координатный поиск

Как показано на рис. 2.11, *слева* для прототипического двумерного примера, в случае *координатного поиска* мы ищем направление спуска случайно только среди координатных осей входного пространства. В общем случае это означает, что для функции входной размерности N мы смотрим только на $2N$ направлений из множе-

ства $\left\{\pm\mathbf{e}_n\right\}_{n=1}^{N}$, где $\mathbf{e}_n$ — это *стандартный базисный вектор*, элементы которого равны нулю, за исключением его n-го элемента, который установлен равным 1.

Именно этот ограниченный набор направлений, среди которых мы производим поиск, отличает подход на основе координатного поиска от подхода на основе случайного поиска, описанного в предыдущем разделе, где множество направлений на каждом шаге было составлено из случайных направлений. В то время как разнообразие (или его отсутствие) координатных осей может ограничить эффективность возможных направлений спуска, с которыми он может столкнуться (и, следовательно, требуется больше шагов для определения приближенного минимума), ограниченный поиск делает координатный поиск гораздо более масштабируемым, чем метод случайного поиска, поскольку на каждом шаге необходимо проверять только $2N$ направлений.

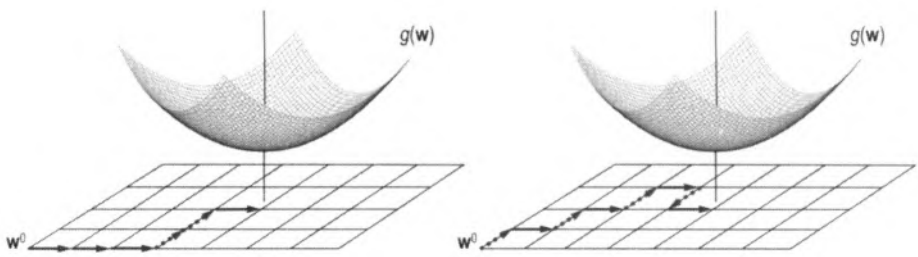


Рис. 2.11. При *координатном поиске* мы ищем направления спуска только среди координатных осей: на каждом шаге, обозначенном сплошной и пунктирной стрелками, мы пробуем $2N = 4$ направления вдоль $N = 2$ координатных осей и подбираем то, которое приводит к наибольшему уменьшению значения функции (*слева*). В случае *координатного спуска* мы (в идеале) делаем шаг сразу после обследования положительного и отрицательного направлений вдоль каждой координаты (*справа*)

2.6.2. Координатный спуск

Незначительное видоизменение в координатном поиске дает гораздо более эффективный алгоритм точно с такой же вычислительной стоимостью, который называется *координатным спуском*. Вместо того чтобы собирать каждое координатное направление (вместе с его отрицательной версией), а затем выбирать одно наилучшее направление из всего этого множества, мы можем просто обследовать одно координатное направление (и его отрицательное) за один раз и шагнуть в этом направлении, если оно производит спуск. Эта идея проиллюстрирована на рис. 2.11, *справа*.

В то время как при координатном поиске мы вычисляем функцию потерь $2N$ раз (дважды на координату), чтобы произвести один шаг, этот альтернативный подход использует то же самое число вычислений функции, но при этом потенциально перемещает на N шагов. Другими словами, в случае координатного спуска мы можем за те же деньги, что и координатный поиск, потенциально минимизировать функцию гораздо быстрее. Действительно, из всех прямых методов (нулевого порядка),

подробно описанных в этой главе, координатный спуск является наиболее практичным.

Пример 2.6. Координатный поиск против координатного спуска

В этом примере мы сравним эффективность алгоритмов координатного поиска и координатного спуска на примере простой квадратичной функции

$$g(w_1, w_2) = 0,26\left(w_1^2 + w_2^2\right) - 0,48 w_1 w_2 . \tag{2.32}$$

На рис. 2.12 сравниваются 20 шагов координатного поиска (*слева*) и координатного спуска (*справа*), которые используют убывающую длину шага. Поскольку координатный спуск на практике предпринимает два действия для каждого отдельного шага, предпринимаемого координатным поиском, мы значительно приближаемся к минимуму функции, используя то же самое число вычислений функции.

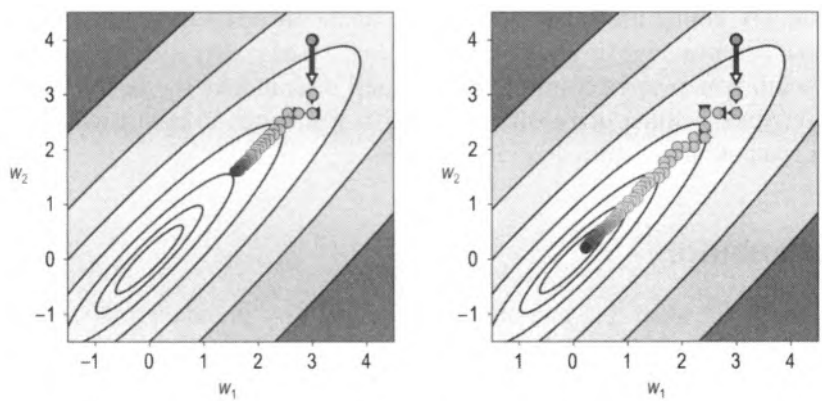


Рис. 2.12. Иллюстрация примера 2.6. Прогоны координатного поиска (*слева*) и координатного спуска (*справа*), задействованные для минимизации одной и той же функции. В то время как оба прогона имеют одинаковую вычислительную стоимость, координатный спуск достигает значительно большей прогрессии

2.7. Заключение

Эта глава заложила основу для широкого спектра фундаментальных идей, связанных с математической оптимизацией (мотивированных в *разд. 1.4*), с которыми мы неоднократно встретимся не только в следующих двух главах, но и во всем остальном тексте книги.

Мы начали с введения концепции математической оптимизации, которая представляет собой математический/вычислительный поиск минимумов или максимумов функции. Затем в *разд. 2.2* мы перевели наше неформальное определение минимумов и максимумов функции на язык математики, причем этот формальный перевод называется *условием оптимальности нулевого порядка*. Опираясь на это определение, мы затем описали методы глобальной оптимизации (по существу, сырое вы-

числение функции над мелкой решеткой входных точек) в *разд. 2.3*, которые, как мы убедились, — по причине проклятия размерности — очень плохо масштабируются с позиций входной размерности и поэтому часто являются не очень практичными. В *разд. 2.4* представлена альтернатива этому ограниченному каркасу — *локальная оптимизация*, охватывающая огромное семейство алгоритмов, которые мы обсудим в оставшейся части этой главы, а также в двух последующих главах. Наконец, в *разд. 2.5* и *2.6* мы рассмотрели ряд примеров алгоритмов локальной прямой оптимизации (нулевого порядка), включая случайный поиск и координатный поиск/спуск. Хотя эти последние методы могут быть очень полезными в конкретных приложениях, в целом эти локальные прямые методы (нулевого порядка) не так широко применяются в машинном обучении по сравнению с теми, которые мы увидим в последующих главах и которые используют первую и/или вторую производные функции для более быстрого формулирования направлений спуска (вместо прямого поиска, требуемого алгоритмами нулевого порядка). Однако относительная простота методов нулевого порядка позволила нам вычеркнуть целый ряд критически важных концепций, связанных с локальной оптимизацией, — идеи, которые, как мы увидим, будут многократно отдаваться эхом в последующих главах в сравнительно неперегруженной окружающей обстановке — включая понятия направлений спуска, длины шага/интенсивности усвоения, схемы с убывающей длиной шага и графиков истории функции потерь.

2.8. Упражнения

† Данные, необходимые для выполнения следующих упражнений, можно скачать из репозитория книги на github по адресу:
github.com/jermwatt/machine_learning_refined.

2.1. Минимизирование квадратичной функции и проклятие размерности

Рассмотрим простую квадратичную функцию:

$$g(\mathbf{w}) = \mathbf{w}^T\mathbf{w}, \tag{2.33}$$

чей минимум всегда находится в начале координат независимо от входной размерности N.

А) Создать диапазон этих квадратичных функций для входной размерности от $N = 1$ до $N = 100$, выбрать из входного пространства каждой по $P = 100$ раз равномерно на гиперкубе $[-1; 1] \times [-1; 1] \times \cdots \times [-1; 1]$ (этот гиперкуб имеет N сторон) и построить график минимального значения, достигаемого для каждой квадратичной функции, против входной размерности N.

Б) Повторить часть (А), используя выборки $P = 100$, $P = 1000$ и $P = 10\,000$, и построить графики всех трех кривых на одном и том же рисунке. Какого рода тренд вы сможете увидеть на этом графике по мере роста N и P?

В) Повторить части (А) и (Б), на этот раз заменив равномерно выбираемые выборки случайными.

2.2. Имплементирование случайного поиска на Python

Имплементировать алгоритм случайного поиска на Python и повторить эксперимент, описанный в примере 2.4.

2.3. Использование случайного поиска для минимизирования невыпуклой функции

Использовать свою имплементацию случайного поиска из упражнения 2.2, чтобы минимизировать вот эту функцию:

$$g(w_1, w_2) = \tanh(4w_1 + 4w_2) + \max\left(0, 4w_1^2, 1\right) + 1. \tag{2.34}$$

Сделать максимум восемь шагов и выполнить поиск в $P = 1000$ случайных направлениях на каждом шаге с длиной шага $\alpha = 1$ и начальной точкой $\mathbf{w}^0 = [2 \quad 2]^T$.

2.4. Случайный поиск с убывающей длиной шага

В этом упражнении вы будете использовать случайный поиск и убывающую длину шага, чтобы минимизировать *функцию Розенброка*:

$$g(w_1, w_2) = 100\left(w^2 - w_1^2\right)^2 + \left(w_1 - 1\right)^2. \tag{2.35}$$

Эта функция имеет глобальный минимум в точке $\mathbf{w}^* = [1 \quad 1]^T$, расположенной в очень узкой и изогнутой долине.

Сделать два прогона случайного поиска, используя $P = 1000$, начальную точку $\mathbf{w}^0 = [-2 \quad -2]^T$ и $K = 50$ шагов. При первом прогоне использовать фиксированную длину шага $\alpha = 1$, а при втором прогоне — убывающую длину шага, как подробно описано в *разд. 2.5.4*. Сравнить оба прогона, построив контурный график функции потерь (где каждый прогон наносится на графике сверху) либо построив график истории функции потерь.

2.5. Вероятности случайного спуска

Рассмотрим квадратичную функцию $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2$, которую мы стремимся минимизировать с помощью случайного поиска, начиная в $\mathbf{w}^0$, определенного в уравнении (2.31), с $\alpha = 1$ и $\left\|\mathbf{d}^0\right\|_2 = 1$.

А) Когда $N = 2$, показать, что вероятность спуска, т. е. вероятность того, что $g\left(\mathbf{w}^0 + \alpha\mathbf{d}^0\right) < g\left(\mathbf{w}^0\right)$ для случайно выбранного единичного направления $\mathbf{d}^0$ ограничена сверху $\sqrt{3}/4$. Подсказка: см. рис. 2.13.

Б) Расширить свой аргумент в части (А), чтобы найти верхнюю границу вероятности спуска для общего N.

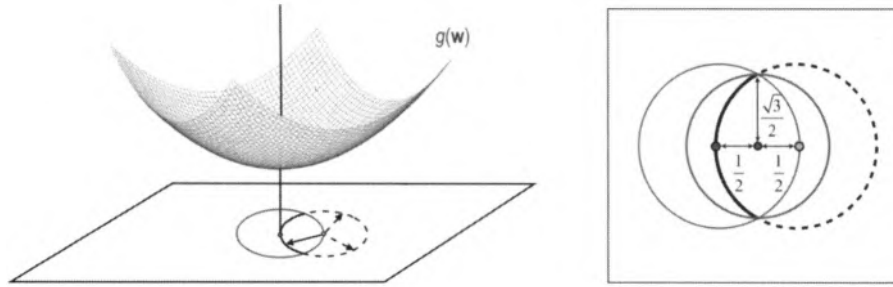


Рис. 2.13. Иллюстрация упражнения 2.5

2.6. Проклятие размерности повторно

В этом упражнении следует эмпирически подтвердить задачу о проклятии размерности, описанную в *разд. 2.5.5*, для простой квадратичной функции $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2$.

Начиная с N-мерной входной точки $\mathbf{w}^0$, определенной в уравнении (2.31), создать P случайных единичных направлений $\left\{\mathbf{d}^p\right\}_{p=1}^{P}$ (где $\left\|\mathbf{d}^p\right\|_2 = 1$) и вычислить точку $\mathbf{w}^0 + \mathbf{d}^p$ посредством квадратичной функции для всех p.

Затем создать график, иллюстрирующий ту часть отобранных направлений, которая обеспечивает уменьшение в вычисленном значении функции (т. е. *числа направлений спуска*, деленных на P) относительно N (для $N = 1, 2, ..., 25$) и для трех значений P: $P = 100$, $P = 1000$ и $P = 10\,000$. Описать тренд, который вы наблюдаете.

2.7. Псевдокод для алгоритма координатного поиска

Разработать псевдокод для алгоритма координатного поиска, описанного в *разд. 2.6.1*.

2.8. Координатный поиск применительно к минимизированию простой квадратичной функции

Сравнить пять шагов алгоритма случайного поиска (с проверкой $P = 1000$ случайных направлений на каждом шаге) с семью шагами координатного поиска, используя одинаковую начальную точку $\mathbf{w}^0 = [3 \quad 4]^T$ и значение параметра фиксированной длины шага $\alpha = 1$, чтобы минимизировать квадратичную функцию:

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2. \tag{2.36}$$

Построить график этой функции вместе с результирующими прогонами обоих алгоритмов и описать любые различия в поведении между этими двумя прогонами.

2.9. Координатный поиск с убывающей длиной шага

Имплементировать алгоритм координатного поиска, подробно описанный в *разд. 2.6.1*, и применить его для минимизирования функции

$$g(w_1, w_2) = 0,26 \cdot \left(w_1^2 + w_2^2\right) - 0,48 w_1 w_2, \tag{2.37}$$

используя правило убывающей длины шага, начиная со случайной начальной точки. Глобальный минимум этой функции лежит в начале координат. Проверить свою имплементацию, убедившись, что она способна достичь точки, значительно близкой (например, в пределах 10^{-2}) к исходной точке из различных случайных инициализаций.

2.10. Координатный поиск против координатного спуска

Имплементировать алгоритмы координатного поиска и координатного спуска и повторить эксперимент, описанный в примере 2.6.

3 Приемы оптимизации первого порядка

3.1. Введение

В этой главе мы опишем фундаментальные алгоритмы оптимизации, которые используют *первую производную* или *градиент* функции. Эти приемы, коллективно именуемые *методами оптимизации первого порядка*, являются одними из наиболее популярных алгоритмов локальной оптимизации, используемых сегодня для решения задач машинного обучения. Мы начнем с обсуждения *условия оптимальности первого порядка*, которое кодифицирует то, как первая производная функции характеризует ее минимумы. Затем мы обсудим фундаментальные понятия, связанные с гиперплоскостями, и в частности аппроксимацию с использованием ряда Тейлора первого порядка. Как мы увидим, задействуя информацию о первой производной функции, мы можем воспользоваться методами локальной оптимизации, прежде всего чрезвычайно популярным *алгоритмом градиентного спуска*, который естественным образом определяет высококачественные направления спуска за цену, которая очень часто меньше цены даже координатно-ориентированных подходов, описанных в предыдущей главе.

3.2. Условие оптимальности первого порядка

На рис. 3.1 представлены две простые квадратичные функции: одна в двух размерностях (*слева*), другая в трех размерностях (справа). Глобальный минимум отмечен зеленой точкой на каждом графике. Кроме того, на графиках построена либо линия, либо гиперплоскость, касательные к функциям в их минимальной точке, которые называются *аппроксимацией с использованием ряда Тейлора первого порядка* (*см. разд. П2.9*, если вы не знакомы с понятием тейлоровой аппроксимации). И прямая, и гиперплоскость являются совершенно плоскими. Этот тип поведения универсален (для дифференцируемых функций) независимо от рассматриваемой функции, и он сохраняется независимо от размерности входа функции. То есть минимальные значения функции естественным образом расположены на *дне долины*, где касательная или гиперплоскость является совершенно плоской и, следовательно, имеет нулевой наклон.

Поскольку *производная* (*см. разд. П2.2*) одновходной функции или *градиент* (*см. разд. П2.4*) многовходной функции в точке дают именно эту информацию о наклоне, значения первых производных обеспечивают удобный способ охаракте-

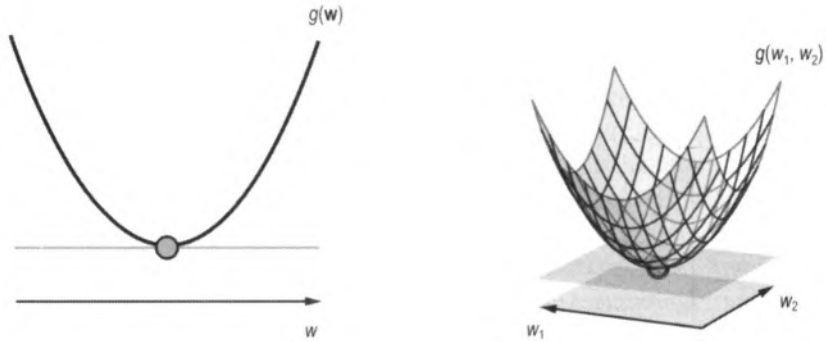


Рис. 3.1. Условие оптимальности первого порядка характеризует точки, в которых производная или градиент функции равны нулю или, другими словами, в которых прямая или гиперплоскость, касательная к функции, имеет нулевой наклон (т. е. она полностью горизонтальна и параллельна входному пространству). С помощью простой выпуклой квадратичной функции это условие идентифицирует глобальный минимум функции, проиллюстрированный здесь с помощью одновходной (*слева*) и многовходной (*справа*) квадратичных функций

ризовать минимальные значения функции g. При $N = 1$ любая точка v одновходной функции $g(w)$, где

$$\frac{d}{dw} g(v) = 0, \tag{3.1}$$

является потенциальным минимумом. Аналогично, для многовходных функций любая N-мерная точка $\mathbf{v}$, где каждая частная производная от g равна нулю, т. е.

$$\begin{aligned} &\frac{\partial}{\partial w_1} g(\mathbf{v}) = 0; \\ &\frac{\partial}{\partial w_2} g(\mathbf{v}) = 0; \\ &\ldots \\ &\frac{\partial}{\partial w_N} g(\mathbf{v}) = 0, \end{aligned} \tag{3.2}$$

является потенциальным минимумом. Эта система N уравнений совершенно естественно называется *системой уравнений дифференциальных уравнений первого порядка*. Можно записать первопорядковую систему компактнее, используя градиентную нотацию:

$$\nabla g(\mathbf{v}) = \mathbf{0}_{N\times 1}. \tag{3.3}$$

Эта очень полезная характеристика минимальных точек является первопорядковым аналогом условия оптимальности нулевого порядка, обсуждаемого в *разд. 2.2*, и поэтому называется *условием оптимальности первого порядка* (или *первопорядковым условием* для краткости). Однако с первопорядковой характеристикой минимумов существуют две проблемы.

Прежде всего, за немногими исключениями (и среди них некоторые интересные примеры, которые мы подробно рассмотрим в *разд. 3.2.1*) практически невозможно решить системы дифференциальных уравнений первого порядка общей функции "вручную" (т. е. решить такие уравнения алгебраически для замкнутой формы). Другая проблема заключается в том, что условие первого порядка определяет глобальные минимумы только для *выпуклых функций*, а для квадратичных функций (см. рис. 3.1) в общем случае это условие справедливо не только для минимумов функции, но и для других особенных точек, включая *максимумы* и *седловые точки невыпуклых функций*. Минимумы, максимумы и седловые точки часто именуются *стационарными*, или *критическими*, точками функции.

Пример 3.1. Визуальный осмотр одновходных функций для стационарных точек

В верхнем ряду на рис. 3.2 приведены графики функций:

$$g(w) = \sin(2w);$$
$$g(w) = w^3; \qquad (3.4)$$
$$g(w) = \sin(3w) + 0,3w^2,$$

а в нижнем ряду — графики производных этих функций. На каждом графике отмечены точки (кружки зеленого цвета), где производная функции равна нулю, и построены касательные, соответствующие каждой такой точке.

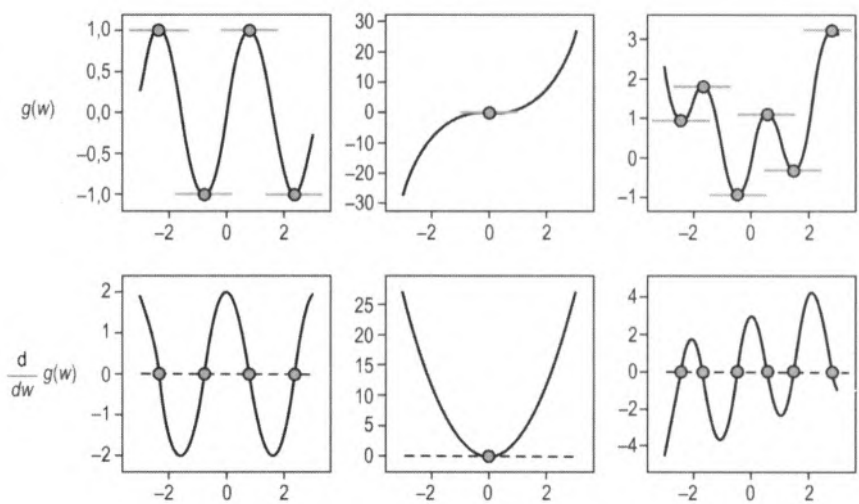


Рис. 3.2. Иллюстрация примера 3.1. Слева направо в верхнем ряду построены графики функций $g(w) = \sin(2w)$, $g(w) = w^3$ и $g(w) = \sin(3w) + 0,3w^2$ вместе с их производными в нижнем ряду

Рассматривая эти графики, легко заметить, что не только *глобальные минимумы* имеют нулевые производные, но и другие разнообразные точки, включая, во-первых, *локальные минимумы*, или точки, которые являются наименьшими по отношению к их ближайшим соседям, например вокруг входного значения $w = -2,5$

на рис. 3.2, *справа сверху*; во-вторых, *локальные* (и *глобальные*) максимумы, или точки, которые являются наибольшими по отношению к их непосредственным соседям, например вокруг входного значения $w = 0{,}5$ на рис. 3.2, *справа сверху*; и в-третьих, *седловые точки*, подобные показанной на рис. 3.2, *сверху в центре*, которые не являются ни максимальными, ни минимальными по отношению к их непосредственным соседям.

3.2.1. Особые случаи, когда система первого порядка может быть решена вручную

В принципе, преимущество использования условия равенства нулю первой производной функции состоит в том, что оно позволяет преобразовывать задачу поиска глобальных минимумов в задачу решения системы уравнений, для которой разработан широкий спектр алгоритмических методов. Здесь акцент на слове "алгоритмический" является ключевым, т. к. решать систему (потенциально нелинейных) уравнений *вручную* очень сложно (если вообще возможно).

Однако есть несколько относительно простых, но важных примеров, когда можно получить решение системы первого порядка вручную или, по крайней мере, показать алгебраически, что они сводятся к *линейной* системе уравнений, которая может быть легко решена численно. Безусловно, наиболее важной из них является многовходная квадратичная функция (см. пример 3.4) и тесно связанное с ней *отношение Рэлея* (см. упражнение 3.3). Эти функции возникают во многих местах при изучении машинного обучения, от фундаментальных моделей, таких как линейная регрессия, архитектурный дизайн второпорядковых алгоритмов, и вплоть до математического анализа алгоритмов.

Пример 3.2. Отыскание стационарных точек одновходных функций посредством условия первого порядка

В этом примере мы используем условие оптимальности первого порядка (когда первая производная функции равна нулю) для вычисления стационарных точек функций:

$$\begin{aligned} g(w) &= w^3; \\ g(w) &= e^w; \\ g(w) &= \sin w; \\ g(w) &= a + bw + cw^2 \quad (c > 0). \end{aligned} \tag{3.5}$$

- $g(w) = w^3$: условие первого порядка дает $\frac{d}{dw} g(v) = 3v^2 = 0$, которое мы можем визуально идентифицировать как седловую точку в $v = 0$ (см. рис. 3.2, *сверху в центре*).
- $g(w) = e^w$: условие первого порядка дает $\frac{d}{dw} g(v) = e^v = 0$, которое удовлетворяется только тогда, когда v стремится к $-\infty$, давая минимум.

- $g(w) = \sin w$: условие первого порядка дает стационарные точки везде, где $\frac{d}{dw} g(v) = \cos v = 0$, что происходит в нечетных целых числах, кратных $\pi/2$, т. е. максимумы в $v = (4k+1)\pi/2$ и минимумы в $v = (4k+3)\pi/2$, где k — это любое целое число.

- $g(w) = a + bw + cw^2$: условие первого порядка дает $\frac{d}{dw} g(v) = 2cv + b = 0$, с минимумом в $v = -b/2c$ (при условии, что $c > 0$).

Пример 3.3. Простая на вид функция

Как упоминалось ранее, подавляющее большинство систем первого порядка невозможно решить вручную алгебраически. Для того чтобы получить представление об этой проблеме, приведем пример достаточно простой функции, глобальный минимум которой не так просто вычислить вручную.

Рассмотрим многочлен четвертой степени

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right) \tag{3.6}$$

и построим его график по узкому диапазону входов, содержащих его глобальный минимум (рис. 3.3).

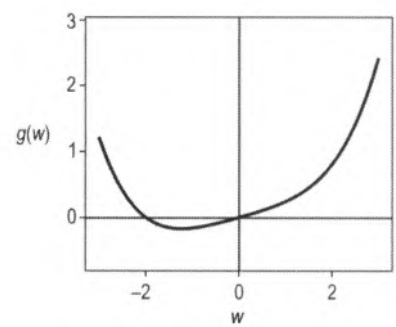


Рис. 3.3. Иллюстрация примера 3.3

Первая производная функции (3.6) такова:

$$\frac{d}{dw} g(w) = \frac{1}{50}\left(4w^3 + 2w + 10\right) = 0, \tag{3.7}$$

что упрощается до

$$2w^3 + w + 5 = 0. \tag{3.8}$$

Это уравнение имеет только одно реальное решение

$$w = \frac{\sqrt[3]{\sqrt{2031} - 45}}{\sqrt[3]{36}} - \frac{1}{\sqrt[3]{6\left(\sqrt{2031} - 45\right)}}, \tag{3.9}$$

которое можно вычислить (после долгих трудов), используя многовековые приемы, разработанные именно для таких задач. На самом деле если бы многочлен в уравнении (3.6) имел шестую степень или выше, то мы не смогли бы гарантировать отыскание его стационарной точки (точек) в замкнутой форме.

Пример 3.4. Стационарные точки многовходных квадратичных функций

Возьмем общую многовходную квадратичную функцию

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}, \tag{3.10}$$

где $\mathbf{C}$ — это симметричная матрица размера $N \times N$; $\mathbf{b}$ — $N \times 1$-вектор; a — скаляр. Вычисляя градиент функции g, мы имеем

$$\nabla g(\mathbf{w}) = 2\mathbf{C}\mathbf{W} + \mathbf{b}. \tag{3.11}$$

Приравняв это нулю, получим симметричную систему линейных уравнений формы

$$\mathbf{C}\mathbf{w} = -\frac{1}{2}\mathbf{b}, \tag{3.12}$$

решение которой является стационарными точками исходной функции. Обратите внимание, что здесь мы явно не занимались поиском решения для этих стационарных точек, а просто показали, что численное решение системы уравнений первого порядка в данном конкретном случае на самом деле — одно из самых простых (см. пример 3.6).

3.2.2. Координатный спуск и условие оптимальности первого порядка

В то время как решить *одновременно* систему первого порядка в уравнении (3.2) часто невозможно, иногда можно решить такую систему *последовательно*. Другими словами, в некоторых (достаточно важных) случаях система первого порядка может быть решена по *одному уравнению за раз*, n-е из которых принимает вид $\frac{\partial}{\partial w_n} g(\mathbf{v}) = 0$. Этот подход, выражающий идею *координатного спуска*, является особенно эффективным, когда каждое из уравнений может быть решено само по себе, вне зависимости от других уравнений (например, когда минимизируемая функция является квадратичной).

Для того чтобы решить систему первого порядка последовательно, нужно выполнить инициализацию во входной точке $\mathbf{w}^0$ и начать с обновления первой координаты путем решения

$$\frac{\partial}{\partial w_1} g\left(\mathbf{w}^0\right) = 0 \tag{3.13}$$

для нахождения первого оптимального веса $w_1^\star$. Важно, что при решении этого уравнения для w_1 все остальные веса остаются фиксированными в своих началь-

ных значениях. Затем следует обновить первую координату вектора $\mathbf{w}^0$ с помощью этого решения $w_1^\star$ и получить обновленный вектор весов $\mathbf{w}^1$. Следуя этой стратегии, для обновления n-го веса нужно решить уравнение

$$\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{n-1}\right) = 0 \tag{3.14}$$

для $w_n^\star$. Опять же, при решении этого уравнения все остальные веса остаются фиксированными в своих текущих значениях. Далее нужно обновить n-й вес с использованием полученного значения, формируя обновленный вектор весов $\mathbf{w}^n$.

После того как мы пройдем через все N весов за один раз, мы можем уточнить наше решение, снова пройдя через веса (как и в любом другом координатном методе). В k-м таком прохождении мы обновляем n-й вес, решая единственное уравнение

$$\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{N(k-1)+n-1}\right) = 0, \tag{3.15}$$

чтобы обновить n-й вес $\mathbf{w}^{N(k-1)+n-1}$ и т. д.

Пример 3.5. Минимизирование выпуклых квадратичных функций посредством координатного спуска

В этом примере мы используем координатный спуск для минимизирования выпуклой квадратичной функции

$$g\left(w_1, w_2\right) = w_1^2 + w_2^2 + 2, \tag{3.16}$$

чей минимум лежит в начале координат. Мы инициализируем прогон в точке $\mathbf{w}^0 = [3 \quad 4]^T$, где один проход по координатам (т. е. два шага) здесь полностью минимизирует функцию. Путь, пройденный этим прогоном, показан на рис. 3.4, *слева* вместе с контурным графиком функции. Можно легко проверить, что каждое урав-

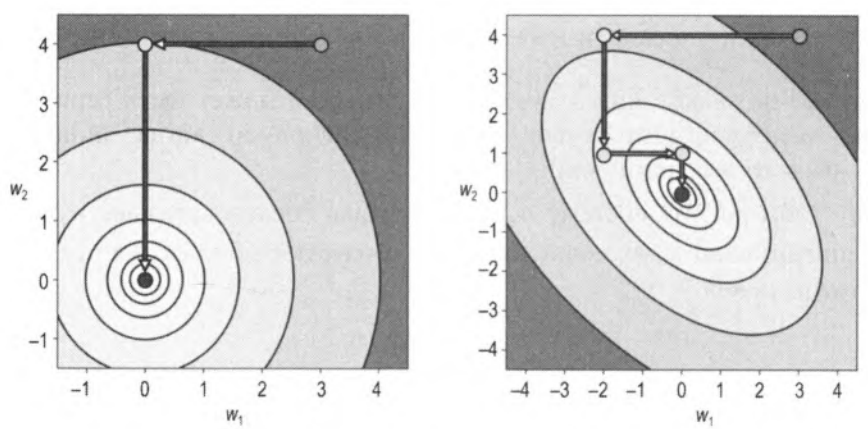


Рис. 3.4. Иллюстрация примера 3.5

нение первого порядка в этом случае является линейным и тривиальным для решения в замкнутой форме.

Затем мы применяем координатный спуск, чтобы минимизировать выпуклую квадратичную функцию

$$g(w_1, w_2) = 2w_1^2 + 2w_2^2 + 2w_1 w_2 + 20, \tag{3.17}$$

чей контурный график показан на рис. 3.4, *справа*. Здесь требуется бесконечно много проходов через переменные, чтобы точно отыскать глобальный минимум (который лежит в начале координат), и по крайней мере три полных прохода, чтобы подобраться к нему достаточно близко.

Пример 3.6. Решение симметричных линейных систем уравнений

В примере 3.4 мы видели, что система первого порядка многовходной квадратичной функции принимает вид

$$\mathbf{C}\mathbf{w} = -\frac{1}{2}\mathbf{b}, \tag{3.18}$$

где **C** — симметричная матрица, как описано в этом примере. Мы можем использовать алгоритм координатного спуска для решения этой системы, благодаря ему минимизируя соответствующую квадратичную функцию. В отрыве от понятия квадратичной функции мы можем думать о координатном спуске в более широком контексте как о методе решения более общих симметричных линейных систем уравнений, который довольно часто встречается на практике, например на каждом шаге метода Ньютона (*см. главу 4*).

3.3. Геометрия ряда Тейлора первого порядка

В этом разделе мы опишем важные характеристики гиперплоскости, включая понятие направления самого *крутого подъема* и *самого крутого спуска*. Затем мы изучим специальную гиперплоскость — аппроксимацию, которая определяет саму суть чрезвычайно популярного алгоритма градиентного спуска, представленного в *разд. 3.5*, аппроксимацию функции с использованием ряда Тейлора первого порядка.

3.3.1. Анатомия гиперплоскостей

Общую N-мерную гиперплоскость можно охарактеризовать как

$$h(w_1, w_2, ..., w_N) = a + b_1 w_1 + b_2 w_2 + ... + b_N w_N, \tag{3.19}$$

где a, b_1, ..., b_N — скалярные параметры. Мы можем переписать функцию h компактнее, используя векторную нотацию:

$$h(\mathbf{w}) = a + \mathbf{b}^T \mathbf{w}, \tag{3.20}$$

обозначив

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} \text{ и } \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{3.21}$$

При $N = 1$ уравнение (3.20) упрощается до формулы

$$h(w) = a + bw, \tag{3.22}$$

которая нам уже знакома для (*одномерной*) прямой. Обратите внимание, что $h(w) = a + bw$ — это одномерный объект, располагающийся в *двумерном* окружающем пространстве, входное пространство которого (характеризуемое членом w) само является одномерным.

То же самое верно и для общего N. То есть $h(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w}$ — это N-мерный математический объект, располагающийся в $(N+1)$-мерном окружающем пространстве, входное пространство которого (характеризуемое w_1, w_2, ..., w_N) является N-мерным.

3.3.2. Направления самого крутого подъема и спуска

Как мы только что увидели, с одномерной гиперплоскостью входное пространство также является одномерным, из чего вытекает, что в любой точке w^0 входного пространства есть только два направления для перемещения: влево или вправо от w^0 (рис. 3.5, *слева*). Видно, что, стартуя из точки w^0 и двигаясь вправо (в направлении $+\infty$), мы *увеличиваем* значение h. Следовательно, это направление является направлением *подъема*. И наоборот, движение влево (в сторону $-\infty$) *уменьшает* h, и, следовательно, это направление является направлением *спуска*.

Однако при $N > 1$ существует бесконечно много направлений для движения (в отличие от всего лишь двух, когда $N = 1$): одни обеспечивают подъем, другие — спуск, а третьи сохраняют значение h (рис. 3.5, *справа* для $N = 2$). Поэтому логично задаться вопросом: можем ли мы найти направление, обеспечивающее наилучший подъем (или спуск), которое принято именовать *направлением самого крутого подъема* (или *спуска*)?

Формализуя поиск направления самого крутого подъема в данной точке $\mathbf{w}^0$, мы стремимся найти такое единичное направление $\mathbf{d}$, что значение $h(\mathbf{w}^0 + \mathbf{d})$ является максимальным. Другими словами, мы стремимся решить задачу

$$\underset{\mathbf{d}}{\text{максимизировать}}\ h\left(\mathbf{w}^0 + \mathbf{d}\right) \tag{3.23}$$

над всеми векторами $\mathbf{d}$ единичной длины. Обратите внимание, что, воспользовавшись формой записи уравнения (3.20), $h(\mathbf{w}^0 + \mathbf{d})$ можно записать как

$$a + \mathbf{b}^T\left(\mathbf{w}^0 + \mathbf{d}\right) = a + \mathbf{b}^T\mathbf{w}^0 + \mathbf{b}^T\mathbf{d}, \tag{3.24}$$

где первые два члена в правой части не зависят от $\mathbf{d}$.

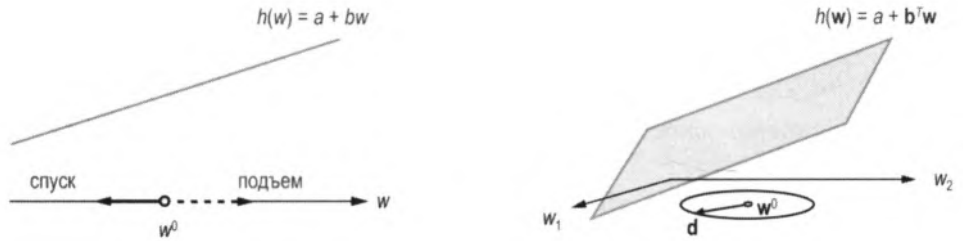


Рис. 3.5. В любой заданной точке w^0 входного пространства одномерной гиперплоскости h существуют только два направления движения: одно увеличивает вычисленное значение h (это направление *подъема*), другое уменьшает его (это направление *спуска*) (*слева*). В пространствах более высоких размерностей (здесь $N = 2$) существует бесконечно много (единичных) направлений $\mathbf{d}$ для перемещения, начиная с заданной N-мерной входной точки $\mathbf{w}^0$ (*справа*). В этом случае конечные точки всех таких направлений образуют единичную окружность с центром в точке $\mathbf{w}^0$

Следовательно, максимизирование значения $h(\mathbf{w}^0 + \mathbf{d})$ эквивалентно максимизированию $\mathbf{b}^T\mathbf{d}$, которое само по себе может быть записано с использованием правила внутреннего произведения[1] (*см. приложение 3*) как

$$\mathbf{b}^T\mathbf{d} = \|\mathbf{b}\|_2 \|\mathbf{d}\|_2 \cos\theta . \tag{3.25}$$

Еще раз обратите внимание на то, что $\|\mathbf{b}\|_2$ (т. е. длина вектора $\mathbf{b}$) не зависит от $\mathbf{d}$ и $\|\mathbf{d}\|_2 = 1$. Поэтому задача в уравнении (3.23) сводится к следующему:

$$\underset{\theta}{\text{максимизировать}}\ \cos\theta , \tag{3.26}$$

где θ — угол между векторами $\mathbf{b}$ и $\mathbf{d}$.

Теперь ясно, что из всех единичных направлений $\mathbf{d} = \dfrac{\mathbf{b}}{\|\mathbf{b}\|_2}$ обеспечивает самый крутой подъем (где $\theta = 0$ и $\cos\theta = 1$).

Аналогично можно показать, что единичное направление $\mathbf{d} = \dfrac{-\mathbf{b}}{\|\mathbf{b}\|_2}$ обеспечивает самый крутой спуск (где $\theta = \pi$ и $\cos\theta = -1$).

---

[1] Для справки: в векторном пространстве внутреннее произведение (inner product), или скалярное произведение — это обобщение точечного произведения в форме умножения векторов, результатом которого является скаляр. Точечное произведение (dot product) определяется для двух векторов $\mathbf{x}$ и $\mathbf{y}$ как $\mathbf{x} \cdot \mathbf{y} = |\mathbf{x}||\mathbf{y}|\cos\theta$, где θ — это угол между векторами и $|\mathbf{x}|$ — норма. Отсюда следует, что $\mathbf{x} \cdot \mathbf{y} = 0$, если $\mathbf{x}$ перпендикулярно $\mathbf{y}$. Поэтому точечное произведение имеет геометрическую интерпретацию как длина проекции $\mathbf{x}$ на единичный вектор $\hat{\mathbf{y}}$, когда два вектора расположены так, что их хвосты совпадают. — *Прим. пер.*

3.3.3. Градиент и направление самого крутого подъема/спуска

Многовходную функцию $g(\mathbf{w})$ можно аппроксимировать локально вокруг заданной точки $\mathbf{w}^0$ гиперплоскостью $h(\mathbf{w})$:

$$h(\mathbf{w}) = g\left(\mathbf{w}^0\right) + \nabla g\left(\mathbf{w}^0\right)^T \left(\mathbf{w} - \mathbf{w}^0\right), \tag{3.27}$$

которую можно переписать как $h(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w}$ (чтобы соответствовать нашей нотации из предыдущего раздела), где

$$a = g\left(\mathbf{w}^0\right) - \nabla g\left(\mathbf{w}^0\right)^T \mathbf{w}^0 \text{ и } \mathbf{b} = \nabla g(\mathbf{w}^0). \tag{3.28}$$

Эта гиперплоскость является аппроксимацией функции g в точке $\mathbf{w}^0$ с использованием *ряда Тейлора первого порядка* и является касательной к g в этой точке (*см. приложение 2*).

Поскольку гиперплоскость h построена так, чтобы можно было близко аппроксимировать g вблизи точки $\mathbf{w}^0$, ее направления самого крутого подъема и спуска также сообщают нам о направлении движения, которое увеличивает или уменьшает значение самой опорной функции g вблизи точки $\mathbf{w}^0$.

3.4. Эффективное вычисление градиентов

Задумайтесь на мгновение о том, как вы выполняете элементарные арифметические операции, например умножение двух чисел. Если эти два числа относительно малы, например, нужно 35 умножить на 21, то вы, вероятно, сможете перемножить их в уме, используя сочетание *свойств умножения* и *результатов простого умножения*, которые вы изучали в начальной школе. Например, вы можете использовать распределительное свойство умножения для разложения $35 \cdot 21$ следующим образом:

$$(30+5)\cdot(20+1) = 30\cdot 20 + 30\cdot 1 + 5\cdot 20 + 5\cdot 1, \tag{3.29}$$

а затем использовать таблицу умножения, чтобы найти $30\cdot 20$, $30\cdot 1$ и т. д. Мы используем такую стратегию ежедневно, когда делаем быстрые подсчеты на "обороте конверта", такие как вычисление процентов по кредиту или инвестициям, вычисление того, сколько нужно дать чаевых в ресторане и т. д.

Однако, несмотря на то, что правила умножения довольно просты и работают независимо от сложности перемножаемых чисел, вы, вероятно, никогда не вычислите произведение двух сколь угодно больших чисел, например 140 283 197 и 2 241 792 341, используя ту же самую стратегию. Вместо этого вы, скорее всего, прибегните к помощи калькулятора, потому что он автоматизирует процесс перемножения двух чисел произвольного размера. Калькулятор позволяет вычислять с гораздо большей эффективностью и точностью, а также дает вам возможность использовать плоды арифметических вычислений для более важных задач.

Именно так можно думать о вычислении производных и градиентов. Если угодно, вы можете легко вычислить производную относительно простой математической функции, такой как $g(w) = \sin(w^2)$, зная сочетание *правил дифференцирования*, а также производные некоторых *элементарных функций*, таких как синусы и многочлены (*см. приложение 2*). В этом конкретном случае вы можете использовать *цепное правило* для записи $\frac{d}{dw} g(w)$ как

$$\frac{d}{dw} g(w) = \left(\frac{d}{dw} w^2 \right) \cos\left(w^2\right) = 2w \cos\left(w^2\right). \tag{3.30}$$

Как и в случае с умножением, хотя правила дифференцирования являются довольно простыми и работают независимо от дифференцируемой функции, вы, вероятно, никогда не вычислите градиент произвольно сложной функции, такой как следующая, самостоятельно и *вручную*:

$$g\left(w_1, w_2\right) = 2^{\sin\left(w_1^2 + w_2^2\right)} \tanh\left(\cos\left(w_1 w_2\right)\right) \tanh\left(w_1 w_1^4 + \tanh\left(w_1^2 + w_2^2\right)\right), \tag{3.31}$$

т. к. это чрезвычайно трудоемко, и легко можно запутаться (как и в случае умножения двух больших чисел). Следуя той же логике, хочется надеяться, что *градиентный калькулятор* будет использоваться для вычисления производных и градиентов с гораздо большей эффективностью и точностью, а значит, позволит нам использовать результаты градиентных вычислений для более важных задач, например для популярных схем *локальной оптимизации* первого порядка, которые являются предметом этой главы и широко используются в машинном обучении. Поэтому, изучая книгу дальше, читатель должен чувствовать себя комфортно, используя *градиентный калькулятор* в качестве альтернативы ручному вычислению.

Градиентные калькуляторы бывают нескольких разновидностей, начиная с тех, которые обеспечивают численные аппроксимации, и заканчивая теми, которые буквально автоматизируют простые правила производной для элементарных функций и операций. Эти различные подходы описаны в *приложении 2*. Для пользователей языка Python мы настоятельно рекомендуем использовать библиотеку автодифференцирования с открытым исходным кодом autograd [10–13] или JAX (расширение библиотеки autograd, работающее на графических и тензорных процессорах). Это высококачественный и простой в использовании профессиональный градиентный калькулятор, который дает пользователю возможность легко вычислять градиент для широкого спектра функций Python, построенных с использованием стандартных структур данных и операций autograd. В *разд. П2.10* представлено краткое учебное руководство, посвященное тому, как начать работу с библиотекой autograd, а также продемонстрированы некоторые ее ключевые функциональные возможности.

3.5. Градиентный спуск

В *разд. 3.3* мы видели, как отрицательный градиент $-\nabla g(\mathbf{w})$ функции $g(\mathbf{w})$, вычисленной в некоторой точке, *всегда* определяет нужное направление спуска в этой точке. Мы вполне естественно могли бы задаться вопросом об эффективности метода локальной оптимизации, состоящего из шагов, выполняемых по формуле $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^k$ (*см. разд. 2.4*), и задействующего направление отрицательного градиента $\mathbf{d}^k = -\nabla g\left(\mathbf{w}^{k-1}\right)$. Такая последовательность шагов приняла бы тогда вид

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right). \tag{3.32}$$

Кажется интуитивно понятным, по крайней мере поначалу, что, поскольку каждое направление гарантированно является направлением спуска (при условии, что мы задаем α соответствующим образом, как мы всегда должны делать при использовании любого метода локальной оптимизации), принятие таких шагов может привести нас к точке, близкой к локальному минимуму целевой функции g. Довольно простой шаг обновления в уравнении (3.32) действительно является чрезвычайно популярным методом локальной оптимизации, именуемым *алгоритмом градиентного спуска*, названным так потому, что он задействует (отрицательный) *градиент* в качестве направления *спуска*.

Траектория, принимаемая градиентным спуском, показана на рис. 3.6 для некой функции одной переменной. На каждом шаге этого метода локальной оптимизации

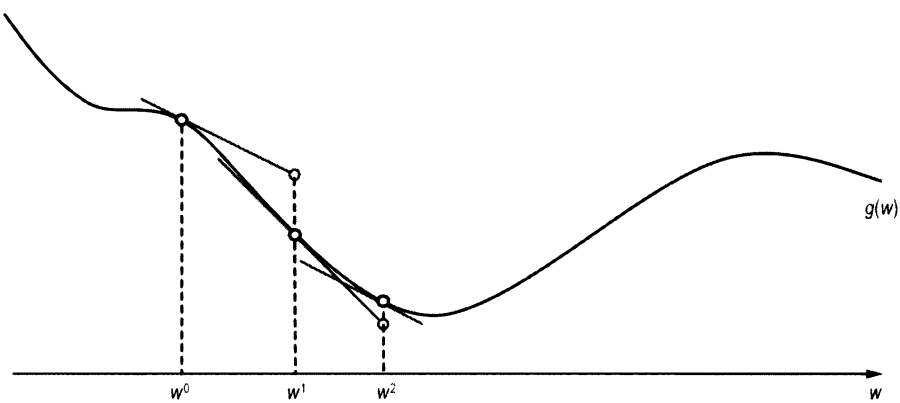


Рис. 3.6. Иллюстрация алгоритма градиентного спуска. Из начальной точки w^0 мы делаем нашу первую аппроксимацию функции $g(w)$ в точке $\left(w^0, g(w^0)\right)$ (показанной в виде полой черной точки на графике), при этом сама аппроксимация с использованием ряда Тейлора первого порядка нарисована красным цветом. Двигаясь в направлении отрицательного градиентного спуска, обеспечиваемого этой аппроксимацией, мы приходим к точке $w^1 = w^0 - \alpha \frac{d}{dw} g\left(w^0\right)$. Затем мы повторяем этот процесс на w^1, двигаясь в направлении отрицательного градиента в сторону к $w^2 = w^1 - \alpha \frac{d}{dw} g\left(w^1\right)$, и т. д.

мы думаем об аппроксимации функции с использованием ряда Тейлора первого порядка и о том, какое направление этой касательной гиперплоскости (т. е. отрицательного градиента функции в этой точке) выбрать в качестве нашего направления спуска для алгоритма.

Как мы увидим в этой и других последующих главах, алгоритм градиентного спуска часто является гораздо более качественным алгоритмом локальной оптимизации, чем подходы нулевого порядка, рассмотренные в предыдущей главе. Действительно, градиентный спуск, наряду с его расширениями, которые подробно описаны в *приложении 1*, является, возможно, самым популярным алгоритмом локальной оптимизации, используемым сегодня в машинном обучении. Это в значительной степени обусловлено тем, что направление убывания, представленное здесь (через градиент), почти всегда легче вычислить (в особенности по мере увеличения размерности входных данных), чем искать направление спуска случайным образом (как это делается с методами нулевого порядка, описанными в *разд. 2.4–2.6*). Другими словами, тот факт, что отрицательный градиент обеспечивает направление убывания функции локально, в сочетании с тем фактом, что градиенты легко вычисляются (в особенности при задействовании автоматического дифференциатора), делает градиентный спуск превосходным методом локальной оптимизации.

Пример 3.7. Минимизирование невыпуклой функции с помощью градиентного спуска

Для того чтобы отыскать глобальный минимум общей невыпуклой функции с помощью градиентного спуска (или любого другого метода локальной оптимизации), может потребоваться выполнить его несколько раз с разными инициализациями и/или схемами длины шага. Продемонстрируем это, используя невыпуклую функцию

$$g(w) = \sin(3w) + 0{,}3w^2, \tag{3.33}$$

проиллюстрированную на рис. 3.7, *сверху*. Та же самая функция была минимизирована в примере 2.4 с помощью случайного поиска. Здесь мы инициализируем два прогона градиентного спуска: один в $w^0 = 4{,}5$ (см. рис. 3.7, *слева сверху*) и другой в $w^0 = -1{,}5$ (см. рис. 3.7, *справа сверху*), используя фиксированную длину шага $\alpha = 0{,}05$ для обоих прогонов. Как видно из результатов, в зависимости от точки инициализации мы можем оказаться рядом с локальным или глобальным минимумом функции (шаги окрашены от зеленого до красного по мере того, как метод движется от начала к концу).

Пример 3.8. Минимизирование выпуклой многовходной функции с использованием градиентного спуска

В этом примере мы выполняем градиентный спуск на выпуклой многовходной квадратичной функции

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2, \tag{3.34}$$

использовавшейся в примере 2.3. Зафиксируем параметр длины шага $\alpha = 0{,}1$ для всех десяти шагов алгоритма. На рис. 3.7, *справа снизу* показан путь, проходимый

этим прогоном градиентного спуска во входном пространстве функции; шаги отмечены точками от зеленого до красного цвета, указывающими на последовательность выполнения метода. Трехмерная поверхность функции на рис. 3.7, *слева снизу* демонстрирует контуры функции на ее входном пространстве.

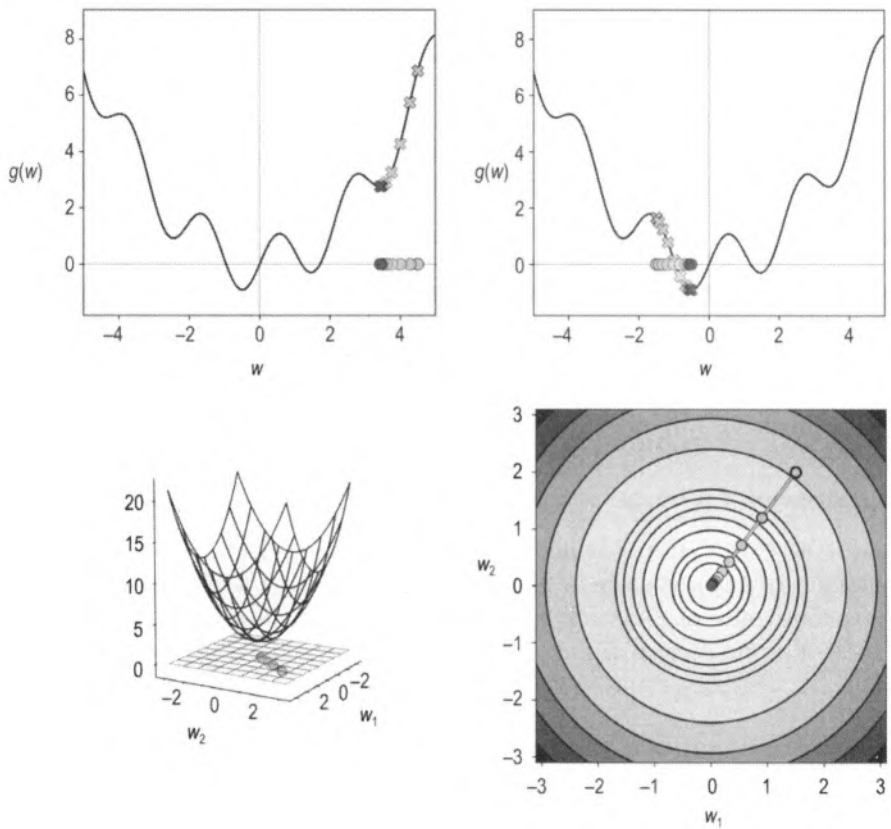


Рис. 3.7. Иллюстрация примера 3.7 (*сверху*) и примера 3.8 (*снизу*)

3.5.1. Базовые варианты длины шага для градиентного спуска

Как и во всех локальных методах, при градиентном спуске необходимо тщательно выбирать параметр длины шага α, или интенсивности усвоения. Хотя существует масса сложных методов выбора α в случае градиентного спуска, наиболее распространенными вариантами, задействуемыми в машинном обучении, являются те базовые подходы, которые вначале подробно описаны в простом контексте методов нулевого порядка в *разд. 2.5*. Эти комбинированные варианты включают, во-первых, использование фиксированного значения α для каждого шага градиентного спуска, которое для простоты обычно принимает вид 10^{γ}, где γ — это (часто отрицательное) целое число, и во-вторых, использование уменьшающейся длины шага, такой как $\alpha = 1/k$, на k-м шаге прогона.

В обоих случаях наша цель в выборе конкретного значения длины шага α на каждом шаге градиентного спуска отражает любой другой метод локальной оптимизации: α должно быть выбрано так, чтобы можно было выполнить наиболее быструю минимизацию. При фиксированной длине шага это часто означает выбор *наибольшего* возможного значения для α, что приводит к надлежащему схождению.

Пример 3.9. Выбор фиксированной длины шага для градиентного спуска

На каждом шаге градиентного спуска у нас всегда есть направление спуска — это явно определяется самим отрицательным градиентом. Однако факт спуска (спускаемся/не спускаемся) полностью зависит от того, как далеко мы двигаемся в направлении отрицательного градиента, который контролируется с помощью нашего параметра длины шага. При неправильно настроенной длине мы можем спускаться в функции бесконечно медленно или даже *подниматься*.

Проиллюстрируем это на рис. 3.8, используя простую одновходную квадратичную функцию $g(w) = w^2$. Здесь приведен результат выполнения пяти шагов градиентного спуска с использованием трех разных фиксированных значений длины шага, инициализированных в одной и той же точке $w^0 = -2{,}5$. В верхнем ряду рис. 3.8 представлены графики самой функции вместе с вычисленными значениями на каждом шаге прогона (значения α, используемые в каждом прогоне, указаны над графиками), которые окрашены от зеленого в начале прогона до красного, когда выполняется последний шаг прогона. От графика к графику фиксированное значение

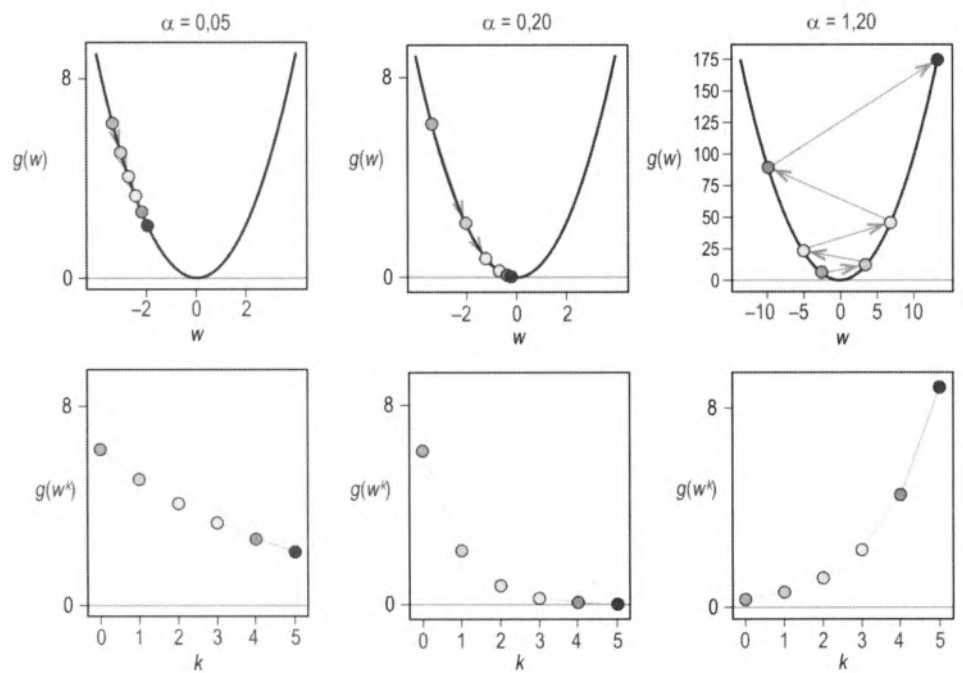


Рис. 3.8. Иллюстрация примера 3.9

длины шага α увеличивается. На первом графике длина шага чрезвычайно мала настолько, что мы вообще не особо спускаемся. На третьем графике, когда установлено слишком большое значение α, алгоритм восходит в функции (в конечном счете *расходясь*).

В нижнем ряду на рис. 3.8 показаны графики истории стоимостной функции, соответствующие каждый своему прогону градиентного спуска из верхнего ряда рисунка. Шаги также окрашены от зеленого (начало прогона) до красного цвета (конец прогона).

Пример 3.10. Сравнение фиксированной и убывающей длины шага

Графики на рис. 3.9 позволяют сравнить варианты с фиксированной длиной шага и убывающей длиной шага для минимизирования функции

$$g(w) = |w| . \tag{3.35}$$

Эта функция имеет один-единственный глобальный минимум в точке $w = 0$. Сделаны два прогона по 20 шагов градиентного спуска, каждый из которых был инициализирован в точке $w^0 = 1,75$: первый с правилом фиксированной длины шага $\alpha = 0,5$ (см. рис. 3.9, *слева сверху*) для каждого шага, а второй с использованием правила убывающей длины шага $\alpha = 1/k$ (см. рис. 3.9, *справа сверху*).

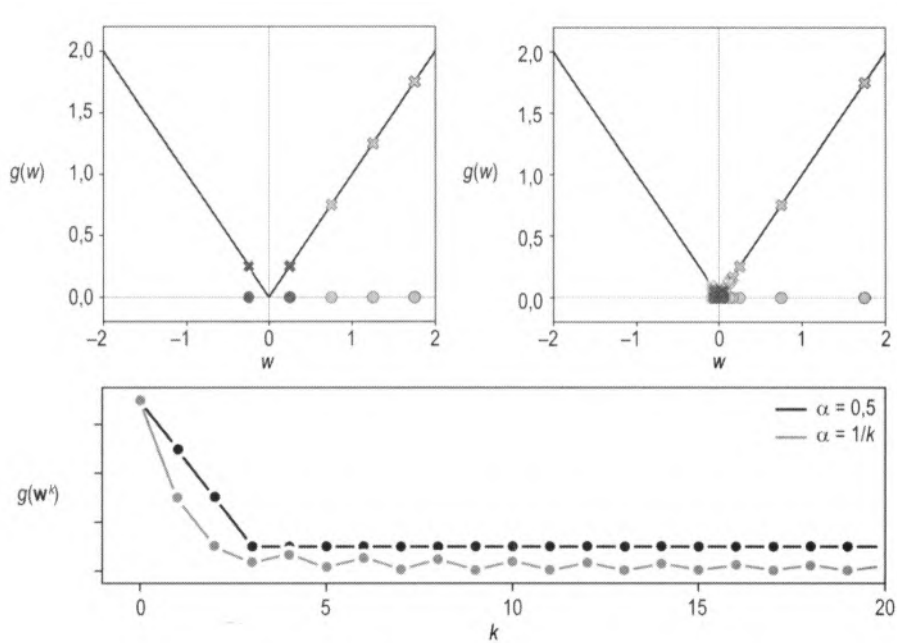


Рис. 3.9. Иллюстрация примера 3.10

Видно, что прогон с фиксированной длиной шага застревает, не давая нам спуститься к минимуму функции, тогда как прогон с убывающей длиной шага хорошо оседает на глобальном минимуме (что также отражено в графике истории стоимо-

стной функции, показанном на рис. 3.9, *снизу*). Застревание обусловлено тем, что производная нашей функции (определяемая везде, кроме точки $w = 0$) принимает вид

$$\frac{d}{dw} g(w) = \begin{cases} +1, & \text{если } w > 0; \\ -1, & \text{если } w < 0, \end{cases} \tag{3.36}$$

что делает использование любой схемы фиксированной длины шага проблематичным, т. к. алгоритм всегда будет двигаться на фиксированное расстояние на каждом шаге. С этой потенциальной проблемой приходится сталкиваться во всех методах локальной оптимизации (впервые мы увидели, как это происходит, в методе нулевого порядка в примере 2.5).

3.5.2. Осцилляция в графике истории стоимостной функции: это не всегда плохо

Помните, что на практике — поскольку мы регулярно имеем дело со стоимостными функциями, которые принимают слишком много входных данных, — мы используем *график истории стоимостной функции* (впервые описанный в *разд. 2.5.2*) для тонкой настройки нашего параметра длины шага α, а также для отладки реализаций алгоритма.

Обратите внимание, что при задействовании графика истории стоимостной функции в выборе правильного значения длины шага в конечном счете не важно, чтобы график, связанный с прогоном градиентного спуска (или любого метода локальной оптимизации), был *строго убывающим* (т. е. показывал, что алгоритм *спускается* на каждом отдельном шаге). Очень важно отыскать значение α, которое позволяет градиентному спуску найти наименьшее возможное значение функции, даже если это означает, что не каждый шаг *спускает* нас к минимуму. Другими словами, наилучший выбранный вариант α для данной минимизации может привести к тому, что градиентный спуск будет немного "прыгать" вокруг, двигаясь вверх и вниз, а не к тому, который показывает спуск в каждом шаге. Рассмотрим пример, иллюстрирующий этот момент.

Пример 3.11. Осцилляторный убывающий график истории стоимостной функции против монотонного

В этом примере мы покажем результат трех прогонов градиентного спуска для минимизирования функции

$$g(\mathbf{w}) = w_1^2 + w_2^2 + 2\sin\left(1{,}5\left(w_1 + w_2\right)\right)^2 + 2, \tag{3.37}$$

чей контурный график представлен на рис. 3.10. Вы видите *локальный минимум* вокруг точки $[1{,}5 \quad 1{,}5]^T$ и глобальный минимум около точки $[-0{,}5 \quad -0{,}5]^T$. Все три прогона начинаются в одной и той же начальной точке $\mathbf{w}^0 = [3 \quad 3]^T$ и занимают десять шагов. Первый прогон (см. рис. 3.10, *слева*) использует фиксированный параметр длины шага $\alpha = 10^{-2}$, второй прогон (см. рис. 3.10, *в центре*) использует $\alpha = 10^{-1}$, а третий прогон (см. рис. 3.10, *справа*) использует $\alpha = 10^{0}$.

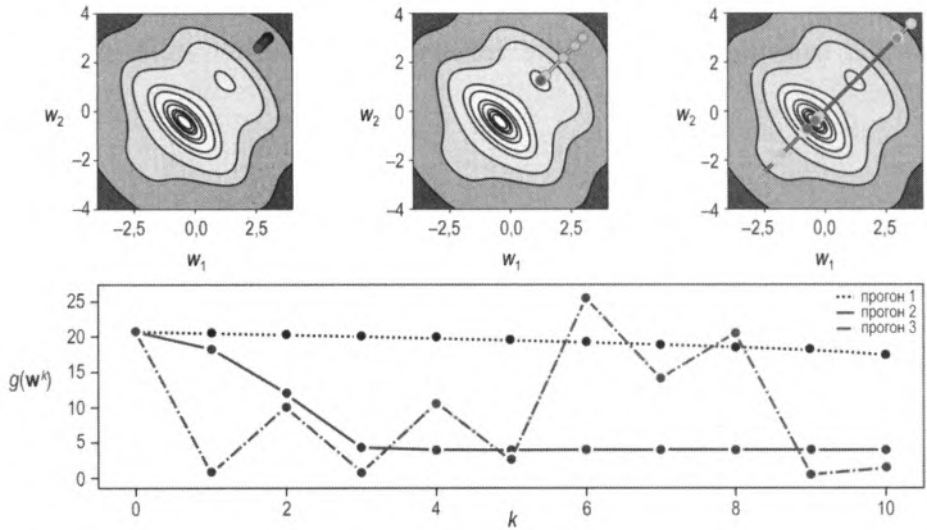


Рис. 3.10. Иллюстрация примера 3.11

На рис. 3.10, *снизу* представлен график истории стоимостной функции, связанный с каждым прогоном градиентного спуска, показывая первый. Видно, что значение нашего первого выбранного варианта было слишком малым (что также подтверждает первый график), второй вариант приводит к схождению к *локальному* минимуму (как видно на втором графике), а финальный прогон, "прыгая вокруг" и не строго уменьшаясь на каждом шаге, находит самую низкую точку из всех трех прогонов. Таким образом, хотя этот прогон использовал самую большую длину шага $\alpha = 10^0$, явно демонстрируя осцилляторное (и, возможно, в конечном итоге расходящееся) поведение, он действительно находит самую низкую точку из всех трех выполненных прогонов.

Рассмотренная функция была "сконструирована" специально для целей данного примера. Однако на практике продемонстрированное здесь поведение, когда история функции потерь, связанная с выполнением градиентного спуска (или любого алгоритма локальной оптимизации в более общем смысле), осциллирует вверх и вниз (а не является совершенно гладкой и монотонно убывающей), соблюдается в целом.

3.5.3. Критерии схождения

Когда градиентный спуск прекращается? Технически, если шаг выбран верно, то алгоритм остановится вблизи стационарных точек функции, обычно минимумов или седловых точек. Откуда нам это известно? По самой форме шагов градиентного спуска. Если шаг $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g(\mathbf{w}^{k-1})$ не приводит к существенному смещению от предыдущей точки $\mathbf{w}^{k-1}$, то это может означать лишь одно: направление, в котором мы движемся, *исчезает*, т. е. $-\nabla g(\mathbf{w}^{k-1}) \approx \mathbf{0}_{N\times 1}$. Это по определению *стационарная точка* функции (как подробно описано в *разд. 3.2*).

В принципе, тогда мы можем подождать до тех пор, пока градиентный спуск достаточно не приблизится к стационарной точке, обеспечивая, например, чтобы магнитуда градиента $\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2$ была достаточно малой. Другие формальные критерии схождения включают, во-первых, остановку, когда шаги больше не достигают достаточной прогрессии (например, когда $\frac{1}{N}\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2$ меньше некоторого ε), или, во-вторых, когда соответствующие вычисляемые значения больше не различаются существенно (например, когда $\frac{1}{N}\left|g(\mathbf{w}^k) - g(\mathbf{w}^{k-1})\right|$ меньше некоторого ε). Наконец, практический способ остановить градиентный спуск — как и любую другую схему локальной оптимизации — просто выполнить алгоритм фиксированное максимальное число итераций. В приложениях машинного обучения это последнее практическое условие очень часто задействуют — либо самостоятельно, либо в сочетании с формальной процедурой остановки.

Как установить максимальное число итераций? Как и в любом локальном методе, оно обычно задается вручную/эвристически и зависит от таких факторов, как вычислительные ресурсы, знание конкретной минимизируемой функции и — что очень важно — выбранного варианта параметра длины шага α. Меньшее значение α хотя и обеспечивает более легкий спуск на каждом шаге, часто требует от алгоритма больше шагов для достижения удовлетворительной прогрессии. И наоборот, если α задан слишком большим, то градиентный спуск может хаотично скакать вечно, никогда не локализуясь в адекватном решении.

3.5.4. Имплементация на Python

В этом разделе мы предлагаем базовую имплементацию алгоритма градиентного спуска на Python. На практике можно использовать ряд вариаций, включая различные условия остановки (описанные выше), а также различные условия вычисления самого градиента. Входами здесь являются минимизируемая функция `g`, длина шага `alpha`, максимальное число итераций `max_its` (наше принятое за основу условие остановки) и начальная точка `w`, которая обычно выбирается случайно. Выходами являются история весовых обновлений и соответствующая история значений стоимостной функции (которую можно использовать для построения графика истории стоимостной функции). Вычисление градиентной функции в строке 16 по умолчанию задействует библиотеку автоматического дифференцирования autograd с открытым исходным кодом (подробно описанную в *разд. 3.4* и *П2.10*), хотя ее можно легко заменить любым другим методом вычисления градиентной функции.

```
1| # импортировать автоматический дифференциатор, чтобы вычислить градиентный модуль
2| from autograd import grad
3|
4| # функция градиентного спуска
5| def gradient_descent(g, alpha, max_its, w):
6|
```

```
    # вычислить градиентный модуль, используя autograd
    gradient = grad(g)

    # цикл градиентного спуска
    weight_history = [w]  # контейнер истории веса
    cost_history = [g(w)] # контейнер истории стоимостной функции
    for k in range(max_its):

        # оценить градиент
        grad_eval = gradient(w)

        # сделать шаг градиентного спуска
        w = w - alpha*grad_eval

        # записать вес и стоимость
        weight_history.append(w)
        cost_history.append(g(w))

    return weight_history, cost_history
```

Учитывая, что вход в `g` является N-мерным, общую случайную инициализацию можно записать, как показано ниже, где функция `random.randn` библиотеки NumPy производит отбор из стандартного нормального распределения с нулевым средним значением и единичным стандартным отклонением. Также нередко такие инициализации шкалируются малыми константами (например, `0.1` или меньше).

```
# общая схема со случайной инициализацией
import numpy as np
scale = 0.1
w = scale*np.random.randn(N,1)
```

3.6. Два естественных недостатка градиентного спуска

Как мы увидели в предыдущем разделе, градиентный спуск является схемой локальной оптимизации, которая задействует отрицательный градиент на каждом шаге. Тот факт, что дифференциальное исчисление дает нам истинное направление спуска в виде отрицательного градиента, в сочетании с тем фактом, что вычисление градиентов часто обходится дешево (независимо от того, используется ли автоматический дифференциатор), означает, что нам не нужно искать разумное направление спуска на каждом шаге метода, как в случае использования методов нулевого порядка, подробно описанных в предыдущей главе. Это чрезвычайно выгодно и является фундаментальной причиной, по которой градиентный спуск так популярен в машинном обучении.

Однако ни одна базовая схема локальной оптимизации не лишена недостатков. В предыдущей главе мы увидели, как, например, естественный недостаток случайного поиска ограничивает его практическое использование функциями лишь низ-

коразмерного входа. Градиентный спуск не страдает от этого конкретного ограничения, однако отрицательный градиент имеет свои слабые стороны, такие как направление спуска, которое мы рассмотрим в этом разделе.

3.6.1. Откуда берутся слабые стороны направления (отрицательного) градиента?

Откуда берутся эти недостатки? Как и любой вектор, отрицательный градиент в своей основе всегда состоит из *направления* и *магнитуды* (рис. 3.11). В зависимости от минимизируемой функции любой из этих атрибутов (или оба) может представлять трудности при использовании отрицательного градиента в качестве направления спуска.

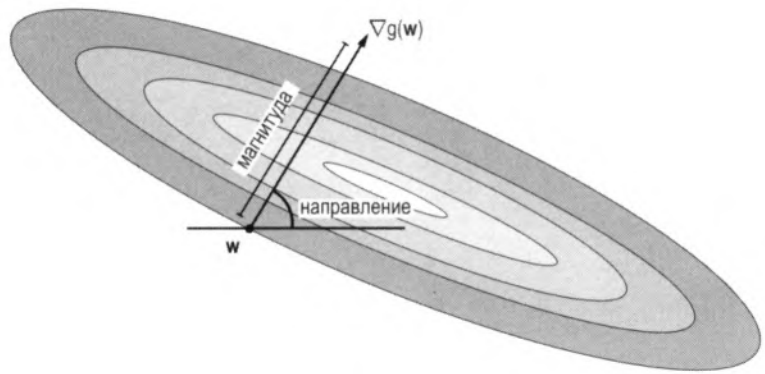


Рис. 3.11. Градиентный вектор любой произвольной функции в любой точке состоит из *магнитуды* и *направления*

Направление отрицательного градиента может быстро осциллировать во время выполнения градиентного спуска, часто производя *зигзагообразные* шаги, которые занимают значительное время на достижение минимальной точки. *Магнитуда* отрицательного градиента может быстро исчезать вблизи стационарных точек, приводя к тому, что градиентный спуск *медленно ползет* вблизи минимумов и седловых точек. Это тоже замедляет прогрессию градиентного спуска вблизи стационарных точек. Указанные две проблемы, хотя и не характерны для минимизирования каждой отдельной функции, все же присутствуют в машинном обучении, потому что многие функции, которые мы стремимся минимизировать, имеют длинные *узкие долины* — длинные плоские области, где контуры функции становятся все более параллельными. Обе эти трудности, вытекающие либо из направления, либо из магнитуды направления отрицательного градиента, рассмотрены далее.

3.6.2. Направление (отрицательного) градиента

Фундаментальным свойством направления (отрицательного) градиента является то, что оно всегда указывает в сторону, перпендикулярную контурам функции. Это утверждение универсально верно и соблюдается для *любой* (дифференцируемой)

функции и для *всех* ее входов, т. е. направление градиентного подъема/спуска во входе $\mathbf{w}^0$ всегда является перпендикулярным контуру $g(\mathbf{w}) = g(\mathbf{w}^0)$.

Пример 3.12. Направление отрицательного градиента

На рис. 3.12 показан контурный график функций:

$$g(\mathbf{w}) = w_1^2 + w_2^2 + 2 \ (\textit{слева сверху});$$

$$g(\mathbf{w}) = w_1^2 + w_2^2 + 2\sin\left(1,5(w_1 + w_2)\right)^2 + 2 \ (\textit{справа сверху});$$

$$g(\mathbf{w}) = \left(w_1^2 + w_2^2 - 11\right)^2 + \left(w_1 + w_2^2 - 6\right)^2 \ (\textit{снизу}).$$

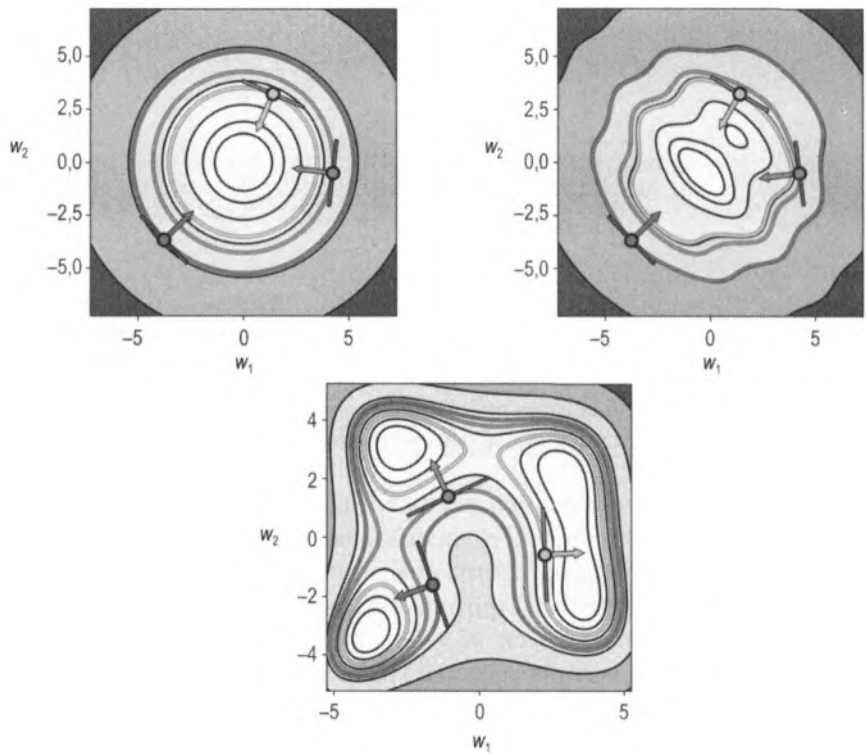


Рис. 3.12. Иллюстрация примера 3.12. Независимо от функции направление отрицательного градиента всегда является *перпендикулярным* контурам функции

На каждом графике показано направление отрицательного градиента, определенное в трех случайных точках. Каждая из выбранных нами точек окрашена в уникальный цвет, причем для целей визуализации контур, на котором они размещаются, окрашен в тот же цвет. Направление *спуска*, определяемое градиентом в каждой точке, нарисовано в виде стрелки, проведены также касательные к контуру на каждом входе (соответствующим цветом).

Видно, что направление градиентного спуска всегда *перпендикулярно* контуру, на котором оно лежит, — в частности, перпендикулярно касательной в каждой контурной точке. Поскольку направления градиентного подъема будут просто указывать в противоположном направлении, они также будут перпендикулярны контурам.

3.6.3. Зигзагообразное поведение градиентного спуска

На практике тот факт, что отрицательный градиент всегда указывает в направлении, перпендикулярном контуру функции, может — в зависимости от минимизируемой функции — заставить направление отрицательного градиента быстро осциллировать или вести себя *зигзагообразно* во время выполнения градиентного спуска. Это может вызвать зигзагообразное поведение самих шагов градиентного спуска, а *слишком большое* зигзагообразное движение замедляет прогрессию минимизации. Когда это происходит, требуется много шагов градиентного спуска на то, чтобы адекватно минимизировать функцию. Проиллюстрируем это явление, рассмотрев ряд простых примеров.

Для заинтересованного читателя описание популярного решения этого зигзагообразного поведения, именуемого *импульсно-ускоренным* градиентным спуском, приведено в *разд. П1.2*.

Пример 3.13. Зигзагообразное поведение градиентного спуска

На рис. 3.13 показано зигзагообразное поведение градиентного спуска с тремя двумерными квадратичными функциями, которые принимают общий вид $g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}$. В каждом случае a и $\mathbf{b}$ равны нулю, а матрица $\mathbf{C}$ задается так, что каждая квадратичная функция становится все более узкой:

- на рис. 3.13, *сверху*:

$$\mathbf{C} = \begin{bmatrix} 0{,}50 & 0 \\ 0 & 12 \end{bmatrix}; \tag{3.38а}$$

- на рис. 3.13, *в центре*:

$$\mathbf{C} = \begin{bmatrix} 0{,}10 & 0 \\ 0 & 12 \end{bmatrix}; \tag{3.38б}$$

- на рис. 3.13, *снизу*:

$$\mathbf{C} = \begin{bmatrix} 0{,}01 & 0 \\ 0 & 12 \end{bmatrix}, \tag{3.38в}$$

и отсюда квадратичные функции отличаются только тем, как мы устанавливаем левое верхнее значение матрицы $\mathbf{C}$. Все три квадратичные функции, контурные графики которых показаны соответственно на рис. 3.13, имеют один и тот же глобальный минимум в начале координат. Однако, когда мы меняем это единственное

значение в матрице **C** от квадратичной функции к квадратичной функции, мы значительно удлиняем контуры вдоль горизонтальной оси, так что в третьем случае контуры кажутся почти полностью параллельными друг другу вблизи нашей стартовой точки (пример *длинной узкой долины*).

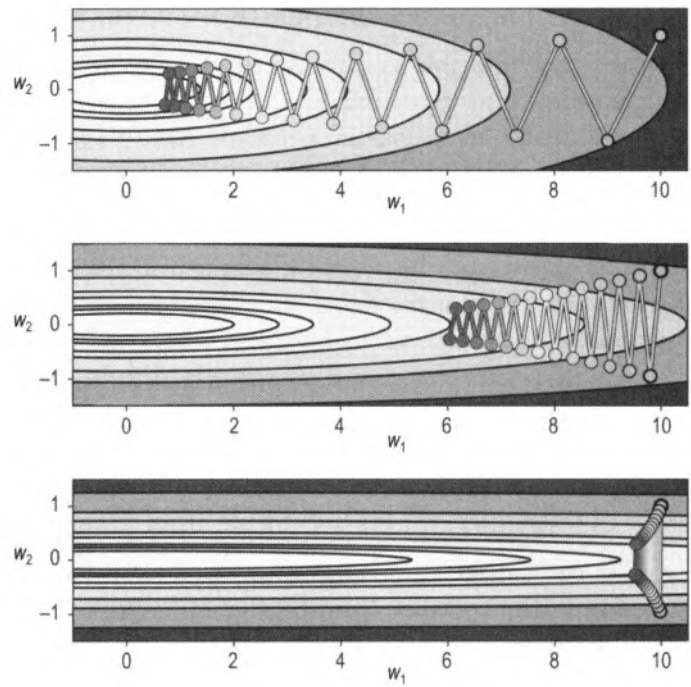


Рис. 3.13. Иллюстрация примера 3.13
с зигзагообразным поведением градиентного спуска

В этом примере мы делаем прогон из 25 шагов градиентного спуска, чтобы минимизировать каждую функцию, используя одинаковую инициализацию в точке $\mathbf{w}^0 = [10 \quad 1]^T$ и значение длины шага $\alpha = 10^{-1}$. В каждом случае веса, найденные на каждом шаге, отмечены точками на контурных графиках и окрашены от зеленого (в начале прогона) до красного цвета (по мере достижения максимального числа итераций). Зигзагообразное поведение градиентного спуска усиливается от первого к третьему графику. С третьей квадратичной функцией вообще не наблюдается большой прогрессии из-за большого числа зигзагов.

Понятна и причина этого зигзагообразного поведения: направление отрицательного градиента постоянно указывает в перпендикулярном направлении относительно контуров функции, и в очень узких функциях эти контуры становятся почти параллельными. Мы, конечно, можем ослабить это зигзагообразное поведение, уменьшив значение длины шага, однако это не решает основную проблему, которую создает зигзагообразное поведение, а именно медленное схождение.

3.6.4. Медленно ползающее поведение градиентного спуска

Как мы знаем из условия оптимальности первого порядка, рассмотренного в *разд. 3.2*, (отрицательный) градиент исчезает в стационарных точках, т. е. если $\mathbf{w}$ — это минимальная, максимальная или седловая точка, то мы знаем, что $\nabla g(\mathbf{w}) = 0$. Это также означает, что магнитуда градиента обращается в ноль в стационарных точках, т. е. $\|\nabla g(\mathbf{w})\|_2 = 0$. Другими словами, (отрицательный) градиент в точках вблизи стационарной точки имеет ненулевое направление, но исчезающую магнитуду, т. е. $\|\nabla g(\mathbf{w})\|_2 \approx 0$.

Исчезающее поведение магнитуды отрицательного градиента вблизи стационарных точек имеет естественное следствие для шагов градиентного спуска: они продвигаются очень медленно, или "ползут", вблизи стационарных точек. Это происходит потому, что, *в отличие* от методов нулевого порядка, рассмотренных в *разд. 2.5* и *2.6* (где мы нормализовали магнитуду каждого направления спуска), расстояние, пройденное на каждом шаге градиентного спуска, не полностью определяется значением длины шага α. Действительно, мы можем легко вычислить общее расстояние, пройденное шагом градиентного спуска, как

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|\left(\mathbf{w}^{k-1} - \alpha\nabla g(\mathbf{w}^{k-1})\right) - \mathbf{w}^{k-1}\right\|_2 = \alpha\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2. \tag{3.39}$$

Другими словами, длина общего шага градиентного спуска равна значению параметра длины шага α, умноженному на магнитуду направления спуска.

Последствия этого довольно легко разгадать. Так как магнитуда градиента $\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2$ велика вдали от стационарных точек, и поскольку мы часто случайно инициализируем градиентный спуск на практике так, что наши начальные точки часто лежат очень далеко от любой стационарной точки функции, первые несколько шагов прогона градиентного спуска в целом будут большими и сделают значительную прогрессию в направлении минимизации. И наоборот, при приближении к стационарной точке магнитуда градиента уменьшается, и поэтому длина, проходимая шагом градиентного спуска, также мала. Это означает, что шаги градиентного спуска слабо продвигаются к минимуму вблизи стационарной точки.

Одним словом, тот факт, что длина каждого шага градиентного спуска является пропорциональной магнитуде градиента, означает, что градиентный спуск часто начинается со значительной прогрессии, но существенно замедляется вблизи минимумов и седловых точек — такое поведение мы называем *медленным ползанием*. Для отдельных функций такое поведение медленного ползания может не только означать, что для достижения адекватной минимизации требуется много шагов, но и способно приводить к полной остановке градиентного спуска вблизи седловых точек невыпуклых функций.

Заинтересованный читатель может обратить внимание на то, что описание популярного решения этого медленно ползущего поведения, именуемого *нормализованным градиентным спуском*, приведено в *разд. П1.3* и *П1.4*.

Пример 3.14. Медленно ползущее поведение градиентного спуска

На рис. 3.14, *слева* построен график функции

$$g(w) = w^4 + 0{,}1\,, \tag{3.40}$$

чей минимум находится в начале координат. Ее мы будем минимизировать, используя десять шагов градиентного спуска и параметр длины шага $\alpha = 10^{-1}$. Результаты этого прогона отражены на графике функции (шаги окрашены от зеленого в начале прогона до красного цвета на последнем шаге). Видно, что этот прогон градиентного спуска начинается с больших шагов, но медленно ползет, приближаясь к минимуму. Оба поведения являются вполне естественными, т. к. магнитуда градиента велика вдали от глобального минимума и исчезает вблизи него.

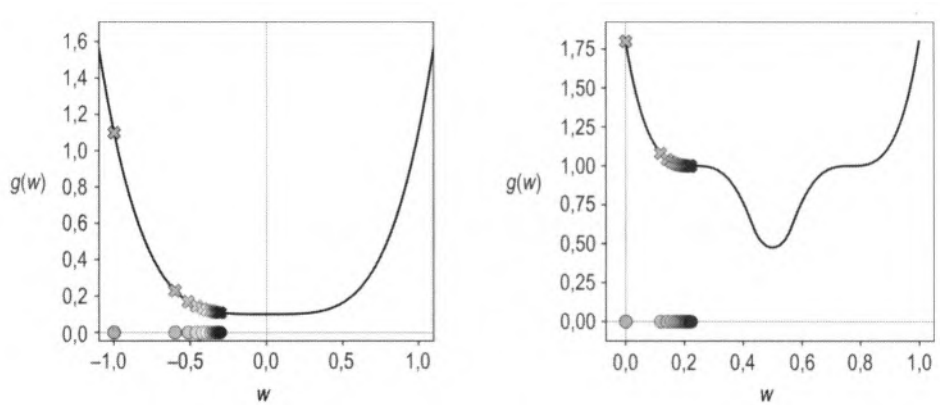


Рис. 3.14. Иллюстрация примера 3.14

На рис. 3.14, *справа* продемонстрирована проблема ползания градиентного спуска вблизи седловых точек с помощью невыпуклой функции

$$g(w) = \max^2\left(0{,}1 + \left(3w - 2{,}3\right)^3\right) + \max^2\left(0{,}1 + \left(-3w + 0{,}7\right)^3\right). \tag{3.41}$$

Эта функция имеет минимум в $w = 1/2$ и седловые точки в $w = 7/30$ и $w = 23/30$. Выполнен прогон градиентного спуска на этой функции с использованием 50 шагов с $\alpha = 10^{-2}$, инициализированных в начале координат.

Видно, что шаги градиентного спуска прекращаются около самой левой седловой точки из-за настроек (инициализация и параметр длины шага), выбранных для этого прогона. Тот факт, что градиентный спуск ползет по мере приближения к этой седловой точке, является вполне естественным (поскольку магнитуда градиента здесь исчезает), но это мешает алгоритму отыскивать глобальный минимум.

3.7. Заключение

В этой главе мы описали схемы локальной оптимизации, которые используют первую производную функции для получения эффективных направлений спуска — иначе именуемых *методами первого порядка*. Такие методы составляют, пожалуй,

самый популярный набор оптимизационных инструментов, используемых в задачах машинного обучения.

Мы начали в *разд. 3.2* с рассмотрения того, как первые производные функции обеспечивают полезное условие для характеризации ее минимумов, максимумов и седловых точек (называемых стационарными точками) посредством условия оптимальности первого порядка. При подготовке к нашему обсуждению локальных методов первого порядка мы описали *восходящее* и *нисходящее направления* гиперплоскости, а также те, которые обеспечиваются касательной гиперплоскостью, связанной с аппроксимацией с использованием ряда Тейлора первого порядка в *разд. 3.3*. В *разд. 3.5* мы увидели, как такие направления спуска, задействуемые в локально-оптимизационном каркасе, естественно приводят к конструированию популярной локальной схемы, именуемой *градиентным спуском*.

Алгоритм градиентного спуска широко применяется в машинном обучении, поскольку направление спуска, обеспечиваемое отрицательным градиентом, почти всегда легко доступно для использования (и поэтому направление спуска не нужно искать явно, как в методах нулевого порядка, описанных в *главе 2*). Однако по самой своей природе направление отрицательного градиента имеет два неотъемлемых недостатка для локальной оптимизации — проблемы *зигзагообразного* и *медленного ползающего* поведения, подробно описанные в *разд. 3.6*, которые снижают его эффективность. Эти проблемы и соответствующие им решения (именуемые продвинутыми методами оптимизации первого порядка) подробно рассмотрены в *приложении 1*.

3.8. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:
github.com/jermwatt/machine_learning_refined.

3.1. Условие оптимальности первого порядка

Использовать условие первого порядка, чтобы найти все стационарные точки функции g (расчеты должны выполняться вручную). Затем построить график функции g, обозначить найденную точку (точки) меткой и определить "на глаз", какой является каждая стационарная точка: минимальной, максимальной или седловой. Примечание: стационарные точки могут находиться в бесконечности!

А) $g(w) = w\log(w) - (1-w)\log(1-w)$, где w лежит между 0 и 1.

Б) $g(w) = \log(1 + e^{w})$.

В) $g(w) = w\tanh(w)$.

Г) $g(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{C}\mathbf{w} + \mathbf{b}^T\mathbf{w}$, где $\mathbf{C} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$ и $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

3.2. Стационарные точки простой квадратичной функции

В ряде приложений мы окажемся в ситуации, когда будем задействовать простую многовходную квадратичную функцию

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}, \tag{3.42}$$

где матрица $\mathbf{C} = \frac{1}{\beta}\mathbf{I}$. Здесь $\mathbf{I}$ — это $N \times N$-матрица тождественности; $\beta > 0$ — положительный скаляр. Найти все стационарные точки функции g.

3.3. Стационарные точки отношения Рэлея

Рэлеево отношение $N \times N$-матрицы $\mathbf{C}$ определяется как нормализованная квадратичная функция

$$g(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{C}\mathbf{w}}{\mathbf{w}^T\mathbf{w}}, \tag{3.43}$$

где $\mathbf{w} \neq \mathbf{0}_{N\times 1}$. Вычислить стационарные точки этой функции.

3.4. Координатный спуск первого порядка как схема локальной оптимизации

А) Выразить метод координатного спуска, описанный в *разд. 3.2.2*, в виде схемы локальной оптимизации, т. е. в виде последовательности шагов вида $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\mathbf{d}^k$.

Б) Закодировать метод координатного спуска для частного случая квадратичной функции и повторить эксперимент, описанный в примере 3.5.

3.5. Опробовать градиентный спуск

Выполнить градиентный спуск для минимизирования функции

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right) \tag{3.44}$$

с начальной точкой $w^0 = 2$ и 1000 итераций. Сделать три отдельных прогона, используя следующие значения длины шага: $\alpha = 1$, $\alpha = 10^{-1}$ и $\alpha = 10^{-2}$. Вычислить производную этой функции вручную и имплементировать ее (а также саму функцию) на языке Python с помощью библиотеки NumPy.

Построить график результирующей истории стоимостной функции каждого прогона на одном рисунке, чтобы сравнить их результативность. Какое значение длины шага лучше всего подходит для этой отдельной функции и начальной точки?

3.6. Сравнить фиксированное и убывающее значения длины шага

Повторить сравнительный эксперимент, описанный в примере 3.10, произведя сравнение кривых на графике истории стоимостной функции, показанном на рис. 3.9, *снизу*.

3.7. Осцилляция на графике истории стоимостной функции

Повторить эксперимент, описанный в примере 3.11, построив график стоимостной функции, показанный на рис. 3.10, *снизу*.

3.8. Настроить фиксированную длину шага для градиентного спуска

Возьмем стоимостную функцию

$$g(\mathbf{w}) = \mathbf{w}^T\mathbf{w}, \tag{3.45}$$

где $\mathbf{w}$ — 10-мерный входной вектор; g — выпуклая функция с одним-единственным глобальным минимумом в $\mathbf{w} = \mathbf{0}_{N\times 1}$. Закодировать градиентный спуск и выполнить его за 100 шагов, используя начальную точку $\mathbf{w}^0 = 10\cdot\mathbf{1}_{N\times 1}$ с тремя значениями длины шага: $\alpha_1 = 0{,}001$, $\alpha_2 = 0{,}1$ и $\alpha_3 = 1$. Построить график истории стоимостной функции, чтобы сравнить три прогона и определить, какой из них работает лучше всего.

3.9. Закодировать импульсно-ускоренный градиентный спуск

Закодировать схему импульсно-ускоренного градиентного спуска, описанную в *разд. П1.2.2*, и задействовать ее для повторения экспериментов, описанных в примере П1.1, используя график истории стоимостной функции, чтобы прийти к тем же самым выводам, сделанным при изучении контурных графиков, показанных на рис. П1.3.

3.10. Медленно ползающее поведение градиентного спуска

В этом упражнении вы будете сравнивать схемы стандартного и полно-нормализованного градиентного спуска в минимизировании функции

$$g(w_1, w_2) = \tanh(4w_1 + 4w_2) + \max(1; 0, 4w_1^2) + 1. \tag{3.46}$$

Используя инициализацию $\mathbf{w}^0 = [2 \quad 2]^T$, выполнить прогон 1000 шагов схем стандартного и полно-нормализованного градиентного спуска, используя значение длины шага $\alpha = 10^{-1}$ в обоих случаях. Построить график истории стоимостной функции для сравнения двух прогонов, отмечая прогрессию, достигаемую с каждым прогоном.

3.11. Сравнение схем нормализованного градиентного спуска

Закодировать схемы полно- и покомпонентно нормализованного градиентного спуска и повторить эксперимент, описанный в примере П1.4, используя график истории стоимостной функции, чтобы прийти к тем же выводам, сделанным при изучении графиков, показанных на рис. П1.6.

3.12. Альтернативное формальное определение липшицева градиента

Альтернатива определению постоянной Липшица уравнением (П1.49) для функций g с липшицевым непрерывным градиентом дается следующим образом:

$$\|\nabla g(\mathbf{x}) - \nabla g(\mathbf{y})\|_2 \le L\|\mathbf{x} - \mathbf{y}\|_2. \tag{3.47}$$

Используя предельное определение производной (*см. разд. П2.2.1*), показать, что это определение является эквивалентным определению, приведенному в уравнении (П1.49).

3.13. Композиция функций с липшицевым градиентом

Предположим, что f и g — две функции с липшицевым градиентом с константами L и K соответственно. Используя определение липшицева непрерывного градиента, приведенное в упражнении 3.12, показать, что композиция $f(g)$ также имеет липшицев непрерывный градиент. Какова соответствующая константа Липшица этой композиции?

4 Приемы оптимизации второго порядка

В этой главе мы опишем фундаментальные алгоритмы оптимизации, использующие первую и вторую производные, а также *градиент* и *гессиану* функции. Эти методы, именуемые *методами оптимизации второго порядка*, сегодня популярны в конкретных приложениях машинного обучения. По аналогии с предыдущей главой здесь мы начнем с обсуждения условия оптимальности второго порядка. Затем обсудим квадратичные функции, а также понятие кривизны, определяемое вторыми производными, и разложение рядов Тейлора второго порядка. Используя первые и вторые производные функции, мы можем конструировать мощные методы локальной оптимизации, включая популярный *метод Ньютона* и его расширения (обычно именуемые *бесгессиановыми* оптимизаторами).

4.1. Условие оптимальности второго порядка

При обсуждении выпуклости/вогнутости общих математических функций мы часто говорим о выпуклости/вогнутости *в точке*. Для того чтобы определить, является ли общая одновходная функция $g(w)$ *выпуклой* или *вогнутой* в точке v, мы проверяем ее кривизну или информацию о второй производной в этой точке (принимая, что она там по крайней мере дважды дифференцируема): если $\frac{d^2}{dw^2}g(v) \geq 0$ (или ≤ 0), то принято говорить, что g является выпуклой (или вогнутой) в v.

Аналогичное утверждение можно сделать для функции g с многомерным входом: если матрица Гессе (Hessian matrix), вычисленная в точке $\mathbf{v}$, обозначаемая через $\nabla^2 g(\mathbf{v})$, имеет все неотрицательные (или неположительные) *собственные значения*, то g называется выпуклой (или вогнутой) в точке $\mathbf{v}$, и в этом случае сама матрица Гессе называется положительным (или отрицательным) полудефинитом, или *полуопределенной матрицей.*

Основываясь на этих точечных определениях выпуклости/вогнутости, принято говорить, что функция $g(\mathbf{w})$ является выпуклой *везде*, если ее вторая производная $\frac{d^2}{dw^2}g(w)$ является всегда неотрицательной. Аналогично $g(\mathbf{w})$ является выпуклой *везде*, если $\nabla^2 g(\mathbf{w})$ всегда имеет неотрицательные собственные значения. Эта формулировка обычно именуется *определением выпуклости второго порядка.*

Пример 4.1. Выпуклость одновходных функций

В этом примере мы используем определение выпуклости второго порядка, чтобы проверить, является ли каждая из функций, показанных на рис. 4.1, выпуклой или нет.

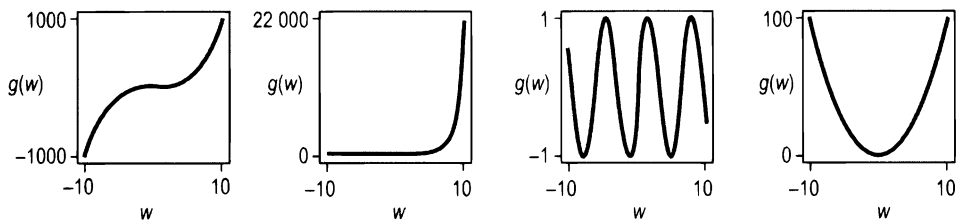


Рис. 4.1. Иллюстрация примера 4.1.
Слева направо графики функций $g(w) = w^3$, $g(w) = e^w$, $g(w) = \sin w$ и $g(w) = w^2$

- $g(w) = w^3$ имеет вторую производную $\frac{d^2}{dw^2} g(w) = 6w$, которая не всегда является неотрицательной; следовательно, g не является выпуклой.
- $g(w) = e^w$ имеет вторую производную $\frac{d^2}{dw^2} g(w) = e^w$, которая является положительной при любом выборе w, и, следовательно, g является выпуклой.
- $g(w) = \sin w$ имеет вторую производную $\frac{d^2}{dw^2} g(w) = -\sin w$. Поскольку она не всегда является неотрицательной, g не является выпуклой.
- $g(w) = w^2$ имеет вторую производную $\frac{d^2}{dw^2} g(w) = 2$, которая всегда положительна, и поэтому g является выпуклой.

Пример 4.2. Выпуклость многовходных квадратичных функций

Многовходная квадратичная функция

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w} \tag{4.1}$$

имеет матрицу Гессе $\nabla^2 g(\mathbf{w}) = 2\mathbf{C}$ (исходя из того, что $\mathbf{C}$ является симметричной). Поэтому выпуклость функции определяется изучением собственных значений матрицы $\mathbf{C}$.

Изучив несколько простых примеров, легко прийти к некоторым далеко идущим выводам о том, как вторая производная помогает раскрывать идентичность статических точек. На рис. 4.2 построены графики трех одновходных функций, которые мы рассматривали в примере 3.1 (определенных в уравнении (3.4)), а также их первые и вторые производные (показанные соответственно в верхнем, среднем и ниж-

нем рядах рисунка). В верхнем ряду вычисленные значения всех стационарных точек функцией отмечены зеленым цветом (а также проведены касательные в этих точках тоже зеленым цветом). Соответствующие значения первой и второй производных также отмечены зеленым цветом соответственно на среднем и нижнем рядах рисунка.

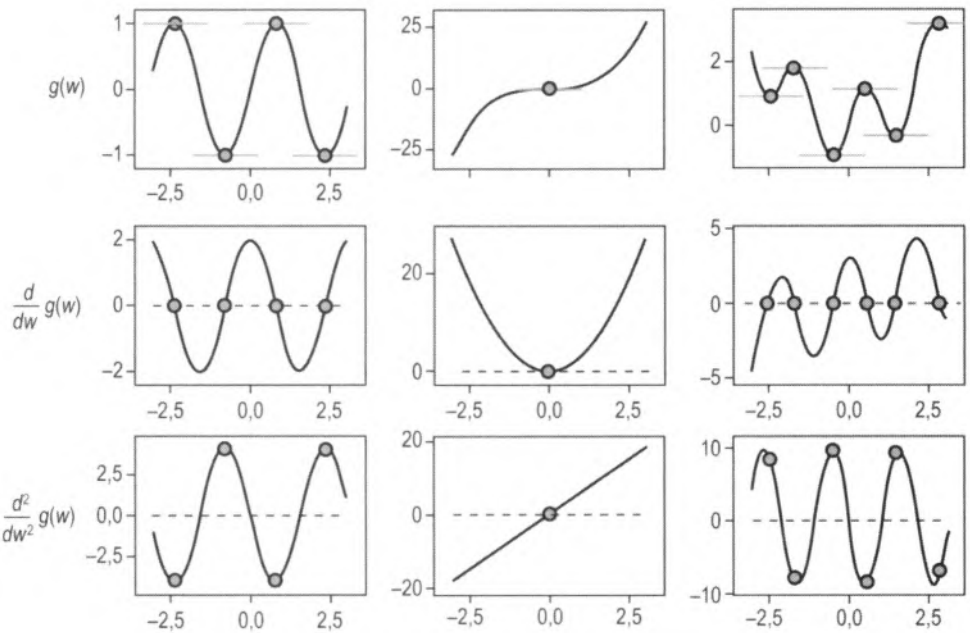


Рис. 4.2. Три одновходные функции и их первые и вторые производные показаны соответственно в верхнем, среднем и нижнем рядах

Изучая эти простые примеры на рис. 4.2, мы видим последовательное поведение некоторых стационарных точек. В частности, видно, как значение второй производной функции в стационарной точке v помогает нам определить, является ли эта точка локальным минимумом, локальным максимумом или седловой точкой. Стационарная точка v является:

- локальным (или глобальным) минимумом, если $\frac{d^2}{dw^2} g(v) > 0$ (т. к. он возникает на *выпуклых* участках функции);
- локальным (или глобальным) максимумом, если $\frac{d^2}{dw^2} g(v) < 0$ (т. к. он возникает на *вогнутых* участках функции);
- седловой точкой, если $\frac{d^2}{dw^2} g(v) = 0$ и $\frac{d^2}{dw^2} g(w)$ меняет знак в $w = v$ (т. к. это происходит в *точке перегиба* функции, т. е. там, где функция переходит из вогнутой в выпуклую или наоборот).

Эти характеристики второго порядка соблюдаются для любой одновходной функции и взятые вместе образуют *условие оптимальности второго порядка* для одновходных функций.

С многовходными функциями соблюдается аналогичное одновходное условие. Как и все, что связано с выпуклостью/вогнутостью и матрицей вторых производных (т. е. матрицей Гессе), условие оптимальности второго порядка для многовходных функций транслируется в *собственные значения* гессианы. Выражаясь конкретнее, стационарная точка **v** многовходной функции $g(\mathbf{w})$ является:

- локальным (или глобальным) минимумом, если все собственные значения матрицы $\nabla^2 g(v)$ являются положительными (т. к. это происходит на *выпуклых* участках функции);
- локальным (или глобальным) максимумом, если все собственные значения матрицы $\nabla^2 g(v)$ являются отрицательными (т. к. это происходит на *вогнутых* участках функции);
- седловой точкой, если собственные значения матрицы $\nabla^2 g(v)$ имеют смешанные значения, т. е. некоторые являются отрицательными, а некоторые — положительными (т. к. это происходит в *точке перегиба* функции).

Обратите внимание: когда входная размерность N равна 1, эти правила сводятся к тем, которые изложены для одновходных функций, поскольку матрица Гессе коллапсирует в единственную вторую производную.

4.2. Геометрия ряда Тейлора второго порядка

Как мы увидим в этой главе, квадратичные функции естественным образом возникают при изучении методов оптимизации. В этом разделе мы сначала обсудим квадратичные функции, сконцентировавшись на определили их общего контура и выяснении, являются ли они *выпуклыми*, *вогнутыми* или имеют более сложную геометрию. Затем мы изучим квадратичные функции, генерируемые аппроксимацией с использованием ряда Тейлора второго порядка (*см. разд. П2.9* с обзором этой концепции), и, в частности, как эти фундаментальные квадратичные функции по своей сути описывают локальную кривизну дважды дифференцируемой функции.

4.2.1. Общий контур одновходных квадратичных функций

Базовая формула для квадратичной одновходной функции принимает знакомую форму

$$g(w) = a + bw + cw^2, \tag{4.2}$$

где a, b и c — константные значения, управляющие контуром функции. В частности, константа c управляет *выпуклостью* или *вогнутостью* функции или, другими словами, тем, обращена ли квадратичная грань вверх или вниз. Когда значение c

является *неотрицательным*, квадратичная функция *выпуклая* и *направлена вверх* независимо от того, как заданы другие параметры. И наоборот, когда значение c является *неположительным*, квадратичная кривая *вогнута* и *направлена вниз*. При $c = 0$ квадратичная функция сводится к линейной функции (которую можно считать как выпуклой, так и вогнутой).

Например, на рис. 4.3, *слева* построены графики двух простых квадратичных функций: выпуклой квадратичной функции $g(w) = 6w^2$ (*слева сверху*) и вогнутой квадратичной функции $g(w) = -w^2$ (*слева снизу*), чтобы проиллюстрировать то, как значение c управляет контуром и сложностью общей квадратичной функции.

4.2.2. Общий контур многовходных квадратичных функций

Многовходная квадратичная функция принимает форму, полностью обобщенную из одновходного случая, который мы записываем как

$$g(\mathbf{w}) = a + b^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}, \tag{4.3}$$

где вход $\mathbf{w}$ является N-мерным, a остается константой; $\mathbf{b}$ — $N \times 1$-вектор; $\mathbf{C}$ — $N \times N$-матрица (которую для наших целей мы принимаем *симметричной*). Поскольку эта квадратичная функция определена вдоль многочисленных размерностей, она может принимать более разнообразные контуры, чем ее одновходной аналог. Например, она может быть выпуклой вдоль некоторых входных размерностей и вогнутой вдоль других.

Обобщение одновходного теста на выпуклость/вогнутость заключается уже не в том, являются ли *значения* матрицы $\mathbf{C}$ положительными или отрицательными, а в том, являются ли ее *собственные значения* таковыми (*см. разд. П3.4.3*). Если все собственные значения матрицы *неотрицательны*, то квадратичная функция является *выпуклой*, если же все *неположительны* — *вогнутой*, а если все равны нулю, то она сводится к линейной функции, которая одновременно является и выпуклой, и вогнутой, и в противном случае (т. е. если некоторые из ее собственных значений положительны, а другие отрицательны) она не является ни выпуклой, ни вогнутой.

В рис. 4.3, *в центре* и *справа* представлены несколько примеров многовходных квадратичных функций с $N = 2$ входами. Во всех примерах мы установили a и $\mathbf{b}$ равными нулю и просто изменили значения матрицы $\mathbf{C}$. Для простоты во всех четырех случаях матрица $\mathbf{C}$ выбрана диагональной с тем, чтобы ее собственные значения удобно располагались по ее диагонали:

- рис. 4.3 (*в центре сверху*):

$$\mathbf{C} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix};$$

- рис. 4.3 (*справа сверху*):

$$\mathbf{C} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix};$$

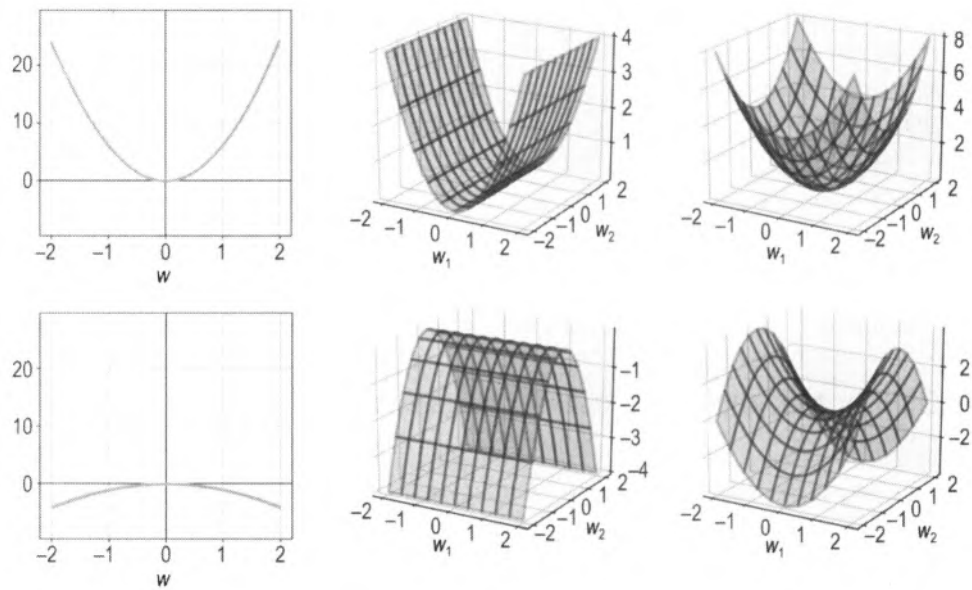


Рис. 4.3. Выпуклая одновходная квадратичная функция (*слева сверху*). Вогнутая одновходная квадратичная функция (*слева снизу*). Две выпуклые двухвходные квадратичные функции (*в центре и справа сверху*). Вогнутая двухвходная квадратичная функция (*в центе снизу*). Двухвходная квадратичная функция, которая не является ни выпуклой, ни вогнутой (*справа снизу*)

- рис. 4.3 (*в центе внизу*):

$$\mathbf{C} = \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix};$$

- рис. 4.3 (*справа внизу*):

$$\mathbf{C} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

4.2.3. Локальная кривизна и ряд Тейлора второго порядка

Делать вывод о локальной выпуклости или вогнутости функции g в точке v можно с помощью еще одного способа — посредством ее аппроксимации с использованием ряда Тейлора второго порядка в этой точке (*см. разд. П2.9*). Эта фундаментальная аппроксимация, принимающая форму

$$h(w) = g(v) = \left(\frac{d}{dw} g(v)\right)(w - v) + \frac{1}{2}\left(\frac{d^2}{dw^2} g(v)\right)(w - v)^2, \tag{4.4}$$

является истинной квадратичной функцией, построенной с использованием первой и второй производной функции. Мало того, что аппроксимация второго порядка совпадает с кривизной неотрицательной функции в каждой точке v в области определения функции, но если функция является выпуклой в этой точке (из-за того, что ее вторая производная неотрицательна), то ряд Тейлора второго порядка является

выпуклым везде. Аналогично, если функция является вогнутой в точке v, то эта аппроксимирующая квадратичная функция является *вогнутой везде*.

Эта концепция соблюдается и для многовходных функций. Аппроксимация с использованием ряда Тейлора второго порядка функции, принимающей N-мерный вход в точке v, задается следующим образом:

$$h(\mathbf{w}) = g(\mathbf{v}) + \nabla g(\mathbf{v})^T (\mathbf{w} - \mathbf{v}) + \frac{1}{2}(\mathbf{w} - \mathbf{v})^T \nabla^2 g(\mathbf{v})(\mathbf{w} - \mathbf{v}). \quad (4.5)$$

Опять же, когда функция g является выпуклой в точке $\mathbf{v}$, соответствующая квадратичная функция выпукла везде. Схожие утверждения можно сделать, когда функция g является вогнутой в точке $\mathbf{v}$, или ни выпуклой, ни вогнутой в некторой точке.

Пример 4.3. Локальная выпуклость/вогнутость и ряд Тейлора второго порядка

На рис. 4.4 показан график функции

$$g(w) = \sin(3w) + 0{,}1w^2, \quad (4.6)$$

построенный черным цветом вместе с квадратичной аппроксимацией с использованием ряда Тейлора второго порядка, показанной синим цветом штрихпунктирной линией в трех точках примера. Видно, что локальная выпуклость/вогнутость функции совершенно отражается в контуре ассоциированной квадратичной аппроксимации, т. е. в точках локальной выпуклости (как на рисунках *слева* и *в центре*) соответствующая квадратичная аппроксимация везде является выпуклой. И наоборот, в точках локальной вогнутости (как и на рис. 4.4, *справа*) соответствующая квадратичная аппроксимация является повсеместно вогнутой.

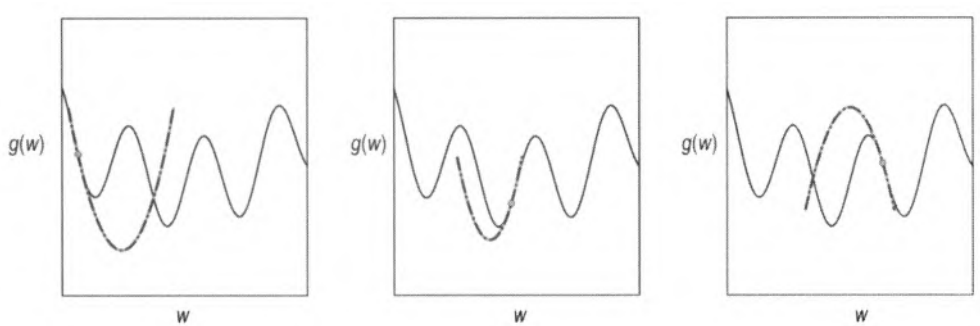


Рис. 4.4. Иллюстрация примера 4.3

4.3. Метод Ньютона

Поскольку аппроксимация функции с использованием ряда Тейлора первого порядка приводит к локально-оптимизационному каркасу градиентного спуска (*см. разд. 3.5*), кажется интуитивно понятным, что аппроксимации с использованием рядов Тейлора более высоких порядков также могут давать алгоритмы, основанные на спуске. В этом разделе мы введем схему локальной оптимизации, осно-

ванную на аппроксимации с применением ряда Тейлора второго порядка, именуемую *методом Ньютона* (названным в честь его создателя Исаака Ньютона). Поскольку метод Ньютона использует информацию о второй производной, он имеет естественные сильные и слабые стороны по сравнению с градиентным спуском. И мы увидим, что кумулятивный эффект этих компромиссов в целом заключается в том, что метод Ньютона особенно полезен для минимизации *выпуклых* функций умеренного числа входов.

4.3.1. Направление спуска

В нашем обсуждении градиентного спуска мы видели, что аппроксимация с использованием ряда Тейлора первого порядка, сама будучи гиперплоскостью, удобно предоставляет нам направление спуска (*см. разд. 3.3*). Для сравнения: квадратичная функция имеет *стационарные точки*, которые являются глобальными минимумами, когда квадратичная функция выпуклая, и глобальными максимумами, когда она вогнутая. Мы можем довольно легко вычислить стационарную точку (точки) квадратичной функции, используя условие оптимальности первого порядка (*см. разд. 3.2*).

Для одновходного случая аппроксимация с помощью ряда Тейлора второго порядка, центрированная в точке v, приведена в уравнении (4.4). Воспользовавшись условием первого порядка для вычисления стационарной точки $w^\star$ этого квадратичного уравнения (см. пример 3.2) путем установки его производной, равной нулю, мы находим, что

$$w^\star = v - \frac{\frac{d}{dw}g(v)}{\frac{d^2}{dw^2}g(v)}. \tag{4.7}$$

Уравнение (4.7) говорит о том, что для попадания точку $w^\star$ мы должны двигаться от v в *направлении*, задаваемом $\left(\frac{d}{dw}g(v)\right)\Big/\left(\frac{d^2}{dw^2}g(v)\right)$.

Такой же расчет может быть выполнен и в случае многовходной аппроксимации рядом Тейлора второго порядка, показанной в уравнении (4.5). Установив градиент квадратичной аппроксимации равным нулю (как показано в примере 3.4) и вычислив, мы получаем стационарную точку

$$\mathbf{w}^\star = \mathbf{v} - \left(\nabla^2 g(\mathbf{v})\right)^{-1}\nabla g(\mathbf{v}). \tag{4.8}$$

Это решение является прямым аналогом одновходного решения в уравнении (4.7) и действительно сводится к нему при $N = 1$. Оно также говорит о том, что для того, чтобы добраться до стационарной точки $\mathbf{w}^\star$, мы перемещаемся из $\mathbf{v}$ в направлении, задаваемом $-\left(\nabla^2 g(\mathbf{v})\right)^{-1}\nabla g(\mathbf{v})$. Когда это направление может быть направлением спуска? Давайте сначала рассмотрим простой пример, чтобы развить нашу интуицию.

Пример 4.4. Стационарные точки аппроксимирования квадратичных функций

На рис. 4.5, *сверху* представлены графики выпуклой функции

$$g(w) = \frac{1}{50}\left(w^4 + w^2\right) + 0{,}5, \tag{4.9}$$

нарисованные черным цветом, вместе с тремя аппроксимациями рядом Тейлора второго порядка, показанными штрихпунктирной линией синего цвета, каждая из которых центрирована в отдельной входной точке. Точки разложения показаны в виде красного круга, а результаты вычисления функций в этих точках — в виде красных крестиков, стационарная точка $w^\star$ ряда Тейлора второго порядка — в виде зеленого круга, а результаты вычисления как квадратичной аппроксимации, так и самой функции в точке $w^\star$ обозначаются соответственно синим и зеленым крестиками.

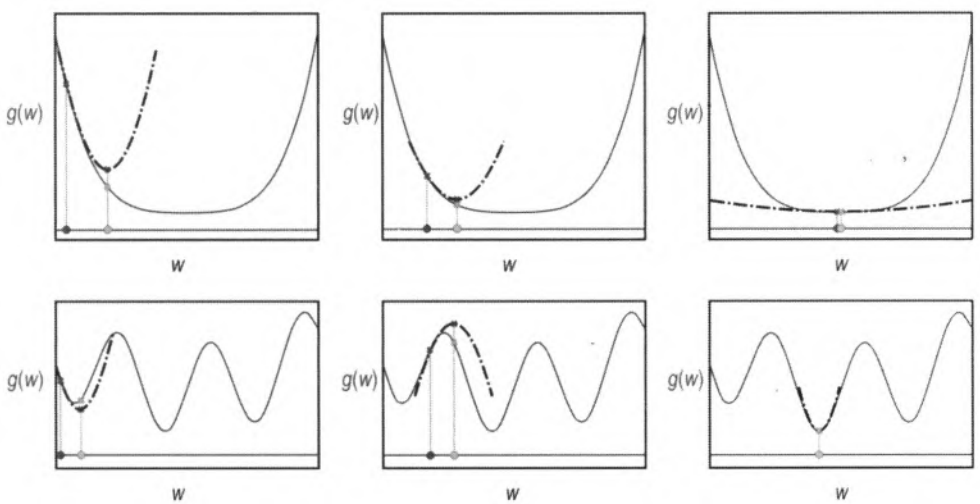


Рис. 4.5. Иллюстрация примера 4.4

Поскольку сама функция g является везде выпуклой, то квадратичная аппроксимация не только совпадает с кривизной в каждой точке, но и всегда является выпуклой и обращена вверх. Поэтому ее стационарная точка всегда является глобальным минимумом. Важно отметить, что минимум квадратичной аппроксимации $w^\star$ всегда приводит к более низкой точке на графике функции, чем вычисленное значение функции в v, т. е. $g\left(w^\star\right) < g(v)$.

На рис. 4.5, *снизу* приведены графики, схожие с описанными выше, только на этот раз для невыпуклой функции

$$g(w) = \sin(3w) + 0{,}1w^2 + 1{,}5. \tag{4.10}$$

Однако сейчас ситуация явно иная, причем виновата невыпуклость. В частности, на вогнутых участках функции (как показано на графике в центре), поскольку квадратичная функция также является вогнутой, стационарная точка $w^\star$ квадратичной

аппроксимации является глобальным максимумом аппроксиматора и стремится привести к точкам, которые *увеличивают* значение функции (а не *уменьшают* его).

Из нашего беглого исследования этих двух простых примеров мы можем интуитивно понять идею схемы локальной оптимизации: многократное перемещение к точкам, определяемым стационарной точкой аппроксимации рядом Тейлора второго порядка. Для выпуклых функций, где стационарная точка каждой квадратичной аппроксимации, по всей видимости, снижает первоначальный результат вычисления исходной функции, эта идея может обеспечить эффективный алгоритм для минимизации функции потерь. Это действительно так, и результирующий алгоритм называется *методом Ньютона*.

4.3.2. Алгоритм

Метод Ньютона — это алгоритм локальной оптимизации, производимый путем многократного итерирования по стационарным точкам аппроксимаций функции рядом Тейлора второго порядка. На k-м шаге этого процесса для функции одной переменной мы выполняем аппроксимацию с использованием ряда Тейлора второго порядка, центрированную в точке w^{k-1}

$$h(w) = g\left(w^{k-1}\right) + \left(\frac{d}{dw} g\left(w^{k-1}\right)\right)\left(w - w^{k-1}\right) + \frac{1}{2}\left(\frac{d^2}{dw^2} g\left(w^{k-1}\right)\right)\left(w - w^{k-1}\right)^2, \quad (4.11)$$

и находим решение для ее неподвижной точки, чтобы создать обновление w^k как

$$w^k = w^{k-1} - \frac{\frac{d}{dw} g\left(w^{k-1}\right)}{\frac{d^2}{dw^2} g\left(w^{k-1}\right)}. \quad (4.12)$$

В общем случае с многовходными функциями, принимающими N-мерный вход, на k-м шаге мы формируем квадратичную аппроксимацию второго порядка

$$h(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T \left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2}\left(\mathbf{w} - \mathbf{w}^{k-1}\right)^T \nabla^2 g\left(\mathbf{w}^{k-1}\right)\left(\mathbf{w} - \mathbf{w}^{k-1}\right) \quad (4.13)$$

и находим решение для стационарной точки этого аппроксиматора, давая обновление $\mathbf{w}^k$ как[1]

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right). \quad (4.15)$$

[1] С точки зрения оптимизации первого порядка, k-й шаг метода Ньютона в уравнении (4.12), применяемый к одновходной функции, также можно рассматривать как шаг градиентного спуска с параметром самокорректирующейся длины шага

$$\alpha = \frac{1}{\frac{d^2}{dw^2} g\left(w^{k-1}\right)}, \quad (4.14)$$

который уточняет длину пройденного шага на основе опорной кривизны функции, подобно взгляду на самокорректирующуюся длину шагов нормализованного градиента, обсуждаемого в *разд. П1.3*. Хотя

Это и есть схема локальной оптимизации, хорошо вписывающаяся в общую форму, которую мы видели в предыдущих двух главах, т. е.

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \mathbf{d}^k, \tag{4.16}$$

где в случае ньютонова метода $\mathbf{d}^k = -\left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right)$ и $\alpha = 1$. Тот факт, что параметр длины шага α здесь имплицитно установлен равным 1, естественно следует из процедуры выведения, которую мы видели.

Обратите внимание, что ньютонова формула обновления в уравнении (4.15) требует, чтобы мы инвертировали $N \times N$-матрицу Гессе (где N — это входная размерность). Однако на практике $\mathbf{w}^k$ в типичной ситуации отыскивается путем решения[2] эквивалентной симметричной системы уравнений

$$\nabla^2 g\left(\mathbf{w}^{k-1}\right)\mathbf{w} = \nabla^2 g\left(\mathbf{w}^{k-1}\right)\mathbf{w}^{k-1} - \nabla g\left(\mathbf{w}^{k-1}\right), \tag{4.17}$$

что может быть сделано экономичнее по сравнению с поиском его решения в замкнутой форме посредством уравнения (4.15).

Как показано на рис. 4.6, *сверху*, для одновходной функции, стартуя в начальной точке w^0, метод Ньютона производит последовательность точек w^1, w^2, ... и т. д., которые минимизируют g, многократно создавая квадратичную аппроксимацию рядом Тейлора второго порядка функции и двигаясь к стационарной точке этой квадратичной функции. Поскольку метод Ньютона на каждом шаге использует квадратичные, а не линейные аппроксимации, причем квадратичная аппроксимация точнее имитирует ассоциированную функцию, он часто бывает гораздо эффективнее градиентного спуска в том смысле, что требует гораздо меньшего числа шагов для схождения [14, 15]. Однако эта зависимость от квадратичной информации также делает метод Ньютона, естественно, более трудным для использования с невыпуклыми функциями, поскольку на вогнутых участках такой функции алгоритм может подняться до локального максимума, как показано на рис. 4.6, *снизу*, или же неконтролируемо осциллировать.

Пример 4.5. Минимизация выпуклой функции с использованием метода Ньютона

На рис. 4.7 показан процесс выполнения метода Ньютона для минимизации функции

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 100w\right) + 0{,}5, \tag{4.18}$$

начиная с точки $w^0 = 2{,}5$, отмеченной зеленым кружком на графике *слева сверху* и соответствующим результатом вычисления функции, отмеченным зеленым крести-

---

эта интерпретация не обобщается непосредственно на многовходный случай в уравнении (4.15), отбросив внедиагональные элементы матрицы Гессе, можно сформировать обобщение этой концепции для многовходного случая. Подробности *см. в разд. П1.8.1.*

[2] Эту систему можно решить с помощью координатного спуска, как описано в *разд. 3.2.2.* Когда существует более одного решения, в типичной ситуации берется наименьшее возможное решение (например, в смысле ℓ_2). Это решение также называется *псевдоинверсией матрицы* $\nabla^2 g(\mathbf{w})$.

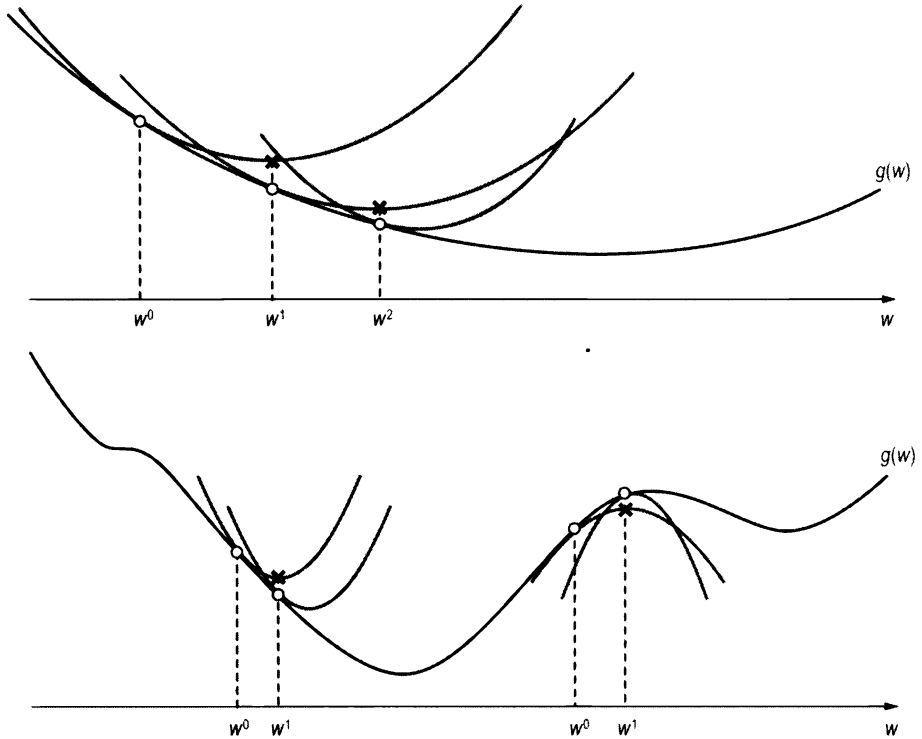


Рис. 4.6. Иллюстрация метода Ньютона. Для того чтобы найти минимум функции g, метод Ньютона перескакивает вниз по стационарным точкам квадратичных аппроксимаций, генерируемых рядом Тейлора второго порядка (*сверху*). Для выпуклых функций эти квадратичные аппроксимации сами по себе всегда являются выпуклыми (единственными стационарными точками которых являются минимумы), и эта последовательность приводит к минимуму исходной функции (*снизу*). Для невыпуклых функций квадратичные аппроксимации могут быть вогнутыми или выпуклыми в зависимости от того, где они построены, что приводит к возможному схождению алгоритма к максимуму

ком. Правый верхний график отображает первый шаг по методу Ньютона, при этом соответствующая квадратичная аппроксимация показана зеленым цветом, ее минимум — красным кружком, а результат вычисления этого минимума на квадратичной функции — синим крестиком. Остальные графики демонстрируют последующие итерации метода Ньютона.

Пример 4.6. Сравнение с градиентным спуском

Как показано на рис. 4.8, *справа*, один-единственный ньютонов шаг — это все, что требуется для полной минимизации выпуклой квадратичной функции

$$g(w_1, w_2) = 0,26w_1^2 + w_2^2 - 0,48w_1w_2 . \tag{4.19}$$

Это может быть сделано за один-единственный шаг, потому что аппроксимация квадратичной функции рядом Тейлора второго порядка — это и есть сама квадра-

тичная функция. Таким образом, метод Ньютона сводится к решению линейной системы первого порядка квадратичной функции. Сравните результат этого одного-единственного ньютонова шага (см. рис. 4.8, *справа*) с соответствующим прогоном 100 шагов градиентного спуска (см. рис. 4.8, *слева*).

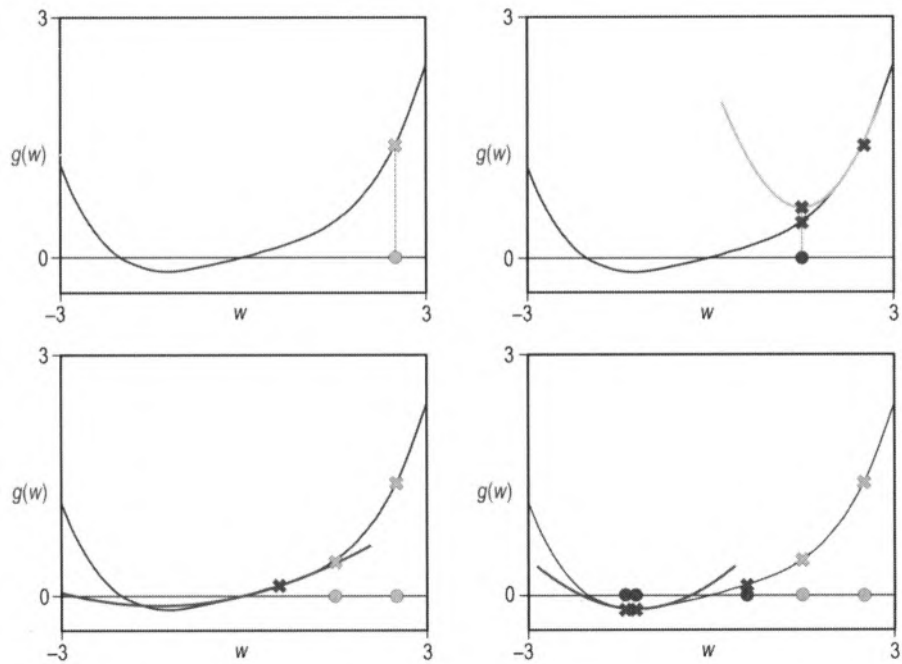


Рис. 4.7. Иллюстрация примера 4.5. Анимирующий прогон метода Ньютона, примененного к функции в уравнении (4.18)

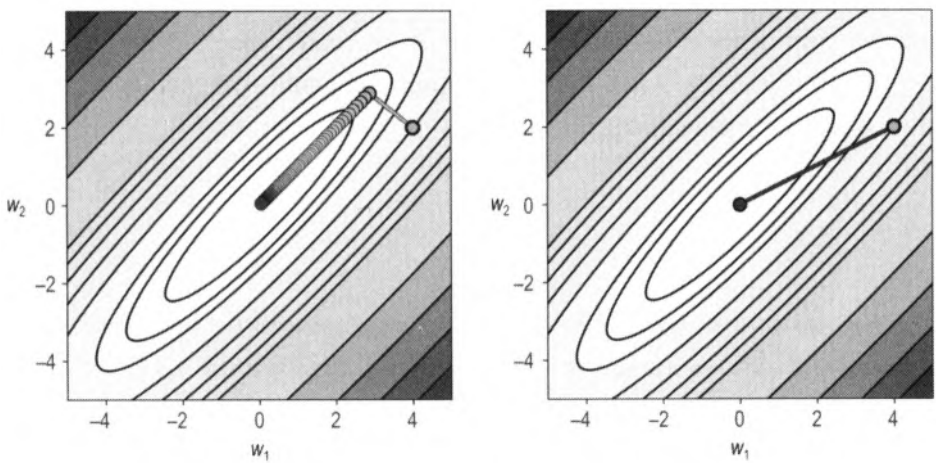


Рис. 4.8. Иллюстрация примера 4.6

4.3.3. Обеспечение числовой стабильности

Вблизи плоских частей функции и числитель $\frac{d}{dw}g\left(w^{k-1}\right)$, и знаменатель $\frac{d^2}{dw^2}g\left(w^{k-1}\right)$ одновходного ньютонова обновления в уравнении (4.12) имеют малые значения, близкие к нулю. Это может вызвать серьезные численные проблемы, как только каждый (но особенно знаменатель) становится меньше порога *прецизионности машины* (машинной точности), т. е. наименьшего значения, которое компьютер может интерпретировать как ненулевое.

Один простой и распространенный способ избежать этой потенциальной проблемы *деления на ноль* состоит в добавлении в знаменатель малого положительного значения ε, когда он уменьшается ниже определенного значения либо для всех итераций. Тогда этот *регуляризованный* ньютонов шаг принимает вид

$$w^k = w^{k-1} - \frac{\frac{d}{dw}g\left(w^{k-1}\right)}{\frac{d^2}{dw^2}g\left(w^{k-1}\right)+\varepsilon}. \tag{4.20}$$

Значение *регуляризационного параметра* ε в типичной ситуации устанавливается равным малому положительному[3] значению (например, 10^{-7}).

Аналогичная корректировка для общего многовходного обновления Ньютона заключается в добавлении $\varepsilon\mathbf{I}_{N\times N}$ ($N\times N$-матрицы тождественности, прошкалированной малым положительным значением ε) в матрицу Гессе уравнения (4.15), в результате чего мы имеем[4]

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)+\mathbf{I}_{N\times N}\right)^{-1}\nabla g\left(\mathbf{w}^{k-1}\right). \tag{4.21}$$

Добавление этого дополнительного члена в гессиан гарантирует, что матрица $\nabla^2 g\left(\mathbf{w}^{k-1}\right)+\mathbf{I}_{N\times N}$ всегда будет обратимой, при условии, что для ε используется достаточно большое значение.

---

[3] Эта корректировка производится, когда известно, что минимизируемая функция является выпуклой, т. к. в этом случае $\frac{d^2}{dw^2}g(w)\geq 0$ для всех w.

[4] Как и в случае исходного шага Ньютона в уравнении (4.15), практически всегда эффективнее найти это обновление численно, решив соответствующую линейную систему $\left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)+\varepsilon\mathbf{I}_{N\times N}\right)\mathbf{w} = \left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)+\varepsilon\mathbf{I}_{N\times N}\right)\mathbf{w}^{k-1} - \nabla g\left(\mathbf{w}^{k-1}\right)$ для w.

4.3.4. Варианты длины шага

Хотя в процессе выведения метода Ньютона мы видели, что (будучи локально оптимизационным подходом) он действительно имеет параметр α длины шага, который неявно устанавливается равным $\alpha = 1$ и поэтому кажется "невидимым". Однако в принципе можно ввести параметр α длины шага явно и использовать корректируемые методы (например, линейный поиск[5] с отслеживанием в обратном порядке, как это представлено в *разд. П1.4*) для его тонкой настройки. Тогда ньютонов шаг, взвешенный явным образом, принимает форму

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right), \tag{4.22}$$

причем стандартный ньютонов шаг проявляется при $\alpha = 1$.

4.3.5. Метод Ньютона как алгоритм отыскания нуля

Метод Ньютона сначала был изобретен не как алгоритм локальной оптимизации, а как алгоритм отыскания нуля. Другими словами, метод Ньютона с самого начала был изобретен для отыскания нулей функции f, т. е. $f(\mathbf{w}) = \mathbf{0}_{N\times 1}$. Традиционно рассматриваемая функция f была своего рода многочленной функцией. В контексте локальной оптимизации мы можем рассматривать метод Ньютона как подход для *итерационного* решения системы первого порядка (*см. разд. 3.2*):

$$\nabla g(\mathbf{v}) = \mathbf{0}_{N\times 1}. \tag{4.23}$$

Возьмем случай, когда наша входная размерность $N = 1$. Вообще говоря, отыскание нулей произвольной функции не является тривиальным делом. Вместо того чтобы пытаться решить уравнение первого порядка $\frac{d}{dw} g(v) = 0$ непосредственно, давайте попробуем сформулировать итерационную процедуру, в которой нужно будет находить приближенное решение этого уравнения, решая связанную последовательность более простых задач. Следуя той же логике, которую мы использовали при выведении формул градиентного спуска и метода Ньютона ранее, вместо поиска нуля самой функции попробуем отыскать ноль касательной, обеспечиваемой аппроксимацией нашей функции с использованием *ряда Тейлора первого порядка*. Отыскание точки (точек), в которой линия или — в более общем смысле — гиперплоскость равна нулю, является сравнительно тривиальным делом.

Перед записью первого шага этой схемы следует помнить о том, что мы рассуждаем об этой процедуре как об итеративном методе, применяемом к производной функции $\frac{d}{dw} g(w)$. Это означает, что, начиная с точки w^0, наша линейная аппроксимация рядом Тейлора первого порядка производной функции

$$h(w) = \frac{d}{dw} g\left(w^0\right) + \frac{d^2}{dw^2} g\left(w^0\right)\left(w - w^0\right) \tag{4.24}$$

---

[5] Также именуемый поиском по прямой (line search). — *Прим. пер.*

естественным образом включает в себя вторую производную функции g (она, в конце концов, является аппроксимацией первого порядка производной этой функции). Мы можем легко определить точку, в которой эта линия пересекает входную ось, приравняв приведенное выше уравнение нулю и решив его. Сделав это и обозначив решение через w^1, мы получим:

$$w^1 = w^0 - \frac{\frac{d}{dw}g\left(w^0\right)}{\frac{d^2}{dw^2}g\left(w^0\right)}. \tag{4.25}$$

При ближайшем рассмотрении мы видим, что это действительно ньютонов шаг. Поскольку мы нашли ноль лишь линейной аппроксимации $\frac{d}{dw}g(w)$, а не самой функции, будет естественным повторить эту процедуру для уточнения нашей аппроксимации. На k-м таком шаге наше обновление принимает форму

$$w^k = w^{k-1} - \frac{\frac{d}{dw}g\left(w^{k-1}\right)}{\frac{d^2}{dw^2}g\left(w^{k-1}\right)}, \tag{4.26}$$

что в точности соответствует форме ньютонова шага в уравнении (4.12). Аналогичное рассуждение в отношении многовходных функций (где $N > 1$), начинающееся с желания итеративно решить систему первого порядка, приводит к выведению многовходного ньютонова шага в уравнении (4.15).

4.3.6. Имплементация на Python

В этом разделе представлена простая имплементация метода Ньютона на языке Python, использующая отличные библиотеки автоматического дифференцирования autograd и NumPy (*см. разд. 3.4* и *П2.10*). В частности, задействованы модули grad и hessian из библиотеки autograd для автоматического вычисления первой и второй производных общей входной функции.

```
# импортировать автоматический дифференциатор из библиотеки autograd
from autograd import grad
from autograd import hessian

# импортировать библиотеку NumPy
import numpy as np

# метод Ньютона
def newtons_method(g, max_its, w, **kwargs):

    # вычислить градиент/гессиан, используя autograd
    gradient = grad(g)
    hess = hessian(g)

```

```
    # установить параметр числовой стабильности
    epsilon = 10**(-7)
    if 'epsilon' in kwargs:
        epsilon = kwargs['epsilon']

    # выполнить цикл метода Ньютона
    weight_history = [w]  # контейнер для истории веса
    cost_history = [g(w)] # контейнер для истории стоимостной функции
    for k in range(max_its):

        # оценить градиент и гессиан
        grad_eval = gradient(w)
        hess_eval = hess(w)

        # реформировать гессиан в квадратную матрицу
        hess_eval.shape = (int((np.size(hess_eval))**(0.5)),int((np.size(hess_
                                                                eval))**(0.5)))

        # решить систему второго порядка для обновления веса
        A = hess_eval + epsilon*np.eye(w.size)
        b = grad_eval
        w = np.linalg.solve(A, np.dot(A,w)-b)

        # записать вес и стоимость
        weight_history.append(w)
        cost_history.append(g(w))

    return weight_history,cost_history
```

Обратите внимание, что, хотя мы использовали критерий схождения, который был основан на максимуме итераций, потенциально высокая вычислительная стоимость каждого ньютонова шага часто стимулирует использование более формальных критериев схождения (например, остановка, когда норма градиента становится меньше заранее определенного порога). Это также часто стимулирует включение контрольных точек, которые измеряют и/или корректируют ход выполнения метода Ньютона во избежание проблем вблизи плоских областей функции. Кроме того, для этой имплементации метода Ньютона можно использовать тот же тип инициализации, что и для градиентного спуска в *разд. 3.5.4*.

4.4. Два естественных недостатка метода Ньютона

Метод Ньютона — это мощный алгоритм, который весьма эффективен в поиске минимума функции на каждом шаге по сравнению с методами нулевого и первого порядка, для достижения эквивалентной прогрессии которых может потребоваться большое число шагов. Поскольку в нем задействуется информация как о первых, так и о вторых производных (т. е. кривизна), метод Ньютона не страдает от про-

блем, присущих методам первого порядка (например, от проблемы зигзагообразного поведения, которую мы видели в *разд. 3.6.3*). Однако метод Ньютона имеет свои уникальные недостатки, в первую очередь те, что касаются *невыпуклости*, а также *масштабирования* входной размерности, которые мы кратко обсудим далее. Хотя эти недостатки не мешают методу Ньютона (как мы его описали) широко использоваться в машинном обучении, они все-таки заслуживают внимания.

4.4.1. Минимизация невыпуклых функций

Как обсуждалось в предыдущем разделе, метод Ньютона может вести себя очень плохо, когда применяется для поиска минимумов невыпуклых функций. Поскольку каждый шаг основан на аппроксимации функцией второго порядка, метод Ньютона, инициированный на *вогнутом* участке, будет естественным образом делать шаг *вверх*. Этот факт проиллюстрирован для абстрактной невыпуклой функции рис. 4.6, *снизу*. Заинтересованный читатель может ознакомиться с *разд. П1.7*, где мы опишем простой и распространенный подход к корректировке метода Ньютона для решения этой проблемы.

4.4.2. Ограничения по масштабированию

Поскольку квадратичная аппроксимация, используемая методом Ньютона, очень хорошо совпадает с функцией *локально*, этот метод может сходиться к глобальному минимуму за гораздо меньшее число шагов (чем методы первого порядка), в особенности, когда он близок к минимуму. Однако в вычислительном плане шаг метода Ньютона обходится намного дороже, чем шаг метода первого порядка, и требует хранения и вычисления не только градиента, но и всей $N \times N$-матрицы Гессе с информацией о второй производной. Если просто хранить гессиан для одиночного шага ньютонова метода с ее N^2 элементами, то это может быстро усложнить задачу для входных данных даже среднего размера. Например, если вход в функцию имеет размерность $N = 10\,000$, то соответствующая матрица Гессе будет содержать $100\,000\,000$ элементов. Функции, используемые в приложениях машинного обучения, могут легко содержать от десятков тысяч до сотен тысяч или даже миллионов входов, что делает невозможным полное хранение ассоциированного гессиана.

Позже в *разд. П1.8* мы обсудим базовые способы смягчения этой проблемы, включая корректировку базового шага метода Ньютона путем замены гессиана какой-нибудь аппроксимацией, которая не страдает от этой врожденной проблемы масштабирования.

4.5. Заключение

В данной главе мы завершаем наше обсуждение математической оптимизации в этой части книги описанием методов оптимизации второго порядка, т. е. тех, которые используют как первую, так и вторую производную функции при формировании направлений спуска.

В *разд. 4.2* мы начали с рассмотрения условия оптимальности второго порядка. Затем мы кратко коснулись кривизны функции, определенной ее второй производной в *разд. 4.1*, прежде чем сразу применить эту концепцию в детализации краеугольного метода локальной оптимизации второго порядка — *метода Ньютона* — в *разд. 4.3*.

Потом в *разд. 4.4* мы коснулись двух естественных проблем алгоритма на основе метода Ньютона — его применения для минимизации невыпуклых функций и к функциям с высокоразмерным входом. Заинтересованный читатель должен обратить внимание на то, что в *разд. П1.7* и *П1.8* подробно описаны общие корректировки стандартной ньютоновой схемы для смягчения этих проблем, причем последний набор корректировок называется *бесгессиановой* оптимизацией.

4.6. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:

github.com/jermwatt/machine_learning_refined.

4.1. Определение собственных значений симметричной матрицы

В данном упражнении мы проследим альтернативный подход к проверке того, что все собственные значения симметричной $N \times N$-матрицы $\mathbf{C}$ (например, матрицы Гессе) являются неотрицательными. Этот подход не предусматривает явного вычисления самих собственных значений и значительно легче задействуется на практике.

А) Пусть $\mathbf{C}$ есть симметричная $N \times N$-матрица. Показать, что если у $\mathbf{C}$ все собственные значения неотрицательные, то величина $\mathbf{z}^T\mathbf{C}\mathbf{z} \geq 0$ для всех z. Подсказка: использовать разложение $\mathbf{C}$ на собственные значения (*см. разд. П2.4*).

Б) Показать обратное, т. е. если симметричная $N \times N$-матрица $\mathbf{C}$ удовлетворяет $\mathbf{z}^T\mathbf{C}\mathbf{z} \geq 0$ для всех z, то она должна иметь все неотрицательные собственные значения.

В) Использовать этот метод для проверки того, что определение выпуклости второго порядка соблюдается для квадратичной функции

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}, \tag{4.27}$$

где $a = 1$, $\mathbf{b} = [1 \quad 1]^T$ и

$$\mathbf{C} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

Г) Показать, что все собственные значения $\mathbf{C} + \lambda\mathbf{I}_{N\times N}$ можно сделать положительными, установив λ достаточно большим. Каково наименьшее значение λ, которое заставит это произойти?

4.2. Матрицы внешнего произведения имеют все неотрицательные собственные значения

А) Использовать метод, описанный в упражнении 4.1, и проверить, что для любого $N \times 1$-вектора $\mathbf{x}$ все собственные значения $N \times N$-матрицы внешнего произведения $\mathbf{x}\mathbf{x}^T$ являются неотрицательными.

Б) Схожим образом показать, что для любого множества P векторов $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_P$ длины N все собственные значения суммы матриц внешнего произведения $\sum_{p=1}^{P} \delta_p \mathbf{x}_p \mathbf{x}_p^T$ являются неотрицательными, если для каждой из них $\delta_p \geq 0$.

В) Показать, что у матрицы $\sum_{p=1}^{P} \delta_p \mathbf{x}_p \mathbf{x}_p^T + \lambda \mathbf{I}_{N \times N}$, где $\delta_p \geq 0$ и $\lambda > 0$, все собственные значения являются положительными.

4.3. Альтернативный способ проверки определения выпуклости второго порядка

Напомним, что определение выпуклости второго порядка для многовходной функции $g(\mathbf{w})$ требует, чтобы мы проверили, являются ли собственные значения матрицы $\nabla^2 g(\mathbf{w})$ неотрицательными для каждого входа $\mathbf{w}$. Однако задача явного вычисления собственных значений гессиана с целью проверки этого условия является громоздкой или даже невозможной для всех функций, кроме самой хорошей из них. Воспользуемся результатом упражнения 4.1, чтобы выразить определение выпуклости второго порядка способом, который часто гораздо легче задействовать на практике.

А) Использовать результат упражнения 4.1 и выяснить, что неотрицательность собственных значений матрицы $\nabla^2 g(\mathbf{w})$ обеспечивает неотрицательность самой матрицы в каждой точке $\mathbf{w}$ и выражается эквивалентным образом как неравенство $\mathbf{z}^T \left(\nabla^2 g(\mathbf{w})\right) \mathbf{z} \geq 0$, соблюдаемое в каждой точке $\mathbf{w}$ для всех $\mathbf{z}$.

Б) Использовать этот способ выражения определения выпуклости второго порядка и проверить, что общая квадратичная функция

$$g(\mathbf{w}) = a + \mathbf{b}^T \mathbf{w} + \mathbf{w}^T \mathbf{C} \mathbf{w},$$

где $\mathbf{C}$ — симметричная матрица, у которой все собственные значения являются неотрицательными, всегда определяет выпуклую функцию.

В) Проверить, что $g(\mathbf{w}) = -\cos\left(2\pi \mathbf{w}^T \mathbf{w}\right) + \mathbf{w}^T \mathbf{w}$ не является выпуклой, показав, что она не удовлетворяет определению выпуклости второго порядка.

4.4. Метод Ньютона, вариант I

Повторить эксперимент, описанный в примере 4.5. Вместо построения графика результирующего пути, проходимого методом Ньютона (см. рис. 4.7), построить график истории стоимостной функции и убедиться, что ваш алгоритм правильно

сходится к точке, близкой к глобальному минимуму функции. В качестве основы для этого упражнения вы можете задействовать имплементацию метода Ньютона, описанную в *разд. 4.3.6*.

4.5. Метод Ньютона, вариант II

А) Использовать условие оптимальности первого порядка (*см. разд. 3.2*) для определения единственной стационарной точки функции $g(\mathbf{w}) = \log\left(1 + e^{\mathbf{w}^T\mathbf{w}}\right)$, где $\mathbf{w}$ — двумерна (т. е. $N = 2$).

Б) Использовать определение выпуклости второго порядка для проверки того, что $g(\mathbf{w})$ является выпуклой, из чего вытекает, что стационарная точка, найденная в части (А), является глобальным минимумом. Подсказка: для проверки определения второго порядка использовать упражнение 4.2.

В) Выполнить метод Ньютона, чтобы найти минимум функции $g(\mathbf{w})$, определенной в части (А). Инициализировать свой алгоритм в точке $\mathbf{w}^0 = \mathbf{1}_{N\times 1}$ и построить график истории стоимостной функции для десяти итераций метода Ньютона, чтобы убедиться, что ваш алгоритм работает правильно и сходится. В качестве основы для этой части упражнения вы можете использовать имплементацию, приведенную в *разд. 4.3.6*.

Г) Выполнить код метода Ньютона из части (В), на этот раз инициализировав процесс в точке $\mathbf{w}^0 = 4 \cdot \mathbf{1}_{N\times N}$. Хотя эта инициализация находится еще дальше от уникального минимума $g(\mathbf{w})$, чем та, которая используется в части (В), ваш алгоритм на основе метода Ньютона, начиная с этой точки, должен сходиться *быстрее*. На первый взгляд этот результат кажется очень нелогичным, т. к. мы (по праву) ожидаем, что начальная точка, находящаяся ближе к минимуму, спровоцирует более быстрое схождение метода Ньютона! Объяснить, почему этот результат на самом деле имеет смысл для конкретной функции $g(\mathbf{w})$, которую мы здесь минимизируем.

4.6. Отыскание квадратных корней

Использовать метод Ньютона для вычисления квадратного корня из 999. Кратко объяснить, как была настроена соответствующая стоимостная функция, которая была минимизирована для получения этого квадратного корня. Объяснить, как были для этого использованы методы оптимизации нулевого и первого порядков (подробно описанные в *главах 2* и *3*).

4.7. Невыпуклая минимизация с использованием метода Ньютона

Использовать (регуляризованный) метод Ньютона для минимизации функции

$$g(w) = \cos w, \tag{4.28}$$

начав в точке $w = 0{,}1$. В частности, убедиться, что на *каждом* шаге метода Ньютона достигается уменьшение значения функции.

4.8. Ньютонов спуск

А) Показать, что, когда $g(\mathbf{w})$ является выпуклой, ньютонов шаг в уравнении (4.15) действительно уменьшает результат вычисления функции g, т. е. $g\left(\mathbf{w}^k\right) \le g\left(\mathbf{w}^{k-1}\right)$.

Б) Показать независимо от минимизируемой функции g, что ε в уравнении (4.21) можно задать достаточно большим, чтобы соответствующий ньютонов шаг мог привести к нижней части функции, т. е. $g\left(\mathbf{w}^k\right) \le g\left(\mathbf{w}^{k-1}\right)$.

4.9. Метод Ньютона как самокорректирующийся метод градиентного спуска

Имплементировать подвыборочный ньютонов шаг, описанный в *разд. П1.8.1* и приведенный в уравнении (П1.78), сформированный за счет игнорирования всех внедиагональных элементов гессиана, и сравнить его с градиентным спуском с использованием тестовой функции

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}, \tag{4.29}$$

где

$$a = 0,\ \mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},\ \mathbf{C} = \begin{bmatrix} 0,5 & 2 \\ 1 & 9,75 \end{bmatrix}.$$

Выполнить прогон каждого локального метода за 25 шагов, стартуя в начальной точке $\mathbf{w}^0 = [10 \quad 1]^T$ и используя наибольшее фиксированное значение длины шага формы 10^γ (где γ — целое число) для градиентного спуска. Создать контурный график тестовой функции и нанести шаги каждого прогона поверх него, чтобы визуализировать работу каждого алгоритма.

4.10. Метод Бройдена — Флетчера — Гольдфарба — Шанно (BFGS)

Начать с того же самого допущения, что и в примере П1.12 (т. е. рекурсии, основанной на двухранговой разнице между $\mathbf{S}^k$ и его предшественником), и задействовать условие секущей с целью выведения рекурсивного обновления для $\mathbf{S}^k$ в терминах $\mathbf{S}^{k-1}$, $\mathbf{a}^k$ и $\mathbf{b}^k$. Затем использовать тождество Шермана — Моррисона, чтобы переписать свое обновление в терминах $\mathbf{F}^k$ — величины, обратной $\mathbf{S}^k$.

Часть II

Линейное обучение с учителем

5 Линейная регрессия

5.1. Введение

В этой главе формально описана задача обучения с учителем, именуемая *линейной регрессией*, или подгонкой репрезентативной прямой (или, в более высоких размерностях, гиперплоскостью) ко множеству точек входных/выходных данных, как это впервые было подробно рассмотрено в *разд. 1.3.1*. Регрессия может выполняться по целому ряду причин: для получения так называемой линии тренда (или в более общем смысле — как мы увидим позже — кривой), которая может использоваться для оказания помощи в визуальном подытоживании, доведении конкретной точки зрения об исследуемых данных или в усвоении модели, благодаря которой можно сделать точные предсказания относительно выходных значений в будущем. В приведенном здесь описании мы рассмотрим различные способы формулирования надлежащих функций потерь для регрессии, включая обсуждение *метода наименьших квадратов* и *наименьших абсолютных отклонений*, а также надлежащие метрики для измерения качества натренированного регрессора (т. е. того, чьи параметры были полностью оптимизированы) и различные часто встречающиеся расширения базовой концепции регрессии (включая взвешенную и многовыходную регрессию).

5.2. Линейная регрессия на основе наименьших квадратов

В этом разделе мы формально введем задачу *линейной регрессии* или подгонки репрезентативной прямой (или гиперплоскости в более высоких размерностях) к множеству входных/выходных точек данных. Мы также рассмотрим формулировку популярной функции потерь *наименьших квадратов*, широко задействуемой для надлежащей настройки параметров регрессора в целом.

5.2.1. Нотация и моделирование

Данные для задач регрессии обычно поступают в виде множества из P пар входных/выходных наблюдений

$$(\mathbf{x}_1, y_1),\ (\mathbf{x}_2, y_2),\ \ldots,\ (\mathbf{x}_P, y_P) \tag{5.1}$$

или $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ для краткости, где $\mathbf{x}_p$ и y_p обозначают соответственно вход и выход p-го наблюдения. Каждый вход $\mathbf{x}_p$ — это в общем случае вектор-столбец длины N

$$\mathbf{x}_p = \begin{bmatrix} x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{N,p} \end{bmatrix}, \tag{5.2}$$

а каждый выход y_p — скалярное значение (векторнозначные выходы рассмотрены в *разд. 5.6*). Геометрически задача линейной регрессии тогда состоит в подгонке *гиперплоскости* к разбросу точек в $(N+1)$-мерном пространстве.

В простейшем случае, когда входные данные тоже являются скалярнозначными (например, $N = 1$), линейная регрессия упрощается до подгонки *прямой* к ассоциированному разбросу точек данных в двумерном пространстве. Прямая в двух размерностях определяется двумя параметрами: вертикальным пересечением w_0 и наклоном w_1. Мы должны установить значения этих параметров таким образом, чтобы между входными и выходными данными соблюдалась следующая приближенная линейная связь

$$w_0 + x_p w_1 \approx y_p, \quad p = 1, ..., P. \tag{5.3}$$

Обратите внимание, что в уравнении (5.3) мы использовали знак приближенного равенства, потому что мы никогда не можем быть абсолютно уверены в том, что абсолютно все данные лежат на одной линии. В более общем случае, когда мы имеем дело с N-мерным входом, у нас есть вес смещения и N связанных с ним весов наклона, которые нужно надлежаще настроить, чтобы вписать гиперплоскость, с аналогичной линейной связью, записываемой как

$$w_0 + x_{1,p} w_1 + x_{2,p} w_2 + ... + x_{N,p} w_N \approx y_p, \quad p = 1, ..., P. \tag{5.4}$$

Как одновходные, так и общие многовходные случаи линейной регрессии фигурально проиллюстрированы на рис. 5.1. На языке машинного обучения каждая размерность входа называется *признаком* или *входным признаком*. Поэтому мы будем часто ссылаться на параметры w_1, w_2, ..., w_N как на *признаковые веса*, т. е. которые касаются признаков, при этом единственным непризнаковым весом является смещение w_0.

Линейная связь в уравнении (5.4) может быть записана компактнее с помощью нотации $\dot{\mathbf{x}}$ для обозначения входа $\mathbf{x}$ с помещенной 1 в качестве его первого элемента. Эта форма записи означает, что мы укладываем 1 поверх каждой нашей входной точки $\mathbf{x}_p$:

$$\dot{\mathbf{x}}_p = \begin{bmatrix} 1 \\ x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{N,p} \end{bmatrix} \tag{5.5}$$

для всех $p = 1, ..., P$. Теперь, поместив все параметры в один вектор-столбец $\mathbf{w}$

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}, \tag{5.6}$$

мы можем записать общие желаемые линейные связи из уравнения (5.4) компактнее:

$$\dot{\mathbf{x}}_p^T \mathbf{w} \approx y_p, \quad p = 1, ..., P. \tag{5.7}$$

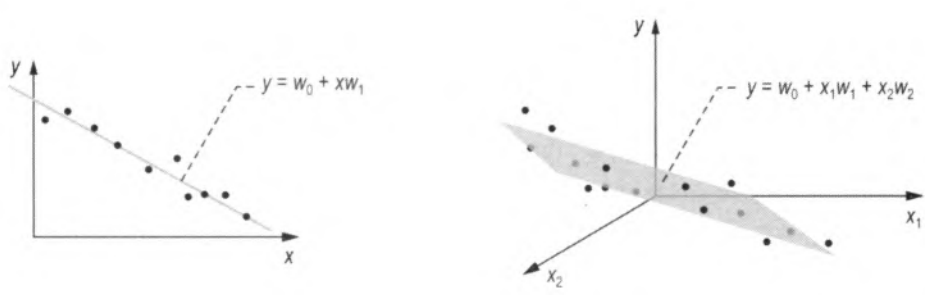


Рис. 5.1. Симулированный набор данных в двух размерностях вместе с хорошо подогнанной прямой (*слева*). Прямая в двух размерностях определяется как $w_0 + xw_1 = y$, где w_0 называется смещением, а w_1 — наклоном, и точка (x_p, y_p) лежит близко к ней, если $w_0 + x_p w_1 \approx y_p$. Симулированный трехмерный набор данных вместе с хорошо подогнанной гиперплоскостью (*справа*). Гиперплоскость в общем случае определяется как $w_0 + x_1 w_1 + x_2 w_2 + ... + x_N y_N = y$, где w_0 — это смещение, а w_1, w_2, ..., w_N — координатные наклоны гиперплоскости, и точка ($\mathbf{x}_p$, y_p) лежит близко к ней, если $w_0 + x_{1,p} w_1 + x_{2,p} w_2 + ... + x_{N,p} y_N \approx y_p$. Здесь $N = 2$

5.2.2. Стоимостная функция наименьших квадратов

Для того чтобы отыскать параметры гиперплоскости, которые наилучшим образом подогнаны к регрессионному набору данных, мы должны сначала сформировать функцию потерь, которая служит мерой того, насколько хорошо линейная модель с конкретным выбором параметров вписывается в регрессионные данные. Один из

наиболее популярных вариантов для этого называется стоимостной функцией *наименьших квадратов*. Для заданного набора параметров в векторе $\mathbf{w}$ эта стоимостная функция вычисляет общую *квадратическую ошибку* между ассоциированной гиперплоскостью и данными (рис. 5.2). Естественно, тогда оптимально подогнанной гиперплоскостью является та, параметры которой *минимизируют* эту ошибку.

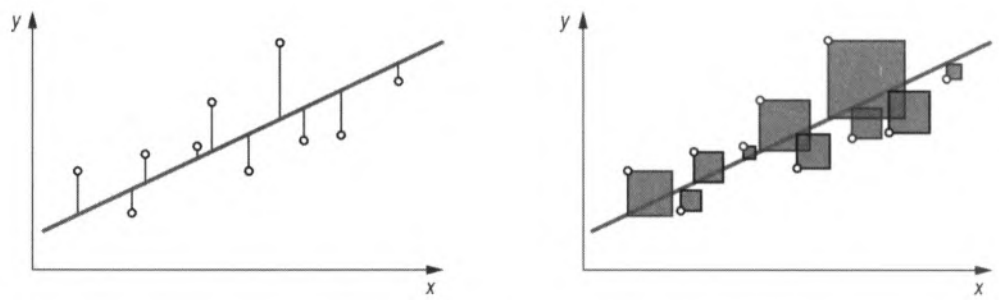


Рис. 5.2. Прототипический двумерный набор данных вместе с прямой, подогнанной к данным с использованием метода наименьших квадратов, который призван восстановить линейную модель, минимизирующую общую квадратичную длину сплошных линий ошибок (*слева*).
Ошибка наименьших квадратов может рассматриваться как общая площадь синих квадратов со сплошными черными линиями ошибок в качестве сторон (*справа*).
Стоимостная функция *наименьших квадратов* называется так потому, что она позволяет определять набор параметров, соответствующая прямая которых минимизирует сумму квадратов отклонений.

Сосредоточившись только на p-й аппроксимации в уравнении (5.7), мы в идеале хотим, чтобы линейные связи $\dot{\mathbf{x}}_p^T\mathbf{w}$ были как можно ближе к p-му выходу y_p, или, что эквивалентно, чтобы ошибка или разница между ними, т. е. $\dot{\mathbf{x}}_p^T\mathbf{w} - y_p$, была как можно меньше. Путем возведения этой величины в квадрат (так, чтобы и отрицательные, и положительные ошибки одинаковой магнитуды рассматривались одинаково) мы можем определить

$$g_p(\mathbf{w}) = \left(\dot{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2, \tag{5.8}$$

как *точечную стоимостную функцию*, которая измеряет ошибку модели (здесь линейной модели) на отдельной точке $(\mathbf{x}_p, y_p)$. Теперь, поскольку мы хотим, чтобы все P таких значений были малы одновременно, мы можем взять их среднее значение над всем набором данных, сформировав стоимостную функцию наименьших квадратов[1] для линейной регрессии

[1] Технически говоря, стоимостная функция наименьших квадратов $g(\mathbf{w})$ является функцией как весов $\mathbf{w}$, так и данных. Однако для нотационной простоты мы часто предпочитаем не показывать зависимость от данных явно. В противном случае нам пришлось бы писать стоимостную функцию как $g\left(\mathbf{w}, \left\{\left(\dot{\mathbf{x}}_p^T, y_p\right)\right\}_{p=1}^P\right)$, в результате чего все начинает становиться слишком запутанным. Более того, для

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} g_p(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\dot{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2. \quad (5.9)$$

Обратите внимание, что чем больше стоимость наименьших квадратов, тем больше квадратическая ошибка между соответствующей линейной моделью и данными, и, следовательно, тем слабее мы представляем конкретный набор данных линейной моделью. Следовательно, мы хотим отыскать оптимальный параметрический вектор $\mathbf{w}$, *минимизирующий* $g(\mathbf{w})$, или, выражаясь формально, мы хотим решить задачу неограниченной оптимизации:

$$\text{минимизировать } \frac{1}{P}\sum_{p=1}^{P}\left(\dot{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2, \quad (5.10)$$

используя инструменты локальной оптимизации, подробно описанные в *главах 2–4*.

5.2.3. Минимизация стоимостной функции наименьших квадратов

Можно доказать, что стоимостная функция наименьших квадратов для линейной регрессии в уравнении (5.9) является *выпуклой* для любого набора данных (*см. разд. 5.9*). В примере 5.1 мы демонстрируем этот факт, используя линейно-регрессионный набор данных.

Пример 5.1. Верификация выпуклости путем визуального обследования

На рис. 5.3, *сверху* показан линейно-регрессионный набор данных, состоящий из $P = 50$ пар входов/выходов, случайно отобранных вне прямой $y = x$, с небольшим количеством случайного шума, добавленного к каждому выходу. На рис. 5.3, *слева снизу* построен график трехмерной поверхности стоимостной функции наименьших квадратов, связанной с этим набором данных, а ее контурный график показан в двух размерностях на рис. 5.3, *справа снизу*. Мы видим, по *изгибу вверх* поверхности стоимостной функции слева или по эллиптической форме ее контурных линий справа, что для этого конкретного набора данных стоимостная функция наименьших квадратов действительно является выпуклой.

Благодаря выпуклости и по причине того, что стоимостная функция наименьших квадратов является *бесконечно дифференцируемой*, мы можем применить практически любую схему локальной оптимизации, чтобы минимизировать ее должным образом. Однако общие практические соображения, связанные с каждым методом локальной оптимизации, по-прежнему следует учитывать: методы нулевого и второго порядков не масштабируются изящно, и при градиентном спуске мы должны выбирать фиксированное значение длины шага, схему убывающей длины шага или

конкретного набора данных веса $\mathbf{w}$ являются важным входом в функцию, поскольку именно их нам нужно настроить, чтобы получить хорошую подгонку. С точки зрения оптимизации сам набор данных считается *фиксированным*. Мы будем делать такое же нотационное упрощение практически для всех будущих стоимостных функций машинного обучения, которые мы здесь изучаем.

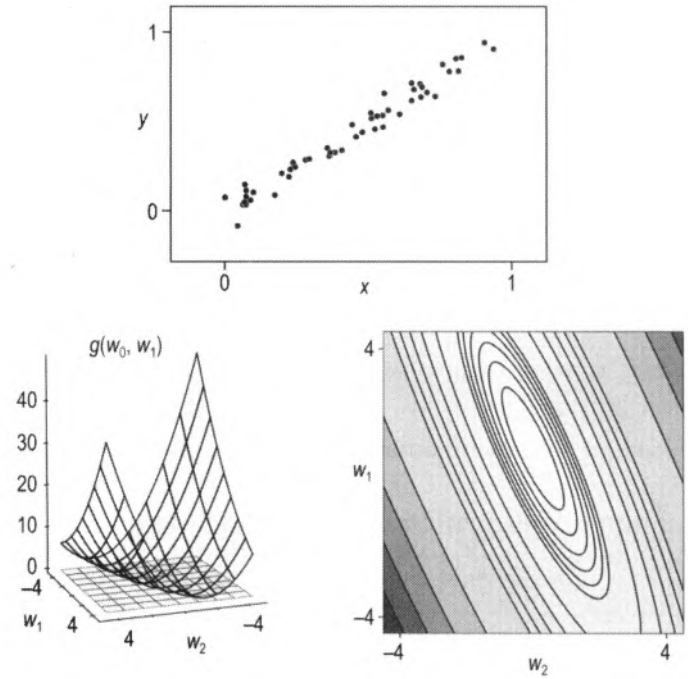


Рис. 5.3. Иллюстрация примера 5.1

корректируемый метод, такой как линейный поиск с отслеживанием в обратном порядке (как подробно описано в *главе 3*). Поскольку стоимость наименьших квадратов является выпуклой квадратичной функцией, один-единственный ньютонов шаг может полностью ее минимизировать. Этот подход иногда называется минимизированием наименьших квадратов путем решения его *нормальных уравнений* (дальнейшее обсуждение этой темы см. в упражнении 5.3).

Пример 5.2. Использование градиентного спуска

На рис. 5.4 показан результат минимизирования стоимости наименьших квадратов с использованием набора данных, представленного в примере 5.1. Мы используем градиентный спуск и задействуем фиксированное значение длины шага $\alpha = 0,5$ для всех 75 шагов до тех пор, пока приближенно не достигнем минимума функции.

На рисунке показана прогрессия процесса градиентного спуска (слева направо как с точки зрения минимизации стоимости наименьших квадратов (верхний ряд), так и с точки зрения прямой, обеспечиваемой соответствующими весами (нижний ряд)). Шаги оптимизации в верхнем ряду окрашены от зеленого в начале выполнения (крайний левый рисунок) до красного в его финале (крайний правый рисунок). Линейная модель окрашена в соответствии с шагом градиентного спуска (зеленый в начале и красный в конце). По мере приближения градиентного спуска к минимуму стоимостной функции соответствующие параметры обеспечивают все более качественную подгонку к данным, причем оптимальная подгонка происходит

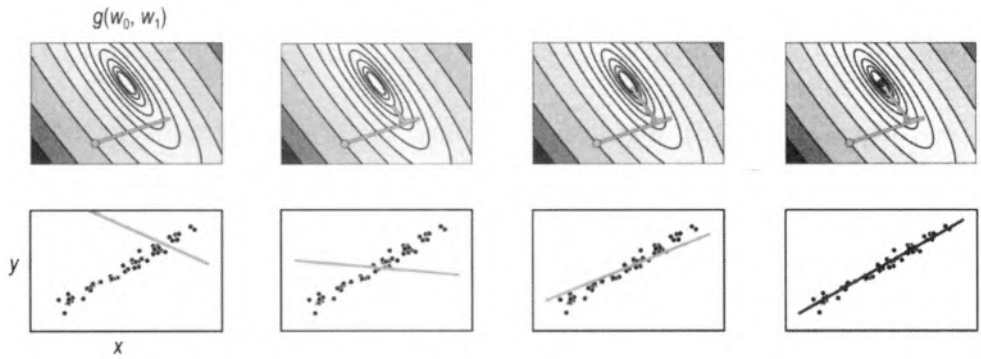


Рис. 5.4. Иллюстрация примера 5.2

в конце прогона в точке, наиболее близкой к минимизатору стоимости наименьших квадратов.

Всякий раз, когда мы используем метод локальной оптимизации, такой как градиентный спуск, мы должны правильно настроить параметр длины шага α (как описано ранее, например, в *разд. 3.5*). На рис. 5.5 представлен график истории стоимостной функции для двух значений длины шага: $\alpha = 0,01$ (штриховой линией фиолетового цвета) и $\alpha = 0,5$ (сплошной черной линией), которые мы в конечном итоге использовали для прогона, показанного на рис. 5.4. Он иллюстрирует причину, почему (в книгах по машинному обучению) параметр длины шага часто упоминается как *интенсивность усвоения* (скорость обучения), поскольку это значение действительно определяет, насколько быстро усваиваются правильные параметры нашей линейно-регрессионной модели (или любой модели машинного обучения в целом).

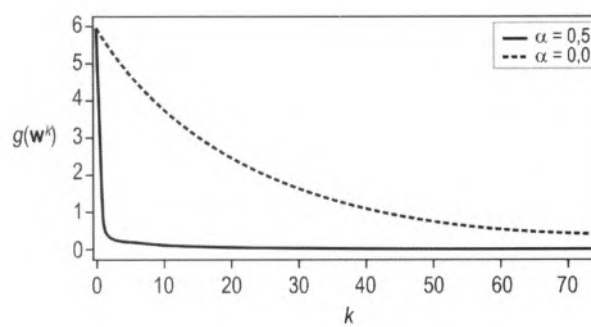


Рис. 5.5. Иллюстрация примера 5.2

5.2.4. Имплементация на Python

Во время имплементации стоимостной функции, такой как наименьшие квадраты, полезно рассуждать модульно, с целью уменьшения объема требуемой умственной "бухгалтерии" разбивая стоимость на несколько четко различимых компонентов.

Здесь мы разбиваем стоимостную функцию на две главные части: *модель*, представляющую собой линейную комбинацию входа и весов, и саму *стоимость* (т. е. квадратическую ошибку).

Выразим нашу (линейную) модель как функцию, достаточно достойную собственной нотации, как

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \dot{\mathbf{x}}_p^T \mathbf{w} \,. \tag{5.11}$$

Если бы мы вернулись назад и использовали эту новую форму записи моделирования, то могли бы переписать наши идеальные настройки весов в уравнении (5.7) как

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) \approx y_p \tag{5.12}$$

и точно так же нашу стоимостную функцию наименьших квадратов в уравнении (5.9) как

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) - y_p\right)^2 \,. \tag{5.13}$$

Этот вид простой деконструкции стоимости наименьших квадратов поддается организованной, модульной и легко расширяемой имплементации. Начиная с модели, обратите внимание на то, что хотя компактнее и удобнее *математически* записать линейную комбинацию $\dot{\mathbf{x}}_p^T \mathbf{w}$, прикрепив 1 поверх сырого входа $\mathbf{x}_p$, в ее *имплементировании* мы можем легче вычислить линейную комбинацию, выставив смещение и признаковые веса *отдельно*, как

$$\dot{\mathbf{x}}_p^T \mathbf{w} = w_0 + \dot{\mathbf{x}}_p^T \boldsymbol{\omega} \,, \tag{5.14}$$

где вектор $\boldsymbol{\omega}$ содержит все признаковые веса

$$\boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} . \tag{5.15}$$

Вспомните, что w_0 называется *смещением*, т. к. оно управляет тем, где наша линейная модель пересекает ось y, а w_1, w_2, ..., w_N называются *признаковыми весами*, потому что они касаются каждой отдельной размерности входа (которые на жаргоне машинного обучения называются *признаками*).

Используя эффективную операцию `np.dot` библиотеки NumPy[2], мы можем имплементировать линейную модель как

```
1| a = w[0] + np.dot(x_p.T,w[1:])
```

[2] В качестве общего правила всякий раз, когда доступны векторизованные имплементации, следует воздерживаться от имплементирования алгебраических выражений на Python поэлементно с использованием, например, явно задаваемых циклов `for`.

что соответствует правой части уравнения (5.14), где `w[0]` обозначает смещение w_0, а `w[1:]` — оставшиеся N признаковых весов в $\boldsymbol{\omega}$. Обернув это в Python-функцию, мы получаем имплементацию нашей линейной модели в виде

```
# вычислить линейную комбинацию входных точек
def model(x_p,w):
    # вычислить линейную комбинацию и вернуться
    a = w[0] + np.dot(x_p.T,w[1:])
    return a.T
```

которую мы можем затем использовать для формирования стоимостной функции, связанной с наименьшими квадратами.

```
# функция наименьших квадратов для линейной регрессии
def least_squares(w,x,y):
    # перебрать точки и вычислить стоимостной вклад из каждой пары вход-выход
    cost = 0
    for p in range(y.size):
        # получить p-ю пару вход-выход
        x_p = x[:,p][:,np.newaxis]
        y_p = y[p]

        ## прибавить к текущей стоимости
        cost += (model(x_p,w) - y_p)**2

    # вернуть среднюю ошибку наименьших квадратов
    return cost/float(y.size)
```

Обратите внимание, что здесь мы явно показываем *все* входы стоимостной функции, а не только $(N+1)\times 1$ весов $\mathbf{w}$, чья Python-переменная обозначается идентификатором `w`. Стоимость наименьших квадратов также принимает все входы (укладываемые поверх каждой точки) $\dot{\mathbf{x}}_p$, которые вместе мы обозначаем Python-переменной `x` размера $(N+1)\times 1$, а также все множество соответствующих выходов, которые мы обозначаем как переменную `y` размера $1\times P$.

Также обратите внимание на то, что указанное выше на самом деле является непосредственной имплементацией алгебраической формы стоимости в уравнении (5.13), где мы рассуждаем о стоимости как о сумме квадратических ошибок входа относительно соответствующего ему выхода в линейной модели `model`. Однако явные циклы `for` (включая операцию включения в список), написанные на Python, работают довольно медленно ввиду самой природы языка.

Бóльшую часть этой неэффективности легко обойти, заменив явные циклы `for` на численно эквивалентные операции, выполняемые с помощью операций библиотеки NumPy. Библиотека NumPy — это API для нескольких очень эффективных библиотек векторных/матричных манипуляций, написанных на языке C. Вообще говоря, при написании такой Python-функции, как эта, с интенсивным использованием функциональности NumPy каждый шаг вычисления — который ранее формировался последовательно в каждой точке данных — принято упаковывать вместе для

всего набора данных одновременно. Как результат, мы избавляемся от явного цикла `for` над каждой нашей точкой из всего множества P точек и делаем одинаковые вычисления для каждой точки одномоментно. Ниже мы приводим одну такую версию с интенсивным применением операций библиотеки NumPy, показанной ранее имплементации наименьших квадратов, которая работает гораздо эффективнее.

Обратите внимание, что при использовании этих функций входная переменная `x` (содержащая все множество из P входов) имеет размер $N \times P$, а соответствующий ей выход `y` имеет размер $1 \times P$. Здесь мы написали этот код, и в частности модельную функцию `model`, чтобы как можно ближе отразить соответствующую ей формулу.

```
# вычислить линейную комбинацию входных точек
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T

# имплементация стоимостной функции наименьших квадратов для линейной регрессии
def least_squares(w):
    # вычислить стоимость наименьших квадратов
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(y.size)
```

Обратите внимание на то, что для простоты вместо Python-функции `least_squares(w,x,y)` мы написали функцию `least_squares(w)`, где в первом случае мы явно перечисляем два других ее аргумента: входные данные `x` и выходные данные `y`. Это делается для нотационной простоты — мы делаем это в нашей математической нотации, а также обозначая нашу стоимость наименьших квадратов через $g(\mathbf{w})$ вместо $g(\mathbf{w}, \mathbf{x}, \mathbf{y})$, — в практическом смысле сгодится совершенно любой формат, поскольку autograd (*см. разд. П2.10*) будет правильно дифференцировать обе формы (т. к. по умолчанию он вычисляет градиент Python-функции только по отношению к ее первому входу). Мы будем использовать этот вид упрощенной Python-нотации и при введении будущих стоимостных функций машинного обучения.

Хотя большинству пользователей мы рекомендуем задействовать библиотеку автоматического дифференцирования autograd (*см. разд. 3.4*) для выполнения и градиентного спуска, и метода Ньютона на наших стоимостных функциях машинного обучения, здесь (поскольку указанная стоимостная функция является достаточно простой) можно "жестко закодировать" градиент формально, записав его "вручную" (используя правила производных, подробно описанные в *разд. П2.3*). При этом можно вычислить градиент стоимости наименьших квадратов в замкнутой форме как

$$\nabla g(\mathbf{w}) = \frac{2}{P}\sum_{p=1}^{P} \dot{\mathbf{x}}_p \left(\dot{\mathbf{x}}_p^T \mathbf{w} - y_p\right). \tag{5.16}$$

При выполнении метода Ньютона можно также вычислить значение стоимости наименьших квадратов вручную. Более того, поскольку стоимость является выпук-

лой квадратичной функцией, минимизировать ее полностью может всего *один-единственный ньютонов шаг*, и это решение принято называть *минимизированием стоимости наименьших квадратов посредством ее нормальных уравнений*. Система уравнений, решаемая в этом единственном ньютоновом шаге, эквивалентна *системе первого порядка* (*см. разд. 3.2*) для стоимостной функции наименьших квадратов:

$$\left(\sum_{p=1}^{P}\dot{\mathbf{x}}_p\dot{\mathbf{x}}_p^T\right)\mathbf{w}=\sum_{p=1}^{P}\dot{\mathbf{x}}_p y_p\ . \tag{5.17}$$

5.3. Наименьшие абсолютные отклонения

В этом разделе мы обсудим незначительное видоизменение в процедуре выведения стоимостной функции для наименьших квадратов, которое приводит к альтернативной стоимости для линейной регрессии, именуемой *наименьшими абсолютными отклонениями*. Эта альтернативная стоимостная функция является гораздо более устойчивой к выбросам в наборе данных, чем исходные наименьшие квадраты.

5.3.1. Подверженность наименьших квадратов к выбросам

Одним из недостатков использования квадратической ошибки в методе наименьших квадратов (в качестве меры, которую мы затем минимизируем для восстановления оптимальных параметров линейной регрессии) является то, что *возведение ошибки в квадрат увеличивает важность более крупных ошибок*. В частности, возведение в квадрат ошибки длины больше 1 делает эти значения значительно крупнее. Это вынуждает веса, усвоенные посредством наименьших квадратов, производить линейную подгонку, которая особенно сосредоточена на попытке минимизировать эти крупные ошибки, иногда вследствие выбросов в наборе данных. Другими словами, стоимость наименьших квадратов создает линейные модели, которые тяготеют к *переподгонке* к выбросам в наборе данных. Проиллюстрируем этот факт с помощью простого набора данных в примере 5.3.

Пример 5.3. Метод наименьших квадратов излишне прилегает к выбросам

В этом примере мы используем набор данных, показанный на графике рис. 5.6, который в основном может быть представлен надлежащей линейной моделью, за исключением одного-единственного выброса. Мы хотим показать, как стоимостная функция наименьших квадратов для линейной регрессии тяготеет к созданию переподогнанных линейных моделей, т. е. которые излишне прилегают к выбросам. Настроим параметры линейного регрессора на этот набор данных, минимизируя стоимость наименьших квадратов посредством градиентного спуска (*см. разд. 3.5*), и построим график соответствующей линейной модели поверх данных. Эта подгонка (показанная линией синего цвета) плохо вписывается в большинство точек данных, явно поднимаясь вверх с целью минимизировать крупную квадратическую ошибку в одиночной аномальной точке.

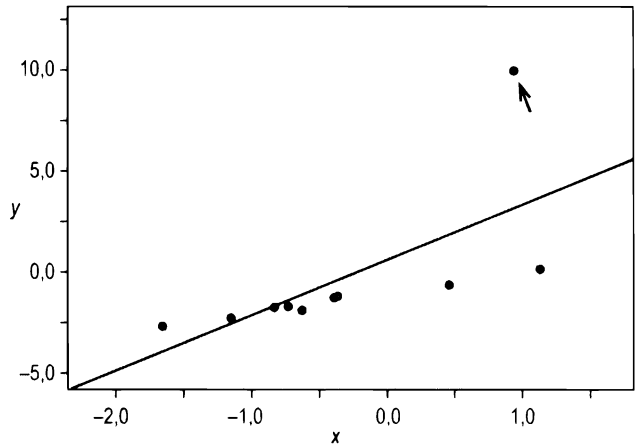


Рис. 5.6. Иллюстрация примера 5.3

5.3.2. Замена квадратической ошибки абсолютной ошибкой

Наше первоначальная процедура выведения стоимостной функции для наименьших квадратов в *разд. 5.2* была ориентирована на усвоение набора идеальных весов с тем, чтобы у нас было

$$\dot{\mathbf{x}}_p^T \mathbf{w} \approx y_p, \quad p = 1, ..., P \tag{5.18}$$

для набора данных из P точек $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$. Затем мы возвели в квадрат разницу обеих частей каждой желаемой аппроксимации

$$g_p(\mathbf{w}) = \left(\dot{\mathbf{x}}_p^T \mathbf{w} - y_p\right)^2, \quad p = 1, ..., P \tag{5.19}$$

и взяли среднее значение этих P квадратических ошибок, чтобы сформировать полную стоимостную функцию наименьших квадратов.

В качестве альтернативы использованию *квадратической ошибки* для нашей точечной стоимости в уравнении (5.19) мы можем измерить *абсолютную ошибку* для каждой желаемой аппроксимации

$$g_p(\mathbf{w}) = \left|\dot{\mathbf{x}}_p^T \mathbf{w} - y_p\right|, \quad p = 1, ..., P. \tag{5.20}$$

Используя абсолютную ошибку вместо квадратической версии, мы по-прежнему рассматриваем отрицательные и положительные ошибки одинаково, но не преувеличиваем важность крупных ошибок, превышающих 1. Беря среднее значение этих абсолютных ошибок точечных стоимостей, мы получаем "двоюродную сестру" стоимости наименьших квадратов — стоимостную функцию так называемых *наименьших абсолютных отклонений*

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} g_p(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left|\dot{\mathbf{x}}_p^T \mathbf{w} - y_p\right|. \tag{5.21}$$

Единственная цена, которую мы платим за задействование абсолютной ошибки вместо квадратической ошибки, является технической: хотя эта стоимостная функция также всегда является выпуклой независимо от входного набора данных, поскольку ее вторая производная (почти везде) равна нулю, мы можем использовать исключительно методы нулевого и первого порядка, чтобы правильно ее минимизировать (но не методы второго порядка).

Пример 5.4. Наименьшие квадраты против наименьших абсолютных отклонений

На рис. 5.7 на основе набора данных из примера 5.3 представлены результаты настройки линейной модели путем минимизирования стоимостной функции наименьших квадратов в сравнении со стоимостной функцией наименьших абсолютных отклонений. В обоих случаях мы выполняем градиентный спуск для одинакового числа шагов и используем один тот же вариант параметра длины шага. Истории стоимостных функций для обоих прогонов показаны на рис. 5.7, *справа*: прогоны с использованием наименьших квадратов черной сплошной линией, прогоны с использованием наименьших абсолютных отклонений — штриховой линией. Рассматривая истории стоимостных функций, мы видим, что значение стоимостной функции наименьших абсолютных отклонений значительно ниже, чем такое же значение наименьших квадратов. Уже одно это свидетельствует о том, что первый вариант обеспечивает значительно более качественную подгонку, чем наименьшие квадраты.

Это преимущество также можно увидеть на рис. 5.7, *слева* на графике оптимально подогнанной прямой. Подгонка наименьшими квадратами показана сплошной линией, подгонка наименьшими абсолютными отклонениями — штриховой. Последнее значительно лучше, т. к. не преувеличивает более крупную ошибку, которую дает один-единственный выброс.

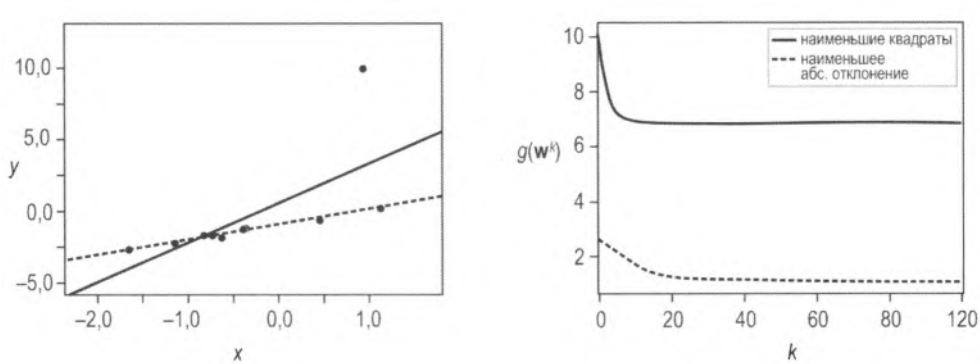


Рис. 5.7. Иллюстрация примера 5.4

5.4. Метрики качества регрессии

В этом кратком разделе мы опишем процедуру предсказания с использованием натренированной регрессионной модели и сопроводим наши объяснения простыми метриками для суждения о качестве такой модели.

5.4.1. Выполнение предсказаний с использованием натренированной модели

Если мы обозначим оптимальное множество весов, найденных путем минимизирования регрессионной стоимостной функции по $\mathbf{w}^\star$, то наша полностью натренированная линейная модель может быть записана как

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = \dot{\mathbf{x}}_p^T \mathbf{w}^\star = w_0^\star + x_1 w_1^\star + x_2 w_2^\star + \ldots + x_N w_N^\star . \tag{5.22}$$

Независимо от того, как мы определяем оптимальные параметры $\mathbf{w}^\star$, путем минимизирования регрессионной стоимости, такой как наименьшие квадраты или наименьшие абсолютные отклонения, мы делаем предсказания, задействуя нашу линейную модель одинаково. То есть, имея вход $\mathbf{x}_0$ (из нашего тренировочного набора данных или совершенно новых входных данных), мы предсказываем выход y_0 путем передачи их вместе с нашими натренированными весами в нашу модель:

$$\text{model}\left(\mathbf{x}_0, \mathbf{w}^\star\right) = y_0 . \tag{5.23}$$

Это наглядно проиллюстрировано на прототипическом линейно-регрессионном наборе данных на рис. 5.8.

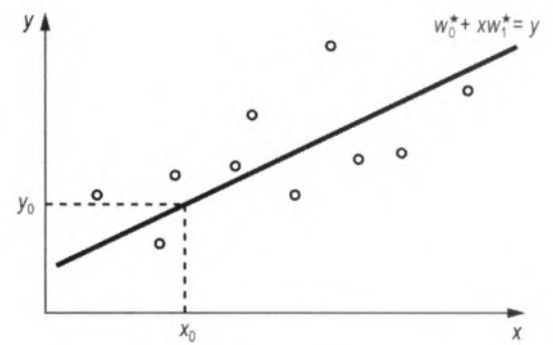


Рис. 5.8. После отыскания оптимальных параметров $w_0^\star$ и $w_1^\star$ регрессионной прямой посредством минимизирования соответствующей стоимостной функции их можно использовать для предсказания выходного значения для любого входа x_0 путем подстановки его в уравнение (5.22). Здесь $N = 1$

5.4.2. Суждение о качестве натренированной модели

После успешной минимизации стоимостной функции линейной регрессии легко определить качество нашей регрессионной модели: мы просто вычисляем стоимостную функцию, используя наши оптимальные веса. Например, мы можем оценить качество натренированной модели посредством метода наименьших квадратов, который совершенно естественно использовать, когда мы задействуем эту функцию потерь в процессе тренировки. Для этого подставим наши модельные параметры

вместе с данными в функцию наименьших квадратов и будем работать с так называемой *среднеквадратической ошибкой* (mean squared error, MSE)

$$\text{MSE} = \frac{1}{P}\sum_{p=1}^{P}\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}^\star\right) - y_p\right)^2. \tag{5.24}$$

Название этой метрики качества регрессии верно описывает точность вычисления стоимости наименьших квадратов, т. е. среднеарифметическую (или просто среднюю) квадратическую ошибку. Для того чтобы уменьшить влияние выбросов и других крупных значений, часто берут *квадратный корень* из этого значения. Эта метрика так и называется — *корень из среднеквадратической ошибки* (root mean squared error, RMSE).

Точно так же для определения качества нашей натренированной модели мы можем задействовать и метод наименьших абсолютных отклонений. Если подставить наши модельные параметры вместе с данными в эту функцию, то будут вычислены *средние абсолютные отклонения* (mean absolute deviations, MAD), которые являются как раз тем, что и вычисляет эта стоимостная функция

$$\text{MAD} = \frac{1}{P}\sum_{p=1}^{P}\left|\text{model}\left(\mathbf{x}_p, \mathbf{w}^\star\right) - y_p\right|. \tag{5.25}$$

Метрики (5.24) и (5.25) различаются именно тем, чем различаются соответствующие им стоимостные функции (например, мера MSE гораздо более чувствительна к выбросам). В общем случае чем *меньше* можно сделать эти метрики качества (путем надлежащей настройки модельных весов), тем *лучше* качество соответствующей натренированной модели, и наоборот. Однако человеческий взгляд на "хорошую" или "отличную" работоспособность модели может зависеть от личных предпочтений индивидуума, профессионального или институционального ориентира или какой-либо другой зависящей от задачи заинтересованности.

Пример 5.5. Предсказание цен на жилье

Как описано в примере 1.4, линейная регрессия имеет мириады бизнес-приложений. Предсказание цены на некий товар является особенно популярным приложением, а набор данных Boston Housing со сведениями о ценах на жилье в Бостоне [16] является легкодоступным примером такой задачи. Этот набор данных состоит из набора базовых статистик (признаков) и соответствующих цен (в долларах) для $P = 506$ домов в Бостоне (США). Тринадцатимерные ($N = 13$) входные признаки включают уровень преступности на душу населения в разбивке по городам (признак 1), среднее число комнат на одно жилище (признак 6), взвешенные расстояния до пяти бостонских центров занятости (признак 8) и процент местного населения, считающегося "низшим классом", обозначаемый LSTAT (признак 13). На основе указанного набора данных можно легко достичь разумных метрик RMSE и MAD, примерно равных соответственно 4,7 и 3,1 (см. упражнение 5.9). Позже мы исследуем критически важные признаки этого набора в процессе *отбора признаков* в примерах 9.6 и 9.11.

Пример 5.6. Предсказание автомобильного пробега в милях на галлон израсходованного топлива

Как подробно описано ранее в примере 1.4, линейная регрессия имеет массу промышленных применений, связанных с точным предсказанием поведения конкретных систем. Набор данных об автомобильном пробеге в милях на галлон израсходованного топлива (Autombile Miles-per-Gallon, Auto-MPG) [17] является популярным набором данных, используемым для таких целей, и состоит из набора базовых данных о $P = 398$ автомобилях. $N = 6$ входных признаков призваны использоваться для предсказания автомобильного пробега в милях на галлон израсходованного топлива каждого автомобиля. Входные признаки этого набора данных включают число цилиндров в блоке автомобильного двигателя (признак 1), общий объем двигателя (признак 2), мощность автомобильного двигателя в лошадиных силах (признак 3), массу автомобиля (признак 4), разгон автомобиля, измеряемый в секундах, затрачиваемых на ускорение до контрольной скорости с момента остановки (признак 5), и год выпуска автомобиля (признак 6)[3]. На основе указанного набора данных можно достичь разумных метрик RMSE и MAD, примерно равных соответственно 3,3 и 2,5 мили на галлон (см. упражнение 5.9). Чуть позже мы исследуем критически важные признаки этих данных при *отборе признаков* в упражнении 9.10.

5.5. Взвешенная регрессия

Поскольку регрессионные стоимостные функции можно разложить по отдельным точкам данных, то, как мы увидим в этом разделе, существует возможность эти точки взвешивать для того, чтобы преувеличить или преуменьшать их важность для регрессионной модели. Эта практика называется *взвешенной регрессией.*

5.5.1. Работа с дубликатами

Представьте себе, что у нас есть линейно-регрессионный набор данных, содержащий несколько копий одной и той же точки, генерируемых не по ошибке, а, например, путем необходимой квантизации входных признаков (или разбиения их на корзины), для того, чтобы облегчить анализ или моделирование данных с периодическим участием человека. Излишне говорить, что в подобной ситуации "дублирующие" точки данных не должны быть выброшены.

Пример 5.7. Квантизация входных признаков может создавать дубликаты точек

На рис. 5.9 показан сырой набор данных из современной реконструкции знаменитого эксперимента Галилея с рампой, в котором для оценивания количественного влияния гравитации неоднократно скатывают шар вниз по рампе, чтобы определить соотношение между расстоянием и временем, которое требуется объекту, чтобы

[3] Последний признак исходного набора данных (под названием "происхождение") был удален, поскольку не удалось отыскать его содержательное описание.

упасть на землю. Этот набор данных состоит из нескольких испытаний одного и того же эксперимента, где числовое значение каждого выходного результата было округлено до двух знаков после запятой. Выполнение этого естественного числового округления (иногда именуемого *квантизацией*) производит многочисленные повторяющиеся точки данных, которые точками отмечены на рисунке, но предварительно отмасштабированы. Чем больше радиус графической точки, тем больше повторяющихся точек данных она представляет.

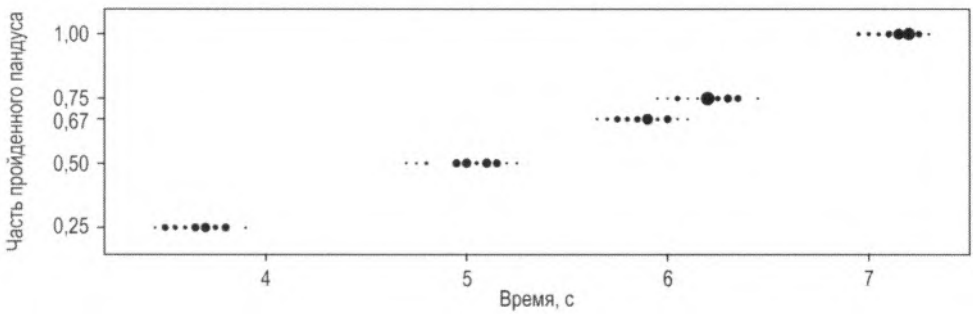


Рис. 5.9. Иллюстрация примера 5.7

Давайте теперь рассмотрим, что происходит с регрессионной стоимостной функцией (например, наименьшими квадратами), когда набор данных содержит повторяющиеся точки данных. В частности, мы исходим из того, что в наших данных есть β_p версий пары "вход — выход" ($\mathbf{x}_p, y_p$). Для регрессионных наборов данных, которые мы видели до сих пор (исключая тот, который показан на рис. 5.9), мы всегда имели $\beta_p = 1$ для всех $p = 1, ..., P$. Используя нашу модельную нотацию для представления нашей линейной модели (например, *см. разд. 5.4.1*), мы можем записать сумму всех точечных квадратических ошибок как

$$\begin{aligned}
&\underbrace{\left(\text{model}(\mathbf{x}_1, \mathbf{w}) - y_1\right)^2 + ... + \left(\text{model}(\mathbf{x}_1, \mathbf{w}) - y_1\right)^2}_{\beta_1} + \\
&+ \underbrace{\left(\text{model}(\mathbf{x}_2, \mathbf{w}) - y_2\right)^2 + ... + \left(\text{model}(\mathbf{x}_2, \mathbf{w}) - y_2\right)^2}_{\beta_2} + \\
&\vdots \\
&+ \underbrace{\left(\text{model}(\mathbf{x}_P, \mathbf{w}) - y_P\right)^2 + ... + \left(\text{model}(\mathbf{x}_P, \mathbf{w}) - y_P\right)^2}_{\beta_P}.
\end{aligned} \tag{5.26}$$

Естественное группирование в уравнении (5.26) помогает нам записать общую стоимостную функцию наименьших квадратов как

$$g(\mathbf{w}) = \frac{1}{\beta_1 + \beta_2 + ... + \beta_P} \sum_{p=1}^{P} \beta_p \left(\text{model}(\mathbf{x}_p, \mathbf{w}) - y_p\right)^2. \tag{5.27}$$

Как мы здесь видим, стоимостная функция наименьших квадратов естественным образом сворачивается во взвешенную версию самой себя в том смысле, что мы

можем сочетать слагаемые так, чтобы повторяющаяся точка в наборе данных была представлена в стоимостной функции одним взвешенным слагаемым. Так как веса β_1, β_2, ..., β_P фиксированы для любого рассматриваемого набора данных, мы можем минимизировать взвешенную регрессионную стоимость точно так же, как и любую другую (настроив лишь **w**). Наконец, заметим, что если установить $\beta_p = 1$ (для всех p) в уравнении (5.27), то мы вернем исходную (невзвешенную) стоимостную функцию наименьших квадратов из уравнения (5.13).

5.5.2. Взвешивание точек по уверенности

Взвешенная регрессия может быть также задействована, когда мы хотим взвесить каждую точку на основе нашей уверенности в *достоверности* каждой точки данных. Например, если бы наш набор данных состоял из двух пакетов: один пакет из надежного источника, а другой — из менее надежного источника (где некоторые точки данных могут быть шумными или ошибочными), то в нашей окончательной регрессии мы хотели бы придать больший вес точкам данных из надежного источника. Это можно сделать очень легко, используя парадигму взвешенной регрессии, введенную ранее, только теперь мы устанавливаем веса β_1, β_2, ..., β_P сами, основываясь на нашей уверенности в каждой точке. Если мы считаем, что точка является очень надежной, то можем установить соответствующий ей вес β_p высоким, и наоборот. Обратите внимание, что в крайнем случае весовое значение $\beta_p = 0$ эффективно удаляет соответствующую точку данных из стоимостной функции, подразумевая, что мы вообще не доверяем этой точке.

Пример 5.8. Корректирование веса одной точки данных для отражения уверенности

На рис. 5.10 показано, как корректирование веса, связанного с одной точкой данных, влияет на окончательную усвоенную модель в обстановке взвешенной линейной регрессии. Набор данных для регрессии, показанный на этом рисунке, содержит точку данных — кружок красного цвета, диаметр которого изменяется пропорционально весу этой самой точки. По мере того как вес (который можно интерпретировать как "уверенность") увеличивается слева направо, регрессор все больше и больше фокусируется на представлении красной точки. Если мы доста-

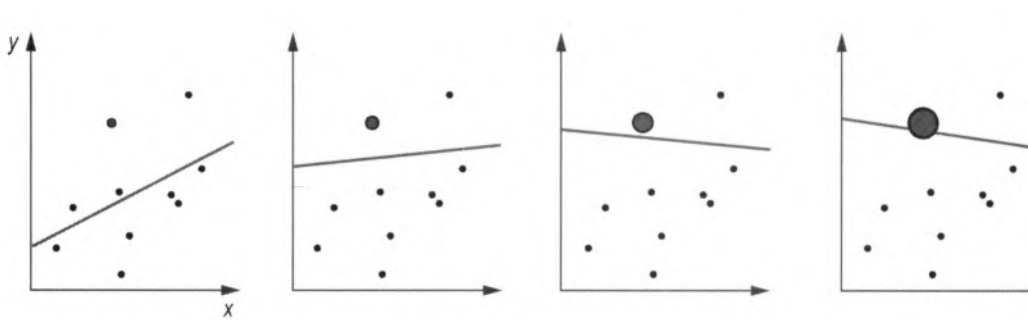


Рис. 5.10. Иллюстрация примера 5.8

точно увеличим ее вес, то полностью усвоенная регрессионная модель естественным образом начнет приспосабливаться к этой единственной точке данных, игнорируя все остальные точки, как показано на самом правом графике рис. 5.10.

5.6. Многовыходная регрессия

До сих пор мы исходили из того, что точки данных для линейной регрессии состоят из *векторнозначных* входов и *скалярнозначных* выходов. Другими словами, точка данных прототипической регрессии принимает форму пары "вход — выход" $(\mathbf{x}_p, y_p)$, где вход $\mathbf{x}_p$ — это N-мерный вектор, а выход y_p — скаляр. Хотя эта конфигурация охватывает подавляющее большинство регрессионных случаев, с которыми можно столкнуться на практике, существует возможность выполнить (линейную) регрессию, где *и вход, и выход* являются векторнозначными. Такая регрессия часто называется *многовыходной регрессией*, которую мы сейчас и обсудим.

5.6.1. Нотация и моделирование

Предположим, что наш регрессионный набор данных состоит из P пар "входов — выходов"

$$(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), ..., (\mathbf{x}_P, \mathbf{y}_P), \tag{5.28}$$

где каждый вход $\mathbf{x}_p$ — N-мерный, а каждый выход $\mathbf{y}_p$ — C-мерный. Хотя, в принципе, мы можем рассматривать $\mathbf{y}_p$ как вектор-столбец размером $C \times 1$, чтобы последующие формулы выглядели аналогично тому, что мы уже видели в скалярном случае, мы будем трактовать вход как *столбцовый* $N \times 1$-вектор, а выход — как *строковый* $1 \times C$-вектор, в виде[4]

$$\mathbf{x}_p = \begin{bmatrix} x_{1,p} \\ x_{2,p} \\ \vdots \\ x_{n,p} \end{bmatrix} \qquad \mathbf{y}_p = \begin{bmatrix} y_{0,p} & y_{1,p} & \cdots & y_{C-1,p} \end{bmatrix}. \tag{5.29}$$

Если мы примем за основу, что *линейная* связь имеет место между входом $\mathbf{x}_p$ и *только* c-й размерностью выхода $y_{c,p}$, то мы вернемся именно к тому типу регрессионного каркаса, который видели до сих пор, и можем написать

$$\dot{\mathbf{x}}_p^T \mathbf{w}_c \approx y_{c,p}, \quad p = 1, ..., P, \tag{5.30}$$

---

[4] Обратите внимание, что, в отличие от входа, мы индексируем выход, начиная с 0. Мы делаем это потому, что в конечном итоге мы будем укладывать 1 поверх каждого входа $\mathbf{x}_p$ (как мы делали со стандартной регрессией в *разд. 5.2*), и эта запись будет иметь нулевой индекс нашего входа.

где $\mathbf{w}_c$ — это набор весов

$$\mathbf{w}_c = \begin{bmatrix} w_{0,c} \\ w_{1,c} \\ w_{2,c} \\ \vdots \\ w_{N,c} \end{bmatrix}, \tag{5.31}$$

а $\dot{\mathbf{x}}_p^T$ — это вектор, образованный добавлением 1 в качестве первого элемента вектора $\mathbf{x}_p$. Если мы далее примем, что соблюдается линейная связь между входом и *всеми* C элементами выхода, то мы можем поместить каждый весовой вектор $\mathbf{w}_c$ в c-й столбец весовой $(N+1)\times C$ *-матрицы* $\mathbf{W}$ как

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,c} & \cdots & w_{0,C-1} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,c} & \cdots & w_{1,C-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,c} & \cdots & w_{2,C-1} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ w_{N,0} & w_{N,1} & \cdots & w_{N,c} & \cdots & w_{N,C-1} \end{bmatrix} \tag{5.32}$$

и записать все множество C линейных моделей посредством векторно-матричного произведения

$$\dot{\mathbf{x}}_p^T\mathbf{W} = \begin{bmatrix} \dot{\mathbf{x}}_p^T\mathbf{w}_0 & \dot{\mathbf{x}}_p^T\mathbf{w}_1 & \cdots & \dot{\mathbf{x}}_p^T\mathbf{w}_{C-1} \end{bmatrix}. \tag{5.33}$$

Это позволяет нам очень компактно записать все множество C линейных связей как

$$\dot{\mathbf{x}}_p^T\mathbf{W} \approx y_p, \quad p = 1, ..., P. \tag{5.34}$$

5.6.2. Стоимостные функции

Мыслительный процесс, связанный с выведением регрессионной стоимостной функции для случая многовыходной регрессии, почти точно отражает случай скалярнозначного выхода, рассмотренный в *разд. 5.2* и *5.3*. Например, для того чтобы вывести стоимостную функцию для наименьших квадратов, мы начинаем (так же как и в *разд. 5.2*), беря разность обеих частей уравнения (5.34). Однако ошибка, связанная с p-й точкой, записываемая как $\dot{\mathbf{x}}_p^T\mathbf{W} - y_p$, теперь является вектором C значений. Для того чтобы возвести эту ошибку в квадрат, мы должны задействовать квадратичную *векторную* ℓ_2-норму (*см. разд. П3.5*, если вы не знакомы с этой векторной нормой). Стоимостная функция наименьших квадратов в этом случае представляет собой среднеквадратичную ℓ_2-норму ошибки в каждой точке, записываемую как

$$g(\mathbf{W}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\dot{\mathbf{x}}_p^T\mathbf{W} - \mathbf{y}_p\right\|_2^2 = \frac{1}{P}\sum_{p=1}^{P}\sum_{c=0}^{C-1}\left(\dot{\mathbf{x}}_p^T\mathbf{w}_c - y_{c,p}\right)^2. \tag{5.35}$$

Обратите внимание, что при $C = 1$ формула сводится к первоначальной стоимости наименьших квадратов, которую мы видели в *разд. 5.2*.

Стоимость наименьших абсолютных отклонений (которая измеряет абсолютную величину каждой ошибки в противоположность к ее квадрату) для нашего настоящего случая принимает аналогичную форму:

$$g(\mathbf{W}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\dot{\mathbf{x}}_p^T\mathbf{W} - \mathbf{y}_p\right\|_1 = \frac{1}{P}\sum_{p=1}^{P}\sum_{c=0}^{C-1}\left|\dot{\mathbf{x}}_p^T\mathbf{w}_c - y_{c,p}\right|, \tag{5.36}$$

где $\|\cdot\|_1$ — это векторная ℓ_1-норма, обобщение функции абсолютного значения для векторов (*см. разд. П3.5.1*, если вы не знакомы с этой векторной нормой).

Как и их скалярнозначные версии, эти стоимостные функции всегда являются выпуклыми независимо от используемого набора данных. Их также можно разложить по весам $\mathbf{w}_c$, связанным с каждой выходной размерностью. Например, мы можем переписать правую часть стоимости наименьших абсолютных отклонений в уравнении (5.36), поменяв местами слагаемые P и C и получив

$$g(\mathbf{W}) = \sum_{c=0}^{C-1}\left(\frac{1}{P}\sum_{p=1}^{P}\left|\dot{\mathbf{x}}_p^T\mathbf{w}_c - y_{c,p}\right|\right) = \sum_{c=0}^{C-1} g_c(\mathbf{w}_c), \tag{5.37}$$

где $g(\mathbf{w}_c) = \frac{1}{P}\sum_{p=1}^{P}\left|\dot{\mathbf{x}}_p^T\mathbf{w}_c - y_{c,p}\right|$. Так как веса из каждых C подзадач не взаимодействуют, мы можем минимизировать каждую g_c для оптимальной настройки $\mathbf{w}_c$ независимо, а затем взять их сумму, чтобы сформировать полную стоимостную функцию g.

Пример 5.9. Подгонка линейной модели к многовыходному регрессионному набору данных

На рис. 5.11 приведен пример многовыходной линейной регрессии с использованием игрушечного набора данных с входной размерностью $N = 2$ и выходной размерностью $C = 2$. Мы построили график входа и *одного* выходного значения в каждой из двух частей рисунка.

Мы настраиваем параметры соответствующей линейной модели, минимизируя стоимость наименьших квадратов посредством градиентного спуска, и иллюстри-

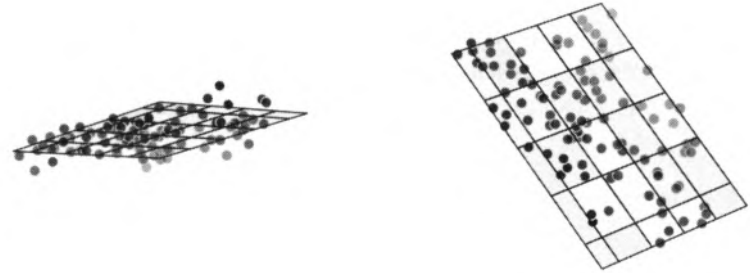


Рис. 5.11. Иллюстрация примера 5.9

руем полностью натренированную модель путем вычисления тонкой решетки точек во входном участке набора данных.

5.6.3. Имплементация на Python

Поскольку Python и NumPy имеют очень гибкий синтаксис, мы можем имплементировать линейную модель

$$\text{model}\left(\mathbf{x}, \mathbf{W}\right) = \dot{\mathbf{x}}_p^T\mathbf{W} \tag{5.38}$$

точно так же, как мы сделали это в случае скалярного выхода в *разд. 5.2.4*. В имплементации этой линейной комбинации нам не нужно формировать скорректированный вход $\dot{\mathbf{x}}_p$ (прикрепляя 1 поверх сырого входа $\mathbf{x}_p$), и мы можем легче вычислить линейную комбинацию, выставив смещения как

$$\dot{\mathbf{x}}_p^T\mathbf{W} = \mathbf{b} + \dot{\mathbf{x}}_p^T\boldsymbol{\mathcal{W}}, \tag{5.39}$$

где смещение $\mathbf{b}$ и признаковые веса $\boldsymbol{\mathcal{W}}$ обозначены как

$$\mathbf{b} = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & \cdots & w_{0,C-1} \end{bmatrix} \quad \boldsymbol{\mathcal{W}} = \begin{bmatrix} w_{0,1} & w_{0,2} & \cdots & w_{0,C-1} \\ w_{1,1} & w_{1,2} & \cdots & w_{1,C-1} \\ w_{2,1} & w_{,2} & \cdots & w_{2,C-1} \\ \vdots & \vdots & \vdots & \vdots \\ w_{N,1} & w_{N,2} & \cdots & w_{N,C-1} \end{bmatrix}. \tag{5.40}$$

Эта нотация используется для того, чтобы соответствовать Python-операции нарезки (как показано в приведенной ниже имплементации), которую мы имплементируем на Python следующим образом:

```
a = w[0] + np.dot(x_p.T,w[1:])
```

То есть `b = w[0]` обозначает смещение, а `W = w[1:]` — оставшиеся признаковые веса. Еще одна причина имплементировать таким образом состоит в том, что конкретная линейная комбинация $\mathbf{x}_p^T\boldsymbol{\mathcal{W}}$ (которая имплементировала модель, используя метод `np.dot` как `np.dot(x_p.T,w[1:])` в приведенном ниже коде) является особенно эффективной, поскольку Python-операция `np.dot` намного эффективнее, чем построение линейной комбинации на Python с помощью явного цикла `for`.

Стоимостные функции для многовыходной регрессии могут быть имплементированы в Python точно так же, как мы видели ранее. Например, наша линейная модель и стоимость наименьших квадратов могут быть записаны так:

```
1| # линейная модель
2| def model(x,w):
3|     a = w[0] + np.dot(x.T,w[1:])
4|     return a.T
5|
```

```
# стоимость наименьших квадратов
def least_squares(w):
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(np.size(y))
```

Обратите внимание, что поскольку любая стоимость для многовыходной регрессии принимает *матрицу* параметров, то при использовании библиотеки autograd как части вашего процесса оптимизации бывает очень удобно сначала *сгладить* выбранную вами стоимость (как описано в *разд. П2.10.3*) перед минимизацией. Это позволяет избежать необходимости явно перебирать веса в вашей процедуре локальной оптимизации, позволяя напрямую задействовать без изменений базовые Python-имплементации, например градиентный спуск (*см. разд. 3.5.4*) и метод Ньютона (*см. разд. 4.3.6*).

5.7. Заключение

Мы начали *часть II* этой книги с описания линейной регрессии, простейшей из наших задач обучения с учителем.

Если быть точнее, то мы начали в *разд. 5.2* с введения важной нотации, формальной линейной модели, а также стоимостной функции *наименьших квадратов* для регрессии. В *разд. 5.3* мы ввели стоимость *наименьших абсолютных отклонений*, которая гораздо менее чувствительна к выбросам, но при этом не является дважды дифференцируемой (таким образом, методы второго порядка неприменимы к ее минимизации непосредственно). Уже описав методы математической оптимизации в *части I*, мы бегло разобрались с минимизацией этих стоимостных функций для случая линейной регрессии. Далее в *разд. 5.4* мы рассмотрели различные метрики для количественной оценки качества натренированной регрессионной модели. Затем в *разд. 5.5* мы описали *взвешенную регрессию*, за которой в *разд. 5.6* последовало обсуждение многовыходной регрессии. Наконец, в *разд. 5.6* мы рассмотрели различные метрики, предназначенные для квантификации качества натренированной регрессионной модели.

5.8. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:

github.com/jermwatt/machine_learning_refined.

5.1. Подгонка регрессионной прямой к данным о студенческой кредитной задолженности

Выполнить подгонку линейной модели к набору данных о долговой нагрузке студентов США (см. рис. 1.8), минимизируя связанную с ней линейно-регрессионную задачу методом наименьших квадратов с использованием одного-единственного ньютонова шага (также именуемую решением нормальных уравнений). Если этот линейный тренд продолжится, то какова будет суммарная студенческая кредитная задолженность в 2030 году?

5.2. Закон Клейбера и линейная регрессия

После сбора значительного объема данных, сравнивающих массу тела и скорость метаболизма (меру расхода энергии в состоянии покоя) различных животных, и построения графика биолог начала XX века Макс Клейбер (Max Kleiber) отметил интересную связь между этими двумя величинами. Обозначив через x_p и y_p соответственно массу тела (кг) и скорость метаболизма (кДж/сут) данного животного, трактуя массу тела как входной признак, Клейбер отметил (при визуальном осмотре), что натуральные логарифмы этих двух значений линейно связаны, т. е.

$$w_0 + \log\left(x_p\right) w_1 \approx \log\left(y_p\right). \tag{5.41}$$

На рис. 1.9 показана крупная коллекция преобразованных точек данных

$$\left\{\left(\log\left(x_p\right), \log\left(y_p\right)\right)\right\}_{p=1}^{P}, \tag{5.42}$$

каждый элемент которой изображает животное, начиная от маленькой чернобородой колибри в левом нижнем углу и заканчивая крупным моржом в правом верхнем.

А) Выполнить подгонку линейной модели к данным, показанным на рис. 1.9.

Б) Использовать оптимальные параметры, найденные в части (А), вместе со свойствами логарифмической функции, чтобы записать нелинейную связь между массой тела x и скоростью метаболизма y.

В) Использовать вашу подогнанную прямую для того, чтобы определить, сколько калорий требуется животному массой 10 кг (обратите внимание, что каждая калория эквивалентна 4,18 Дж).

5.3. Стоимостная функция наименьших квадратов и один ньютонов шаг

Как уже упоминалось в *разд. 5.2.3*, один-единственный ньютонов шаг может полностью минимизировать стоимость наименьших квадратов для линейной регрессии. Применить один ньютонов шаг, чтобы полностью минимизировать стоимость наименьших квадратов над набором данных, показанным на рис. 5.12. Этот набор данных примерно лежит на гиперплоскости, поэтому подгонка, обеспечиваемая

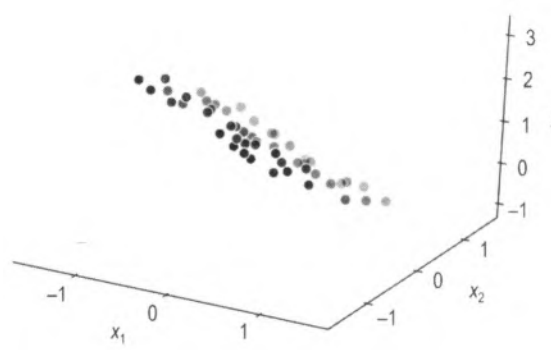


Рис. 5.12. Иллюстрация к упражнению 5.3

полностью минимизированной стоимостью наименьших квадратов, должна очень хорошо вписываться. Использовать график истории стоимостной функции, чтобы проверить надлежащую настройку параметров линейной модели.

5.4. Решение нормальных уравнений

Как обсуждалось в *разд. 5.2.4*, стоимостная функция наименьших квадратов для линейной регрессии может быть полностью минимизирована с помощью одного шага метода Ньютона (*см. разд. 4.3*) с помощью соответствующей системы уравнений (5.17), также именуемой *нормальными уравнениями*. В каких обстоятельствах, по вашему мнению, этот метод решения не является отличным методом для минимизирования стоимости наименьших квадратов? Почему? Подсказка: *см. разд. 4.4.2*.

5.5. Константа Липшица для стоимости наименьших квадратов

Вычислить константу Липшица (*см. разд. П1.6.4*) стоимостной функции наименьших квадратов.

5.6. Сравнить стоимости наименьших квадратов и наименьших абсолютных отклонений

Повторить эксперимент, описанный в примере 5.4. Вам нужно будет имплементировать стоимость наименьших абсолютных отклонений, что можно сделать аналогично имплементации наименьших квадратов из *разд. 5.2.4*.

5.7. Эмпирически подтвердить выпуклость для игрушечного набора данных

Эмпирически подтвердить, что стоимостная функция наименьших абсолютных отклонений является выпуклой для набора данных, показанного в *разд. 5.3*.

5.8. Стоимость наименьших абсолютных отклонений является выпуклой

Доказать, что стоимость наименьших абсолютных отклонений является выпуклой, используя приведенное ниже определение выпуклости нулевого порядка.

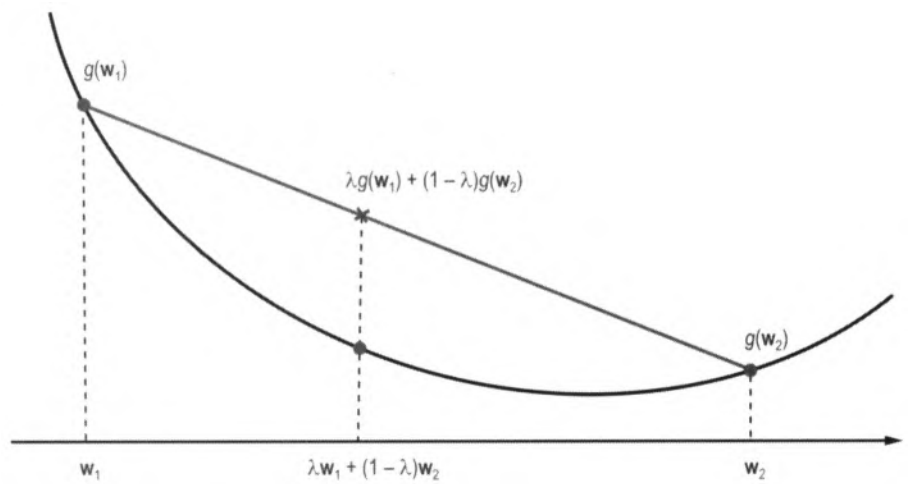


Рис. 5.13. Иллюстрация к упражнению 5.8

Неограниченная функция g является выпуклой, если и только если любой отрезок прямой, соединяющий две точки на графике g, лежит *выше* его графика. На рис. 5.13 проиллюстрировано это определение выпуклой функции.

Утверждая этот геометрический факт алгебраически, g является выпуклой, если и только если для всех $\mathbf{w}_1$ и $\mathbf{w}_2$ в области определения g и всех $\lambda \in [0; 1]$, мы имеем:

$$g(\lambda \mathbf{w}_1 + (1-\lambda)\mathbf{w}_2) \le \lambda g(\mathbf{w}_1) + (1-\lambda) g(\mathbf{w}_2). \quad (5.43)$$

5.9. Предсказание цен на жилье и автомобильного пробега в милях на галлон израсходованного топлива

Проверить метрики качества, приведенные в примерах 5.5 и 5.6 для наборов данных Boston Housing с данными о ценах на жилье в Бостоне и Automobile Miles-per-Gallon с данными об автомобильном пробеге в милях на галлон израсходованного топлива. Вследствие больших вариаций входных значений в этих наборах данных вы должны *стандартно нормализовать* входные признаки каждого из них (как подробно описано в *разд. 9.3*) перед оптимизацией.

5.10. Ненадлежащая настройка и взвешенная регрессия

Предположим, что кто-то предложил настроить точечные веса β_1, ..., β_P путем минимизирования взвешенной стоимости наименьших квадратов в уравнении (5.27) относительно и β_1, ..., β_P, и $\mathbf{w}$ весов линейной модели, поддерживая при этом все β_1, ..., β_P неотрицательными (отсюда наименьшее значение, взятое стоимостной функцией, равно нулю). Предположим, что вы способны выполнить оптимизацию до полного ее завершения. Что может пойти не так с точки зрения эффективности окончательной модели на общем наборе данных? Свои идеи объяснить с помощью рисунка.

5.11. Многовыходная регрессия

Повторить эксперимент, описанный в примере 5.9. В качестве основы для своей имплементации вы можете использовать имплементацию, описанную в *разд. 5.6.3*.

5.9. Обобщения

5.9.1. Доказательство того, что стоимостная функция наименьших квадратов всегда является выпуклой

Покажем, что стоимостная функция наименьших квадратов для линейной регрессии всегда является выпуклой квадратичной функцией. Рассматривая только p-е слагаемое стоимости наименьших квадратов, мы имеем

$$\left(\dot{\mathbf{x}}_p^T \mathbf{w} - y_p\right)^2 = \left(\dot{\mathbf{x}}_p^T \mathbf{w} - y_p\right)\left(\dot{\mathbf{x}}_p^T \mathbf{w} - y_p\right) = y_p^2 - 2\dot{\mathbf{x}}_p^T \mathbf{w} y_p + \dot{\mathbf{x}}_p^T \mathbf{w} \dot{\mathbf{x}}_p^T \mathbf{w}, \quad (5.44)$$

где мы расположили члены в порядке возрастания степени.

Теперь, поскольку $\dot{\mathbf{x}}_p^T\mathbf{w} = \mathbf{w}^T\dot{\mathbf{x}}_p$, мы можем поменять местами первое внутреннее произведение в третьем члене справа, получив эквивалентную форму

$$y_p^2 - 2\dot{\mathbf{x}}_p^T\mathbf{w}y_p + \mathbf{w}^T\dot{\mathbf{x}}_p\dot{\mathbf{x}}_p^T\mathbf{w}\,. \tag{5.45}$$

Это всего лишь p-е слагаемое. Суммирование по всем точкам аналогично дает

$$\begin{aligned} g(\mathbf{w}) &= \frac{1}{P}\sum_{p=1}^{P}\left(y_p^2 - 2\dot{\mathbf{x}}_p^T\mathbf{w}y_p + \mathbf{w}^T\dot{\mathbf{x}}_p\dot{\mathbf{x}}_p^T\mathbf{w}\right) = \\ &= \frac{1}{P}\sum_{p=1}^{P}y_p^2 - \frac{2}{P}\sum_{p=1}^{P}y_p\dot{\mathbf{x}}_p^T\mathbf{w} + \frac{1}{P}\sum_{p=1}^{P}\mathbf{w}^T\dot{\mathbf{x}}_p\dot{\mathbf{x}}_p^T\mathbf{w}. \end{aligned} \tag{5.46}$$

Отсюда мы можем заметить, что стоимостная функция наименьших квадратов на самом деле является квадратичной, т. к. обозначив через

$$\begin{aligned} a &= \frac{1}{P}\sum_{p=1}^{P}y_p^2; \\ \mathbf{b} &= -\frac{2}{P}\sum_{p=1}^{P}\dot{\mathbf{x}}_p y_p; \\ \mathbf{C} &= \frac{1}{P}\sum_{p=1}^{P}\dot{\mathbf{x}}_p\dot{\mathbf{x}}_p^T, \end{aligned} \tag{5.47}$$

мы можем записать стоимость наименьших квадратов эквивалентным образом как

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}\,. \tag{5.48}$$

Кроме того, поскольку матрица $\mathbf{C}$ построена из суммы матриц *внешних произведений*, она также является выпуклой, т. к. собственные значения такой матрицы всегда неотрицательны (более подробную информацию о выпуклых квадратичных функциях этой формы *см. в разд. 4.1* и *4.2*).

6 Линейная двухклассовая классификация

6.1. Введение

В этой главе мы обсудим *линейную двухклассовую классификацию* — еще один вид задачи *обучения с учителем* (*см. разд. 1.3.1*). Разница между классификацией и регрессией (подробно описанной в предыдущей главе) является довольно тонкой: двухклассовая (или двоичная) классификация — это название, которое мы даем задаче обучения с учителем, когда выход набора данных принимает форму только двух *дискретных* значений, часто именуемых двумя *классами*. В эту категорию попадают многие популярные задачи машинного обучения, включая обнаружение лиц (два класса здесь — это объекты с лицом и без лица) и обнаружение объектов в целом, сентиментный анализ на основе текста (здесь два класса состоят из письменных отзывов о товаре с положительным или отрицательным мнением), автоматическую диагностику патологических состояний здоровья (два класса в данном случае относятся к медицинским данным, которые соответствуют пациентам, имеющим либо не имеющим конкретное заболевание) и многое другое.

Это тонкое различие является важным и стимулирует разработку новых стоимостных функций, которые лучше подходят для работы с такими данными. Эти новые стоимостные функции сформулированы на основе широкого спектра мотивирующих точек зрения, включая *логистическую регрессию*, *персептрон* и *машины опорных векторов*, которые мы подробно обсудим. Хотя эти точки зрения сильно различаются навскидку, все они (как мы увидим) сводятся практически к одному и тому же существенному принципу двухклассовой классификации. Как и в предыдущей главе, здесь мы рассмотрим метрики, которые помогают нам определять качество натренированного классификатора, а также расширения базовой концепции (включая категориальную и взвешенную двухклассовую классификацию).

6.2. Логистическая регрессия и стоимость перекрестной энтропии

В этом разделе мы опишем фундаментальный каркас для линейной двухклассовой классификации, именуемый *логистической регрессией*, задействующей так называемую стоимостную функцию *перекрестной энтропии*.

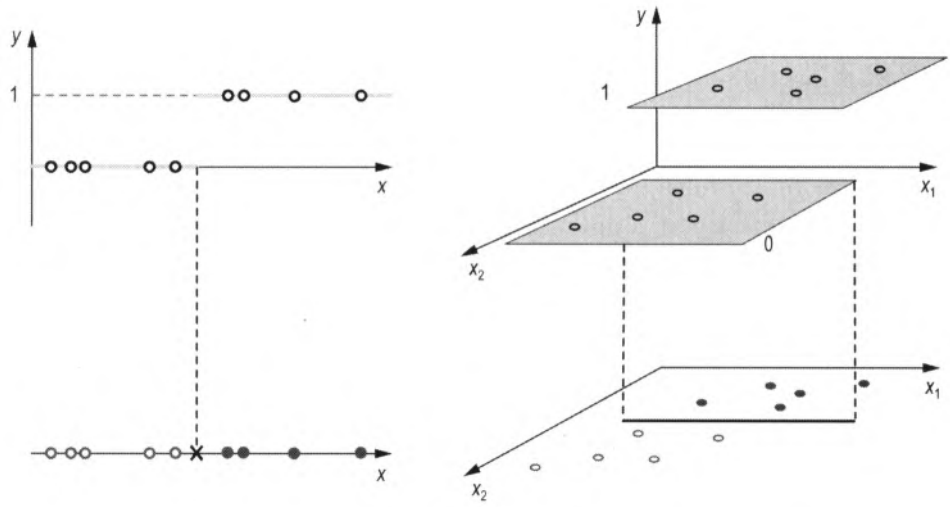
6.2.1. Нотация и моделирование

Двухклассовая классификация — это частный случай регрессии, в котором данные по-прежнему поступают в форме P пар "вход — выход" $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$, и каждый вход $\mathbf{x}_p$ является N-мерным вектором. Однако соответствующий выход y_p больше не является непрерывным, а принимает форму лишь двух дискретных чисел. Хотя фактическое значение этих чисел в принципе произвольное, некоторые пары значений более полезные, чем другие, для целей процедуры выведения. Будем предполагать, что результат наших данных принимает либо значение 0, либо +1, т. е. $y_p \in \{0, +1\}$. Часто в контексте классификации выходные значения y_p называются *метками*, а все точки, относящиеся к одинаковому значению метки, называются *классом* данных. Следовательно, набор данных, содержащий точки со значениями меток $y_p \in \{0, +1\}$, называется набором данных, состоящим из двух классов.

Самый простой способ, которым такой набор данных может быть распределен, базируется на множестве смежных шагов (рис. 6.1, *слева* для $N = 1$ и *справа* для $N = 2$). Здесь *нижний шаг* — это участок пространства, содержащий класс 0, т. е. все точки, имеющие значение метки $y_p = 0$. *Верхний шаг* содержит класс 1, т. е. все точки, имеющие значение метки $y_p = +1$. С этой позиции задачу двухклассовой классификации можно естественным образом рассматривать как случай *нелинейной* регрессии, где наша цель — добиться регрессии (или подгонки) нелинейной шаговой функции к данным. Мы называем эту точку зрения *регрессионным взглядом* на классификацию.

В качестве альтернативы мы можем изменить взгляд и смотреть на набор данных непосредственно "сверху", будто смотрим на данные сверху вниз с точки, расположенной высоко на оси y. Другими словами, мы смотрим на данные так, как будто они проецируются на плоскость $y = 0$. С этой точки зрения, которая проиллюстрирована на рис. 6.1, *снизу*, мы удаляем вертикальную y-ю размерность данных и визуально представляем набор данных, используя только его вход, изображая выходные значения каждой точки путем окраски точек одним из двух уникальных цветов: мы используем синий (незакрашенный) для точек с меткой $y_p = 0$ и красный (закрашенный) для точек с меткой $y_p = +1$. Со второй точки зрения, которую мы называем *взглядом с точки зрения персептрона* и иллюстрируем на рис. 6.1, *снизу*, ребро, разделяющее два шага (и, следовательно, точки данных на них) при проецировании на входное пространство, принимает форму одной точки при $N = 1$ (см. рис. 6.1, *слева снизу*) и прямой при $N = 2$ (см. рис. 6.1, *справа снизу*). Для общего N два класса данных разделяются *гиперплоскостью*[1], которая в контексте классификации также называется *границей решения*.

[1] Точка и прямая — это особые низкоразмерные экземпляры гиперплоскости.



одномерный вход:
граница решения — это одна-единственная точка

двумерный вход:
граница решения — это прямая

Рис. 6.1. Две точки зрения на классификацию иллюстрируются с использованием игрушечных наборов данных с одним входом (*слева*) и двумя входами (*справа*). Регрессионная точка зрения, показанная в верхней части рисунка, эквивалентна взгляду с точки зрения персептрона, отраженному в нижней части рисунка, где мы смотрим на каждый соответствующий набор данных "сверху". Во взгляде с точки зрения персептрона отмечена граница решения. Именно здесь шаговая функция (окрашенная в светло-коричневый цвет на верхних панелях) переходит из своего нижнего шага к верхнему

В этом и следующем разделах мы сосредоточимся исключительно на регрессионном взгляде на двухклассовую классификацию, а к взгляду с точки зрения персептрона вернемся в *разд. 6.4*.

6.2.2. Подгонка прерывистой шаговой функции

С позиции регрессионной точки зрения на двухклассовую классификацию у нас, возможно, возникнет искушение сначала просто применить линейно-регрессионый каркас, описанный в *главе 5*, чтобы выполнить подгонку к таким данным. Именно это мы и делаем в примере 6.1.

Пример 6.1. Подгонка линейного регрессора к классификационным данным

На рис. 6.2 показан простой двухклассовый набор данных, в который мы вписали прямую посредством линейной регрессии (сплошная линия зеленого цвета). Сама прямая обеспечивает слабое представление этих данных, т. к. ее выход принимает только два дискретных значения. Даже когда мы пропускаем этот полностью настроенный линейный регрессор через дискретную шаговую функцию, назначая метку +1 всем выходным значениям больше 0,5 и метку 0 всем выходным значениям меньше 0,5, результирующая шаговая функция (показанная красной штриховой

линией) все равно обеспечивает менее идеальное представление данных. Это обусловлено тем, что параметры (зеленой) прямой были настроены в самом начале (до прохождения результирующей модели через шаг), в итоге окончательная шаговая модель не смогла надлежаще идентифицировать две точки на верхнем шаге. На языке классификации эти типы точек называются *неправильно классифицированными точками* или, короче, *неправильными классифицированиями*.

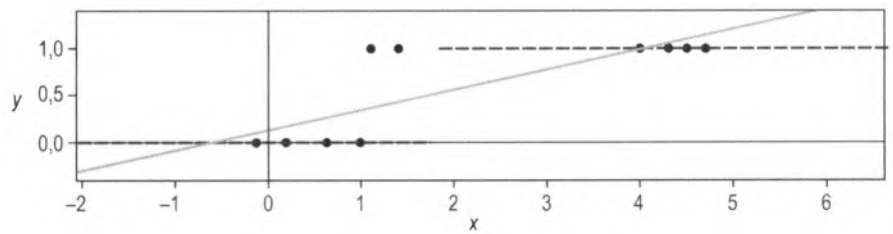


Рис. 6.2. Иллюстрация примера 6.1

Пример 6.1 намекает на тот факт, что прямое использование линейной регрессии для представления классификационных данных — это плохой вариант. Даже после прохождения результирующей линейной модели через шаговую функцию не удалось уловить истинную шаговую функцию, на которой лежали данные этого примера. Вместо настройки параметров линейной модели (выполнения линейной регрессии) и последующего пропускания результата через шаговую функцию мы можем в принципе добиться лучшего, если настроим параметры линейной модели *после* ее пропускания через шаговую функцию.

Для того чтобы описать этот вид регрессии более формально, сначала вспомним нашу нотацию (введенную в *разд. 5.2*) для обозначения линейной модели N-мерного входа

$$\dot{\mathbf{x}}^T \mathbf{w} = w_0 + x_1 w_1 + x_2 w_2 + \ldots + x_N w_N , \tag{6.1}$$

где

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \text{ и } \dot{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}. \tag{6.2}$$

Далее обозначим шаговую функцию алгебраически через[2]

$$\text{step}(t) = \begin{cases} 1, & \text{если } t > 0; \\ 0, & \text{если } t < 0. \end{cases} \tag{6.3}$$

---

[2] Значение 0 для $t < 0$ выбрано произвольно. Можно задать любое фиксированное значение либо оставить неопределенным, как мы сделали в нашем случае.

Протолкнув нашу линейную модель из уравнения (6.1) через эту функцию, мы получаем шаговую функцию[3]

$$\text{step}\left(\dot{\mathbf{x}}^T\mathbf{w}\right), \tag{6.4}$$

при этом *линейная граница* решения между ее нижним и верхним шагами определяется всеми точками $\dot{\mathbf{x}}$, где $\dot{\mathbf{x}}^T\mathbf{w} = 0$. Любой вход, лежащий *точно* на границе решения, может получить метку произвольно.

Для того чтобы надлежаще настроить весовой вектор $\mathbf{w}$, мы можем (опять же, как и в случае с линейной регрессией в *главе 5*) настроить стоимостную функцию наименьших квадратов, стремясь к идеальной связи между входом и выходом из нашего набора данных. В идеале мы хотим, чтобы точка $(\mathbf{x}_p, y_p)$ лежала на правильной стороне от границы оптимального решения, или, другими словами, выход y_p лежал на надлежащем шаге. Выраженное алгебраически, это идеальное желание может быть записано как

$$\text{step}\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) = y_p, \quad p = 1, ..., P. \tag{6.5}$$

Для того чтобы отыскать веса, удовлетворяющие этому множеству из P равенств, мы формируем стоимостную функцию наименьших квадратов путем возведения в квадрат разницы между обеими частями каждого равенства и беря их среднее арифметическое значение:

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{step}\left(\dot{\mathbf{x}}^T\mathbf{w}\right) - y_p\right)^2. \tag{6.6}$$

Наши идеальные веса тогда соответствуют минимизатору этой стоимостной функции.

К сожалению, очень трудно (если не невозможно) надлежаще минимизировать функцию наименьших квадратов с использованием локальной оптимизации, т. к. практически в каждой точке указанная функция является *полностью плоской* локально (см. пример 6.2). Эта проблема, унаследованная от нашего использования шаговой функции, делает градиентный спуск и метод Ньютона неэффективными, поскольку на плоских областях стоимостной функции оба метода немедленно останавливаются.

Пример 6.2. Визуальный осмотр классификационных стоимостных функций

На рис. 6.3, *слева* построен график стоимостной функции наименьших квадратов из уравнения (6.6) для набора данных, показанного ранее на рис. 6.2, над широким диапазоном значений его двух параметров w_0 и w_1. Эта поверхность наименьших

---

[3] Технически $\text{step}\left(\dot{\mathbf{x}}^T\mathbf{w} - 0{,}5\right)$ — это функция, которая отображает выходные значения $\dot{\mathbf{x}}^T\mathbf{w}$ больше (меньше) 0,5 в 1 (0). Однако мы можем соединить константу −0,5 с весом смещения w_0, переписав его как $w_0 \leftarrow w_0 - 0{,}5$ (в конце концов, это тот параметр, который должен быть усвоен), чтобы выразить шаговую функцию компактнее, как это делается в уравнении (6.4).

квадратов состоит из дискретных шагов на многих разных уровнях, и каждый шаг является *полностью плоским*. По этой причине методы локальной оптимизации (подобные тем, которые подробно описаны в *главах 2–4*) не получится использовать для эффективного ее минимизирования.

На рис. 6.3, *в центре* и *справа* построены графики поверхности двух родственных стоимостных функций над одним и тем же набором данных. Мы ввели стоимостную функцию на рис. 6.3, *в центре*, и стоимостную функцию на рис. 6.3, *справа*. Обе они намного превосходят, с точки зрения нашей способности надлежаще их минимизировать, шаговую стоимостную функцию наименьших квадратов (см. рис. 6.3, *слева*).

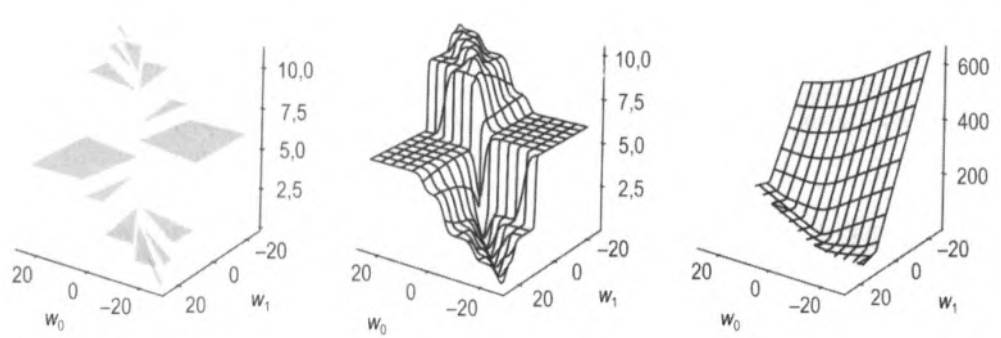


Рис. 6.3. Иллюстрация примера 6.2

6.2.3. Логистическая сигмоидная функция

Для того чтобы сделать возможной минимизацию стоимости наименьших квадратов, мы можем заменить шаговую функцию в уравнении (6.6) *непрерывной* аппроксимацией, которая близка к ней. *Логистическая сигмоидная функция*

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6.7}$$

является такой аппроксимацией. На рис. 6.4 построен график этой функции (*слева*), а также нескольких ее внутренне взвешенных версий (*справа*). Видно, что для правильной установки внутренних весов логистическую сигмоиду можно сделать похожей на шаговую функцию.

6.2.4. Логистическая регрессия с использованием стоимости наименьших квадратов

Замена шаговой функции ее сигмоидной аппроксимацией в уравнении (6.5) дает соответствующее множество *приближенных* равенств, которые мы желаем соблюсти:

$$\sigma\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right) \approx y_p, \quad p = 1, ..., P, \tag{6.8}$$

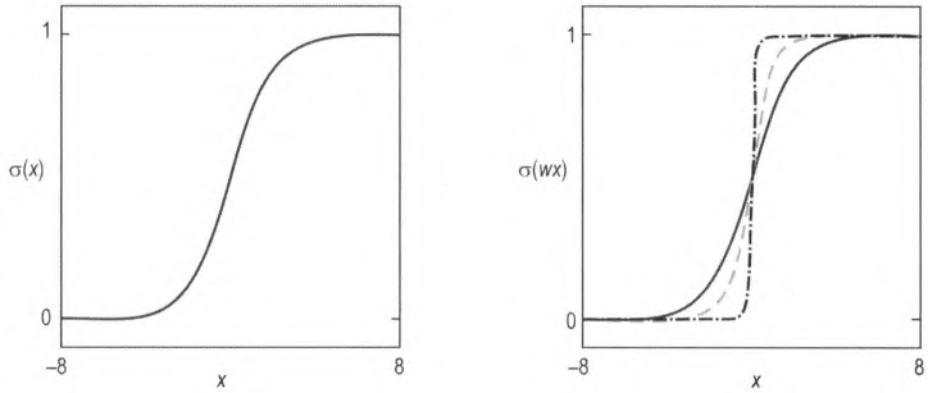


Рис. 6.4. График сигмоидной функции $\sigma(x)$ (*слева*). По мере увеличения веса w в $\sigma(wx)$ с $w = 1$ (сплошная линия) до $w = 2$ (штриховая линия) и, наконец, до $w = 10$ (штрихпунктирная линия) внутренне взвешенная версия сигмоидной функции становится все более и более приемлемым аппроксиматором шаговой функции (*справа*)

а также соответствующую стоимостную функцию наименьших квадратов:

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\sigma\left(\dot{\mathbf{x}}^{T}\mathbf{w}\right) - y_p\right)^2. \tag{6.9}$$

Подгонка логистической сигмоиды к классификационным данным путем минимизирования этой стоимостной функции часто упоминается как выполнение *логистической регрессии*[4]. Хотя результирующая стоимостная функция обычно не является выпуклой, тем не менее она может быть надлежаще минимизирована с помощью массы приемов локальной оптимизации.

Пример 6.3. Использование нормализованного градиентного спуска

На рис. 6.5 показано, как нормализованный градиентный спуск (легкая вариация алгоритма стандартного градиентного спуска, описанного в *главе 3*, более подробную информацию *см. в разд. П1.3*) может использоваться для минимизирования стоимости наименьших квадратов из уравнения (6.9) над набором данных, впервые представленным на рис. 6.2. Здесь был выполнен прогон нормализованного градиентного спуска, инициализированного в точке $w_0 = -w_1 = 20$. На рис. 6.5, *слева* построен график сигмоидальной подгонки, обеспечиваемый надлежащим минимизированием функции наименьших квадратов, а на рис. 6.5, *справа* контурный график стоимостной функции показан с (нормализованным) градиентным спуском, окрашенным от зеленого цвета до красного по мере прогрессии к минимизатору стоимостной функции. График поверхности этой стоимостной функции (в трех размерностях) приведен на рис. 6.3, *в центре*. Хотя эта стоимостная функция во многих местах является очень плоской, нормализованный градиентный спуск сконструирован специально для работы с такими стоимостями (*см. разд. П1.3*).

[4] Потому что мы по существу выполняем *регрессию*, используя *логистическую* функцию.

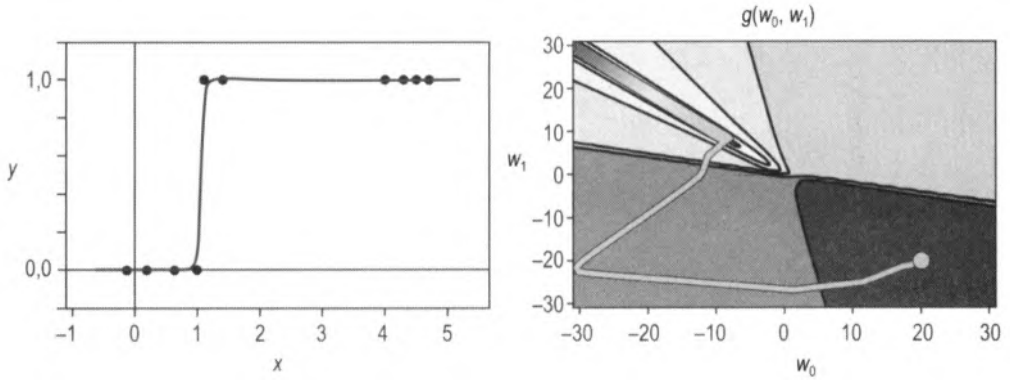


Рис. 6.5. Иллюстрация примера 6.3

6.2.5. Логистическая регрессия с использованием стоимости перекрестной энтропии

Поточечная стоимость на основе *квадратической ошибки* $g_p(\mathbf{w}) = \left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) - y_p\right)^2$, которую мы усредняем над всеми P точками в данных для формирования стоимости наименьших квадратов из уравнения (6.9), определена универсально, независимо от значений, принимаемых y_p на выходе. Однако, поскольку мы *знаем*, что выход, с которым мы имеем дело в двухклассовой классификации, ограничен *дискретными* значениями $y_p \in \{0, 1\}$, разумно задаться вопросом: можем ли мы создать более подходящую функцию, специально настроенную для работы именно с такими значениями?

Одна из таких поточечных стоимостей, которую мы называем *логарифмической ошибкой*, определяется следующим образом:

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right), & \text{если } y_p = 1; \\ -\log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right), & \text{если } y_p = 0. \end{cases} \tag{6.10}$$

Во-первых, обратите внимание на то, что эта поточечная функция всегда неотрицательна (независимо от входных и весовых значений), а ее минимальное[5] значение равно 0.

Во-вторых, обратите внимание на то, что эта стоимость *логарифмической ошибки* штрафует нарушения наших желаемых (приближенных) равенств из уравнения (6.8) гораздо жестче, чем *квадратическая ошибка*, как это видно на рис. 6.6, где они обе построены на графике для сравнения.

Наконец, обратите внимание: поскольку наши значения меток $y_p \in \{0, 1\}$, мы можем записать *логарифмическую ошибку* из уравнения (6.10) эквивалентным образом одной строкой как:

$$g_p(\mathbf{w}) = -y_p \log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) - \left(1 - y_p\right)\log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right). \tag{6.11}$$

[5] Инфимумным (т. е. самым нижним).

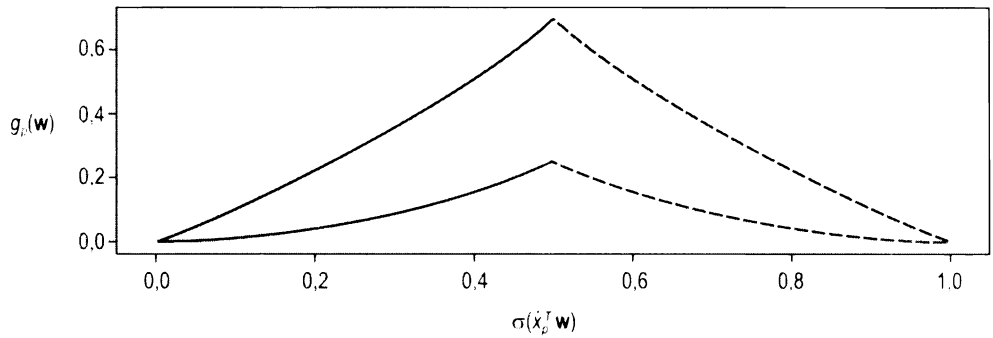


Рис. 6.6. Визуальное сравнение квадратической ошибки (синим цветом) и логарифмической ошибки (красным цветом) для двух случаев: $y_p = 0$ (сплошные кривые) и $y_p = 1$ (штриховые кривые). В обоих случаях логарифмическая ошибка штрафует отклонение от истинного значения метки в большей степени, чем квадратическая ошибка

Указанная эквивалентная форма позволяет нам записать совокупную стоимостную функцию (формируемую путем взятия среднего значения поточечных стоимостей над всеми P точками данных) как

$$g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P} y_p \log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) + \left(1 - y_p\right)\log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right). \tag{6.12}$$

Эта формула называется функцией перекрестной *энтропии для логистической регрессии.*

6.2.6. Минимизирование стоимости перекрестной энтропии

На рис. 6.3, *справа* показана поверхность стоимости перекрестной энтропии, взятой над набором данных, представленным на рис. 6.2. Не случайно построенная на графике поверхность выглядит выпуклой. Действительно (в отличие от наименьших квадратов), стоимость перекрестной энтропии всегда является выпуклой независимо от используемого набора данных (см. упражнения главы). Это означает, что для ее правильного минимизирования можно использовать более широкое разнообразие схем локальной оптимизации (по сравнению с обычно невыпуклыми наименьшими квадратами на основе сигмоиды), включая схемы стандартного градиентного спуска (*см. разд. 3.5*) и ньютоновы методы второго порядка (*см. разд. 4.3*). По этой причине стоимость перекрестной энтропии очень часто применяется на практике для выполнения логистической регрессии.

Пример 6.4. Минимизирование перекрестно-энтропийной логистической регрессии

В этом примере мы повторяем эксперименты примера 6.3, используя стоимость перекрестной энтропии и стандартный градиентный спуск, инициализированные в точках $w_0 = 3$ и $w_1 = 3$. На рис. 6.7, *слева* показана сигмоидальная аппроксимация, обеспечиваемая за счет надлежащего минимизирования функции перекрестной энтропии. *Справа* показан контурный график стоимостной функции с траекторией

градиентного спуска, окрашенной в диапазоне от зеленого цвета до красного по мере продвижения к минимизации. График поверхности этой стоимостной функции изображен на рис. 6.3, *справа*.

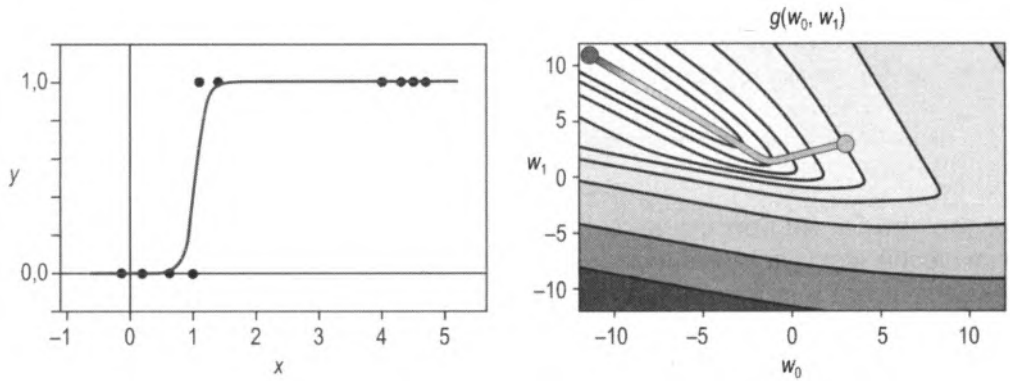


Рис. 6.7. Иллюстрация примера 6.4

6.2.7. Имплементация на Python

Мы можем имплементировать функцию перекрестной энтропии очень похоже на имплементацию функции наименьших квадратов для линейной регрессии (*см. разд. 5.2.4*), разбив нашу имплементацию на линейную модель и саму функцию ошибки. Наша линейная модель принимает дополненную входную точку $\dot{\mathbf{x}}_p$ и множество весов $\mathbf{w}$

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \dot{\mathbf{x}}_p^T \mathbf{w} \,. \tag{6.13}$$

И ее мы по-прежнему можем имплементировать так, как показано ниже.

```
# вычислить линейную комбинацию входных точек
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

Затем мы можем имплементировать функцию перекрестной энтропии, используя логарифмическую ошибку из уравнения (6.10), как показано ниже.

```
# определить сигмоидную функцию
def sigmoid(t):
    return 1/(1 + np.exp(-t))

# выпуклая стоимостная функция перекрестной энтропии
def cross_entropy(w):
    # вычислить сигмоиду модели
    a = sigmoid(model(x,w))

```

```
    # вычислить стоимость точек метки 0
    ind = np.argwhere(y == 0)[:,1]
    cost = -np.sum(np.log(1 - a[:,ind]))

    # добавить стоимость в точки метки 1
    ind = np.argwhere(y==1)[:,1]
    cost -= np.sum(np.log(a[:,ind]))

    # вычислить перекрестную энтропию
    return cost/y.size
```

Для того чтобы минимизировать эту функцию, мы можем использовать практически любой метод локальной оптимизации, подробно описанный в *главах 2–4*. Для методов первого и второго порядков (например, схем градиентного спуска и метода Ньютона) библиотека автоматической дифференциации autograd (*см. разд. 3.4*) может быть использована для автоматического вычисления его градиента и гессиана.

В качестве альтернативы на самом деле можно вычислить градиент и гессиан функции перекрестной энтропии в замкнутой форме и имплементировать их непосредственно. Используя простые правила производных, описанные в *разд. П2.3*, градиент можно вычислить следующим образом:

$$\nabla g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P}\left(y_p - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right)\dot{\mathbf{x}}_p \,. \tag{6.14}$$

В дополнение к задействованию метода Ньютона "вручную" можно вычислить гессиан функции перекрестной энтропии вручную как

$$\nabla^2 g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P}\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right)\dot{\mathbf{x}}_p\dot{\mathbf{x}}_p^T \,. \tag{6.15}$$

6.3. Логистическая регрессия и функция softmax (функция мягкого максимума)

В предыдущем разделе мы рассмотрели, как выводить логистическую регрессию при задействовании значений меток $y_p \in \{0, 1\}$. Однако, как упоминалось ранее, эти значения меток носят произвольный характер, и существует возможность вывести логистическую регрессию, используя другой набор меточных значений, например $y_p \in \{-1, +1\}$. В данном коротком разделе мы сделаем как раз это, пройдя шаги, полностью аналогичные тому, что мы видели ранее, и в результате получив новую стоимостную функцию, именуемую *softmax-функцией* (или *функцией мягкого максимума*) для логистической регрессии. В то время как softmax-функция отличается по форме алгебраически, фактически же она эквивалентна стоимости перекрестной энтропии. Однако с концептуальной точки зрения softmax-функция является весьма ценной, поскольку позволяет нам соединить много разных точек зрения на двух-

классовую классификацию в единую когерентную идею, что мы и увидим в последующих разделах.

6.3.1. Разные метки, одинаковая история

Если мы поменяем значения меток с $y_p \in \{0, 1\}$ на $y_p \in \{-1, +1\}$, то бóльшая часть истории, которая, как мы видели, разворачивалась ранее, развернется и здесь. То есть вместо идеального расположения наших данных на шаговой функции, в которой нижние и верхние шаги принимают соответственно значения 0 и 1, они принимают значения -1 и $+1$, как показано на рис. 6.8 для прототипических случаев, когда $N = 1$ (*слева*) и $N = 2$ (*справа*).

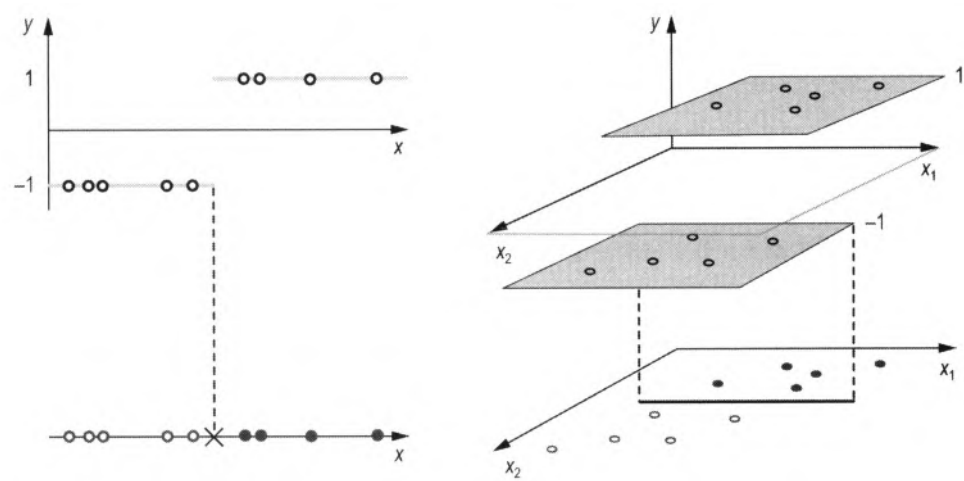
Эта конкретная шаговая функция называется *знаковой функцией*, т. к. она возвращает числовой знак своего входа:

$$\text{sign}(x) = \begin{cases} +1, & \text{если } x > 0; \\ -1, & \text{если } x < 0. \end{cases} \tag{6.16}$$

Проталкивание *линейной* модели через знаковую функцию дает нам шаговую функцию

$$\text{sign}\left(\dot{\mathbf{x}}^T \mathbf{w}\right), \tag{6.17}$$

при этом *линейная* граница решения между двумя ее шагами определяется всеми точками $\dot{\mathbf{x}}$, где $\dot{\mathbf{x}}^T \mathbf{w} = 0$. Любой вход, лежащий *точно* на границе решения, может получить метку произвольно. Точка классифицируется правильно, когда ее истинная метка предсказана правильно, т. е. когда $\text{sign}\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right) = y_p$. В противном случае считается, что она была классифицирована *неправильно* (misclassified).



одномерный вход:
граница решения — это одна-единственная точка

двумерный вход:
граница решения — это прямая

Рис. 6.8. Условия, аналогичные рис. 6.1, только здесь мы используем значения меток $y_p \in \{-1, +1\}$

Как и при использовании значений меток $y_p \in \{0, 1\}$, мы можем снова попытаться сформировать функцию наименьших квадратов с помощью знаковой функции. Однако, как и в случае с исходными наименьшими квадратами на основе шагов из уравнения (6.6), она тоже будет полностью плоской почти везде, и, следовательно, минимизировать ее надлежаще будет чрезвычайно трудно.

Как и ранее, мы можем попытаться заменить прерывистую знаковую функцию $(\cdot)$ *гладкой аппроксимацией*: слегка скорректированной версией логистической сигмоидной функции, чтобы ее значения находились в интервале от -1 до 1 (вместо 0 и 1). Эта шкалированная версия сигмоиды, именуемая *функцией гиперболического тангенса*, записывается как

$$\tanh(x) = 2\sigma(x) - 1 = \frac{2}{1 + e^{-x}} - 1. \tag{6.18}$$

Учитывая, что сигмоидная функция $\sigma(\cdot)$ гладко варьируется между 0 и 1, легко понять, почему $\tanh(\cdot)$ гладко варьируется между -1 и $+1$.

Аналогично стоимости наименьших квадратов из уравнения (6.9) мы можем сформировать стоимость наименьших квадратов для восстановления оптимальных модельных весов, используя функцию $\tanh(\cdot)$:

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\tanh\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) - y_p\right)^2, \tag{6.19}$$

которая также является невыпуклой с нежелательными плоскими участками, требуя для ее надлежащей минимизации специализированных локальных методов (см. пример 6.3).

Как и в случае с метками 0/1, здесь мы также можем задействовать поточечную стоимость *логарифмической ошибки*

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right), & \text{если } y_p = +1; \\ -\log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right), & \text{если } y_p = -1, \end{cases} \tag{6.20}$$

которую затем можем использовать для формирования так называемой softmax-функции для логистической регрессии

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} g_p(\mathbf{w}). \tag{6.21}$$

Как и в случае с функцией перекрестной энтропии, гораздо более распространено выражать функцию мягкого максимума по-другому, переписав логарифмическую ошибку эквивалентно следующим образом. Прежде всего, обратите внимание: поскольку $1 - \sigma(\mathbf{x}) = \sigma(-\mathbf{x})$, поточечная функция в уравнении (6.20) может быть переписана эквивалентным образом как $-\log\left(\sigma\left(\dot{\mathbf{x}}^T\mathbf{w}\right)\right)$, и поэтому поточечная стоимостная функция может быть записана как

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right), & \text{если } y_p = +1; \\ -\log\left(\sigma\left(-\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right), & \text{если } y_p = -1. \end{cases} \tag{6.22}$$

Теперь обратите внимание на то, что из-за особого выбора значений меток, которые мы здесь используем, т. е. $y_p \in \{-1, +1\}$, мы можем переместить значение метки в каждом случае *внутрь* самых внутренних скобок и записать оба случая в одну строку следующим образом:

$$g_p(\mathbf{w}) = -\log\left(\sigma\left(y_p\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right). \tag{6.23}$$

Наконец, поскольку $-\log(x) = \log(1/x)$, мы можем записать поточечную функцию из уравнения (6.23) эквивалентным образом (используя определение сигмоиды) как

$$g_p(\mathbf{w}) = \log\left(1 + e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}\right). \tag{6.24}$$

Подставив эту форму поточечной функции логарифмической ошибки в уравнение (6.21), мы получаем более общий внешний вид *softmax-функции* для логистической регрессии

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}\right). \tag{6.25}$$

Эта стоимостная функция, как и стоимость перекрестной энтропии, подробно описанная в предыдущем разделе, всегда является выпуклой независимо от используемого набора данных (см. упражнения главы). Более того, как мы видим здесь по процедуре ее выведения, стоимостные функции мягкого максимума и перекрестной энтропии полностью эквивалентны (при замене значения $y_p = -1$ метки на $y_p = 0$ и наоборот), т. к. построены с использованием одинаковой поточечной стоимостной функции логарифмической ошибки.

Пример 6.5. Минимизирование мягкомаксимумной логистической регрессии с помощью стандартного градиентного спуска

В этом примере мы повторяем эксперименты примера 6.4, меняя метки $y_p = 0$ на $y_p = -1$, чтобы сформировать softmax-функцию и использовать градиентный спуск

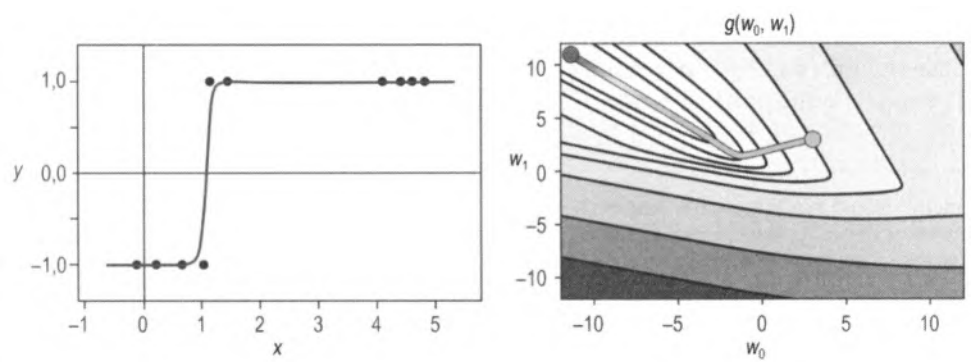


Рис. 6.9. Иллюстрация примера 6.5

(с той же начальной точкой, параметром длины шага и числом итераций) для ее минимизации. Результаты показаны на рис. 6.9.

6.3.2. Имплементация на Python

Если мы выразим softmax-функцию, используя логарифмическую ошибку, как в уравнении (6.21), то можем имплементировать ее почти полностью так же, как мы сделали это с функцией перекрестной энтропии (*см. разд. 6.2.7*).

Для реализации softmax-функции, как показано в уравнении (6.25), мы сначала реализуем линейную модель, которая принимает как дополненную входную точку $\dot{\mathbf{x}}_p$, так и множество весов $\mathbf{w}$ в виде

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \dot{\mathbf{x}}_p^T \mathbf{w} . \tag{6.26}$$

С этой нотацией для нашей модели соответствующая softmax-функция может быть записана как

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1 + e^{-y_p \text{model}\left(\mathbf{x}_p, \mathbf{w}\right)}\right).$$

Затем мы можем разработать функцию по частям — сначала приведенную ниже функцию `model` точно так же, как мы это сделали с линейной регрессией (*см. разд. 5.2.4*).

```
# вычислить линейную комбинацию входных точек
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

Затем мы можем разработать softmax-функцию:

```
# выпуклая стоимостная функция мягкого максимума
def softmax(w):
    cost = np.sum(np.log(1 + np.exp(-y*model(x,w))))
    return cost/float(np.size(y))
```

В качестве альтернативы использованию автоматического дифференциатора (который мы применяем по умолчанию — задействуя библиотеку autograd (*см. разд. 3.4*)) здесь можно выполнить градиентный спуск и метод Ньютона, вручную вычислив градиент и гессиан softmax-функции. Используя простые правила производных, описанные в *разд. П2.3*, градиент можно вычислить как[6]

---

[6] Градиент можно выразить алгебраически несколькими способами. Однако написание градиента таким образом помогает избежать численных проблем, связанных с использованием экспоненциальной функции на современном компьютере. Это происходит из-за экспоненциального "переполнения" в случае крупных экспонент, например e^{1000}, т. к. эти числа слишком велики для явного хранения на компьютере и поэтому представляются символически как ∞. Такая ситуация становится проблемой при вычислении выражения $e^{1000}/\left(1+e^{1000}\right)$, которое, хотя в сущности и равно значению 1, считается компьютером равным NaN (not a number, т. е. не числом).

$$\nabla g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P}\frac{e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}}{1+e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}}y_p\dot{\mathbf{x}}_p\,. \tag{6.27}$$

В дополнение к методу Ньютона "вручную" можно вычислить гессиан функции softmax-вручную как

$$\nabla^2 g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\frac{1}{1+e^{y_p\dot{\mathbf{x}}_p^T\mathbf{w}}}\right)\left(1-\frac{1}{1+e^{y_p\dot{\mathbf{x}}_p^T\mathbf{w}}}\right)\dot{\mathbf{x}}_p\dot{\mathbf{x}}_p^T\,. \tag{6.28}$$

6.3.3. Шумные классификационные данные

Дискретная природа выходных данных в классификации отличает понятия *шума* и *шумных* данных в линейной классификации от того, что мы видели ранее в случае с линейной регрессией в *главе 5*. В линейной регрессии шум приводит к тому, что данные не попадают точно на одну прямую (или гиперплоскость в более высоких размерностях). В случае линейной двухклассовой классификации шум проявляется в нашей *неспособности* отыскать единственную прямую (или гиперплоскость в более высоких размерностях) для разделения двух классов данных. На рис. 6.10 показан такой шумный классификационный набор данных, состоящий из $P = 100$ точек, два класса которых можно разделить прямой, но не полностью.

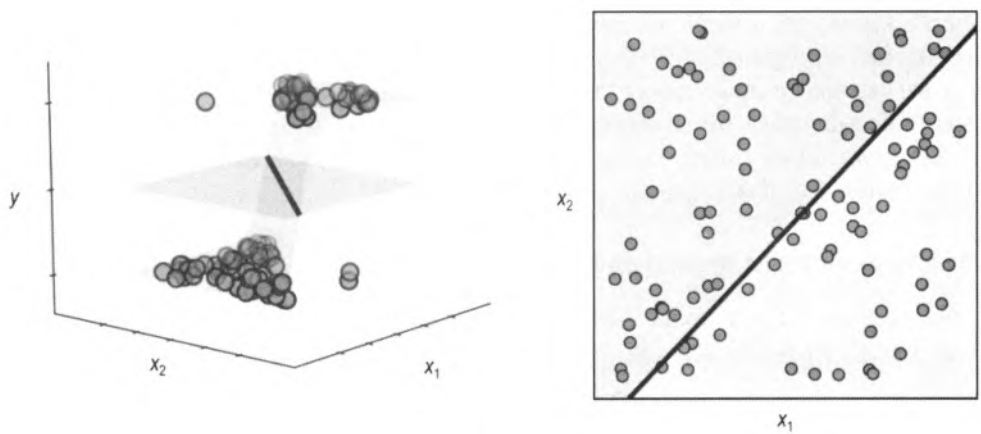


Рис. 6.10. Двухклассовый классификационный набор данных, рассматриваемый с точки зрения регрессии (*слева*) и с точки зрения персептрона (*справа*)

На рис. 6.10, *слева* показаны данные в трех размерностях (рассматриваемых с точки зрения регрессии) вместе с натренированным классификатором — трехмерной гиперболической тангенсной функцией. На рис. 6.10, *справа* показан тот же самый набор данных в двух размерностях (рассматриваемых с точки зрения персептрона) вместе с усвоенной линейной границей решения. Здесь эти два полупространства, созданные границей решения, также окрашены (светло-синим и светло-красным цветами) в соответствии с классом, замкнутым внутри каждого из них. Как вы видите, в этом случае имеются три точки (две синие точки и одна красная), которые

выглядят так, как будто они находятся на *неправильной стороне* нашего классификатора. Такие *шумные* точки часто классифицируются натренированным классификатором неправильно, и, следовательно, значение их истинной метки не будет правильно предсказано. Двухклассовые классификационные наборы данных обычно имеют такого рода шум и поэтому часто не полностью разделяемы линейной границей решения.

6.4. Перцептрон

Как мы видели в случае с логистической регрессией в предыдущем разделе, мы трактуем классификацию как особую форму нелинейной регрессии (задействуя — с вариантами значений меток $y_p \in \{-1, +1\}$ — нелинейность функции tanh). Это приводит к усвоению надлежащего нелинейного регрессора и соответствующей *линейной границы решения*

$$\dot{\mathbf{x}}^T \mathbf{w} = 0 . \tag{6.29}$$

Вместо того чтобы усваивать эту границу решения в результате нелинейной регрессии, процедура выведения *перцептрона*, описанная в данном разделе, призвана определять эту идеальную линейную границу решения непосредственно. Как мы увидим, этот непосредственный подход приводит обратно к *функции мягкого максимума*, и на практике перцептрон и логистическая регрессия часто приводят к усвоению *одинаковой линейной границы решения*. Вместе с тем фокус внимания перцептрона на непосредственном усвоении границы решения обеспечивает ценный новый взгляд на процесс двухклассовой классификации. Так, мы увидим, что перцептрон обеспечивает простой геометрический контекст для введения важного понятия *регуляризации* (идеи, которая возникает в различных формах на протяжении всей остальной книги).

6.4.1. Стоимостная функция перцептрона

Как мы видели в предыдущем разделе при обсуждении логистической регрессии (где наш выход/значения меток $y_p \in \{-1, +1\}$), в простейшем случае наши два класса данных в значительной степени разделены *линейной границей решения*, заданной коллекцией входов $\mathbf{x}$, где $\dot{\mathbf{x}}^T \mathbf{w} = 0$, причем элементы классов располагаются по одной стороне (главным образом) от границы решения. Эта граница решения является *точкой*, когда размерность входных данных равна $N = 1$, *прямой*, когда $N = 2$, и в более общем случае для произвольных N — *гиперплоскостью*, определенной во входном пространстве набора данных.

Этот классификационный сценарий лучше всего визуализируется в случае $N = 2$, когда мы смотрим на задачу классификации "сверху" и видим входы набора данных, окрашенными для обозначения принадлежности к классу. Принятая здесь по умолчанию и используемая нами схема окраски — совпадающая со схемой, использованной в предыдущем разделе, — состоит в том, чтобы раскрасить точки с меткой $y_p = -1$ синим и с меткой $y_p = +1$ красным. Линейной границей решения

здесь является прямая, которая наилучшим образом отделяет точки класса $y_p = -1$ от точек класса $y_p = +1$, как показано для прототипического набора данных на рис. 6.11.

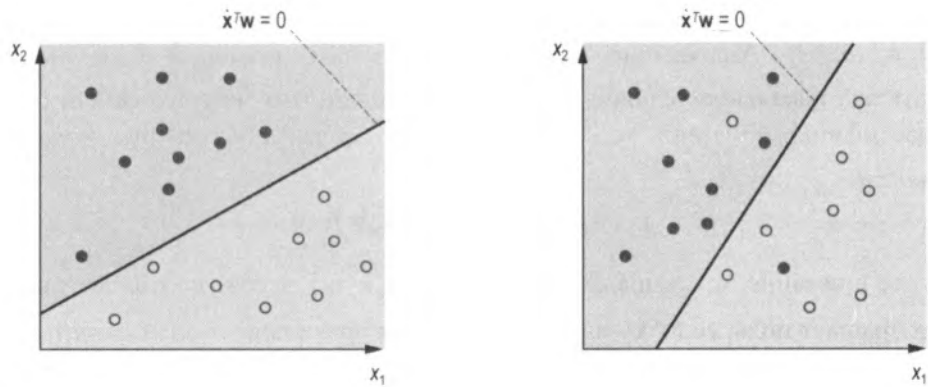


Рис. 6.11. С перцептроном мы стремимся усвоить линейную границу решения $\dot{\mathbf{x}}^T\mathbf{w} = 0$ (показанную черным цветом) непосредственно, чтобы разделить два класса данных, окрашенных красным (класс $+1$) и синим цветами (класс -1), путем разделения входного пространства на красное полупространство, где $\dot{\mathbf{x}}^T\mathbf{w} > 0$, и синее полупространство, где $\dot{\mathbf{x}}^T\mathbf{w} < 0$. Линейно разделяемый набор данных, где существует возможность усвоить гиперплоскость, которая полностью разделяет два класса (*слева*). Набор данных с двумя пересекающимися классами (*справа*). Хотя распределение данных не допускает полного линейного разделения, персептрон по-прежнему стремится отыскать гиперплоскость, минимизирующую число неправильно классифицированных точек, которые оказываются в неправильном полупространстве

Линейная граница решения разделяет входное пространство на два *полупространства*: одно лежит "выше" гиперплоскости, где $\dot{\mathbf{x}}^T\mathbf{w} > 0$, а другое — "ниже" нее, где $\dot{\mathbf{x}}^T\mathbf{w} < 0$. Обратите внимание, как показано визуально на рис. 6.11, что надлежащее множество весов $\mathbf{w}$ определяет линейную границу решения, которая разделяет двухклассовый набор данных настолько хорошо, насколько это возможно, с максимально возможным числом членов одного класса, лежащих над ней, и таким же максимально возможным числом членов другого класса, лежащих ниже него. Поскольку мы всегда можем изменить ориентацию идеальной гиперплоскости, умножив ее на -1 (или на какое-то другое значение, потому что мы всегда можем поменять местами наши два значения меток), мы можем сказать в общем, что, когда веса гиперплоскости настроены надлежаще, члены класса $y_p = +1$ лежат (главным образом) "выше" ее, в то время как члены класса $y_p = -1$ лежат (главным образом) "ниже" ее. Другими словами, наш *желаемый* набор весов определяется гиперплоскостью, такой, что мы в большинстве случаев имеем:

$$\begin{aligned} \dot{\mathbf{x}}_p^T\mathbf{w} > 0, &\quad \text{если } y_p = +1; \\ \dot{\mathbf{x}}_p^T\mathbf{w} < 0, &\quad \text{если } y_p = -1. \end{aligned} \tag{6.30}$$

Ввиду нашего выбора значений меток мы можем консолидировать приведенные выше идеальные условия в единое уравнение:

$$-y_p\dot{\mathbf{x}}_p^T\mathbf{w} < 0\,. \tag{6.31}$$

Опять-таки, мы можем сделать это специально, потому что выбрали значения меток $y_p \in \{-1, +1\}$. Аналогично, взяв максимум этой величины и ноль, мы можем записать это идеальное условие, которое утверждает, что гиперплоскость правильно классифицирует точку $\mathbf{x}_p$, эквивалентным образом формируя *поточечную стоимость*

$$g_p(\mathbf{w}) = \max\left(0,\ -y_p\dot{\mathbf{x}}_p^T\mathbf{w}\right) = 0\,. \tag{6.32}$$

Обратите внимание, что выражение $\max\left(0,\ -y_p\dot{\mathbf{x}}_p^T\mathbf{w}\right)$ всегда неотрицательно, т. к. оно возвращает ноль, если $\mathbf{x}_p$ классифицируется правильно, и *положительное значение*, если точка классифицируется неправильно. Функциональная форма этой точечной стоимости $\max(0, \cdot)$ часто по историческим причинам называется *выпрямленной линейной единицей* (rectified linear unit, ReLU, *см. разд. 13.3*). Поскольку эти точечные стоимости неотрицательны и равны *нулю*, когда наши веса настроены правильно, мы можем взять их среднее арифметическое значение над всем набором данных, чтобы сформировать надлежащую стоимостную функцию как

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\max\left(0,\ -y_p\dot{\mathbf{x}}_p^T\mathbf{w}\right) = 0\,. \tag{6.33}$$

При надлежащем минимизировании эта стоимостная функция может использоваться для восстановления идеальных весов, удовлетворяющих приведенным выше желаемым уравнениям, как можно чаще.

6.4.2. Минимизирование стоимости перцептрона

Эта функция известна под многими именами, такими как стоимость персептрона, стоимость *выпрямленной линейной единицы* (или ReLU-стоимость) и *петлевая*

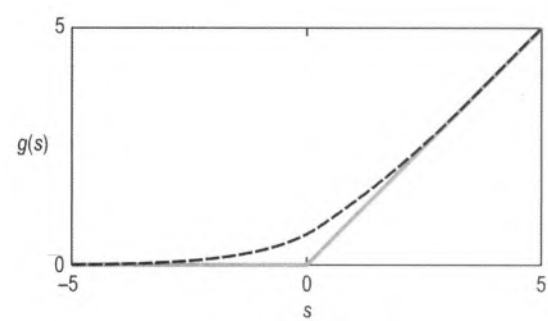


Рис. 6.12. Графики перцептрона $g(s) = \max(0, s)$ (сплошная линия зеленого цвета) и его гладкой softmax-аппроксимации $g(s) = \text{soft}(0, s) = \log(1 + e^s)$ (пунктирная кривая черного цвета)

стоимость (поскольку при построении графика функция ReLU выглядит как петля, рис. 6.12). Эта стоимостная функция *всегда* является *выпуклой*, но имеет только одну (прерывистую) производную в каждой входной размерности. Из этого вытекает, что мы можем использовать схемы только нуль- и первопорядковой локальной оптимизации (но не метод Ньютона). Обратите внимание, что стоимость перцептрона *всегда* имеет тривиальное решение в $\mathbf{w} = \mathbf{0}$, т. к. действительно $g(\mathbf{0}) = 0$, и поэтому на практике следует позаботиться о том, чтобы случайно не отыскивать его (либо слишком близкую к нему точку).

6.4.3. Softmax-аппроксимация перцептрона

В этом разделе мы опишем часто встречающийся подход к облегчению трудностей оптимизации, подробно рассмотренных выше, в отношении функции потерь перцептрона. Наши действия будут похожи на замену дискретной шаговой функции гладко аппроксимирующей сигмоидной функцией из предыдущих разделов, но здесь мы заменим максимумную часть функции перцептрона на гладкую (или, по крайней мере, дважды дифференцируемую) альтернативу, которая почти совпадает с ней везде. Сделаем это посредством функции softmax, определяемой как

$$\text{soft}(s_0, s_1, ..., s_{C-1}) = \log\left(e^{s_0} + e^{s_1} + ... + e^{s_{C-1}}\right), \tag{6.34}$$

где $s_0, s_1, ..., s_{C-1}$ — это любые C скалярных значений, которые являются обобщенной гладкой аппроксимацией функции max, т. е.

$$\text{soft}(s_0, s_1, ..., s_{C-1}) \approx \max(s_0, s_1, ..., s_{C-1}). \tag{6.35}$$

Для того чтобы понять, почему функция softmax аппроксимирует функцию max, рассмотрим простой случай, когда $C = 2$. На мгновение предположим, что $s_0 \le s_1$, и значит, $\max(s_0, s_1) = s_1$. Следовательно, $\max(s_0, s_1)$ может быть записана как $\max(s_0, s_1) = s_0 + (s_1 - s_0)$ или эквивалентным образом как $\max(s_0, s_1) = \log\left(e^{s_0}\right) + \log\left(e^{s_1 - s_0}\right)$. Записав таким образом, мы видим, что $\log\left(e^{s_0}\right) + \log\left(1 + e^{s_1 - s_0}\right) = \log\left(e^{s_0} + e^{s_1}\right) = \text{soft}(s_0, s_1)$ всегда больше, чем $\max(s_0, s_1)$, но не намного, особенно когда $e^{s_1 - s_0} \gg 1$. Поскольку тот же самый аргумент можно выдвинуть, если $s_0 \ge s_1$, то в общем случае можно сказать, что $\text{soft}(s_0, s_1) \approx \max(s_0, s_1)$. Аналогично можно получить рассуждения относительно и более общего случая.

Возвратимся к стоимостной функции персептрона из уравнения (6.33) и заменим p-е слагаемое его softmax-аппроксимацией, в результате создав нашу поточечную функцию потерь

$$g_p(\mathbf{w}) = \text{soft}\left(0, -y_p \dot{\mathbf{x}}_p^T \mathbf{w}\right) = \log\left(e^0 + e^{-y_p \dot{\mathbf{x}}_p^T \mathbf{w}}\right) = \log\left(1 + e^{-y_p \dot{\mathbf{x}}_p^T \mathbf{w}}\right) \tag{6.36}$$

и получив совокупную стоимостную функцию в виде

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1 + e^{-y_p \dot{\mathbf{x}}_p^T \mathbf{w}}\right), \tag{6.37}$$

т. е. softmax-функция, которая, как мы видели ранее, была выведена с точки зрения логистической регрессии на двухклассовую классификацию. Вот почему указанная стоимостная функция называется softmax, поскольку она выводится из общей softmax-аппроксимации функции max.

Обратите внимание, что, *как и* функция персептрона, функция softmax является выпуклой. Однако, *в отличие* от стоимости перцептрона, softmax-функция имеет бесконечно много производных, и, следовательно, для ее минимизирования можно использовать метод Ньютона. Более того, она не имеет тривиального решения при нуле как функция перцептрона. Тем не менее тот факт, что softmax-функция так близко аппроксимирует перцептрон, показывает, насколько точно совпадают — в конечном счете — взгляды на классификацию как с точки зрения логистической регрессии, так и с точки зрения персептрона. Практически говоря, их различия заключаются в том, насколько хорошо — для конкретного набора данных — можно оптимизировать любую стоимостную функцию, наряду с (очень часто незначительными) различиями в том, что касается качества границы решения, усваиваемой каждой стоимостной функцией. Разумеется, когда softmax задействуется с точки зрения перцептрона, между перцептроном и логистической регрессией вообще нет никакой качественной разницы.

6.4.4. Функция softmax и линейно разделяемые наборы данных

Представьте себе, что у нас есть набор данных, два класса которого могут быть полностью разделены гиперплоскостью, и мы выбрали соответствующую стоимостную функцию для ее минимизирования, чтобы определить надлежащие веса для нашей модели. Вообразите, что нам *чрезвычайно повезло* и наша *инициализация* $\mathbf{w}^0$ производит линейную границу решения $\dot{\mathbf{x}}^T\mathbf{w}^0 = 0$ с *полным разделением*. Это означает согласно уравнению (6.31), что для каждой из наших P точек мы имеем, что $-y_p\dot{\mathbf{x}}_p^T\mathbf{w}^0 < 0$ и поточечная стоимость перцептрона из уравнения (6.33) равна нулю для каждой точки, т. е. $g_p\left(\mathbf{w}^0\right) = \max\left(0,\ -y_p\dot{\mathbf{x}}_p^T\mathbf{w}^0\right) = 0$, и, следовательно, стоимость перцептрона в уравнении (6.33) *строго равна нулю*.

Поскольку стоимость перцептрона уже равна нулю, своему наименьшему значению, любой алгоритм локальной оптимизации немедленно остановится (т. е. мы никогда не сделаем ни одного оптимизационного шага). Однако это будет не так, если мы использовали одинаковую инициализацию, но вместо персептрона задействовали softmax.

Поскольку мы всегда имеем $e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}^0} > 0$, то поточечная функция softmax всегда является неотрицательной $g_p\left(\mathbf{w}^0\right) = \log\left(1 + e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}^0}\right) > 0$ и, следовательно, функция

softmax тоже. Это означает, что при применении любой схемы локальной оптимизации, такой как, например, градиентный спуск, мы действительно будем делать шаги в сторону, противоположную от инициализации $\mathbf{w}^0$, для того чтобы направлять значение softmax все ниже и ниже к ее минимуму в нуле. На самом деле — с данными, которые действительно линейно разделяемы, — функция мягкого максимума достигает этой нижней границы *только тогда*, когда *магнитуда весов растет до бесконечности*. Это понятно из того факта, что каждый отдельный член $\log\left(1+e^{-C}\right)$ только по мере того, как $C\to\infty$. Действительно, если мы умножим нашу инициализацию $\mathbf{w}^0$ на любую константу $C>1$, то сможем уменьшить значение любого отрицательного экспоненциала с участием одной из наших точек данных, поскольку $e^{-C}<1$, и поэтому $e^{-y_p\mathbf{x}_p^T(C\mathbf{w}^0)}=e^{-C}e^{-y_p\mathbf{x}_p^T\mathbf{w}^0}<e^{-y_p\mathbf{x}_p^T\mathbf{w}^0}$.

Это также уменьшает функцию мягкого максимума, при этом минимум достигается только по мере того, как $C\to\infty$. Однако важно отметить, что независимо от скалярного $C>1$ значения граница решения, определенная начальными весами $\dot{\mathbf{x}}^T\mathbf{w}^0=0$, *не меняет своего местоположения*, т. к. мы по-прежнему имеем, что $C\dot{\mathbf{x}}^T\mathbf{w}^0=0$ (на самом деле, это верно для любого ненулевого скаляра C). Поэтому, хотя расположение разделяющей гиперплоскости не должно изменяться, с функцией мягкого максимума мы по-прежнему будем делать все больше и больше шагов в минимизации, поскольку (в случае полностью линейно разделяемых данных) ее минимум лежит в бесконечности. Этот факт может вызвать серьезные проблемы численной нестабильности со схемами локальной оптимизации, которые делают *большую прогрессию* вперед на каждом шаге — в особенности метод Ньютона (*см. разд. 4.3*), т. к. они будут тяготеть к быстрому расхождению до бесконечности[7].

Пример 6.6. Полностью разделяемые данные и стоимость мягкого максимума

Применяя метод Ньютона для минимизации функции мягкого максимума над полностью линейно разделяемыми данными, легко столкнуться с проблемами численной нестабильности, поскольку глобальный минимум стоимости технически лежит в бесконечности. Мы рассмотрим простой пример такого поведения, используя один входной набор данных, показанный на рис. 6.9. На двух верхних графиках проиллюстрирована прогрессия пяти ньютоновых шагов вначале в точке $\mathbf{w}=\begin{bmatrix}1\\1\end{bmatrix}$.

В пределах пяти шагов мы достигли точки, обеспечивающей очень хорошую подгонку к данным (на рис. 6.9, *слева сверху* построен график подгонки функции $\tanh(\cdot)$ с использованием взгляда с точки зрения логистической регрессии на функцию мягкого максимума), и той, которая уже достаточно велика по магнитуде (см. рис. 6.9, *справа сверху*, где показан контурный график стоимостной функции).

[7] Поскольку стоимости мягкого максимума и перекрестной энтропии эквивалентны (как обсуждалось в предыдущем разделе), эта проблема в равной степени возникает и при использовании стоимости перекрестной энтропии.

На графике справа по траектории шагов, которые движутся линейно в сторону минимума в $\begin{bmatrix} -\infty \\ \infty \end{bmatrix}$, мы видим, что расположение линейной границы решения (здесь точка) не изменяется после первого или двух шагов. Другими словами, после первых нескольких шагов каждый последующий шаг просто умножает своего предшественника на скалярное значение $C > 1$.

Обратите внимание, что если мы просто перевернем одну из меток, сделав этот набор данных не полностью линейно разделяемым, то соответствующая стоимостная функция не будет иметь глобального минимума в бесконечности, как проиллюстрировано на контурном графике на рис. 6.13, *справа снизу*.

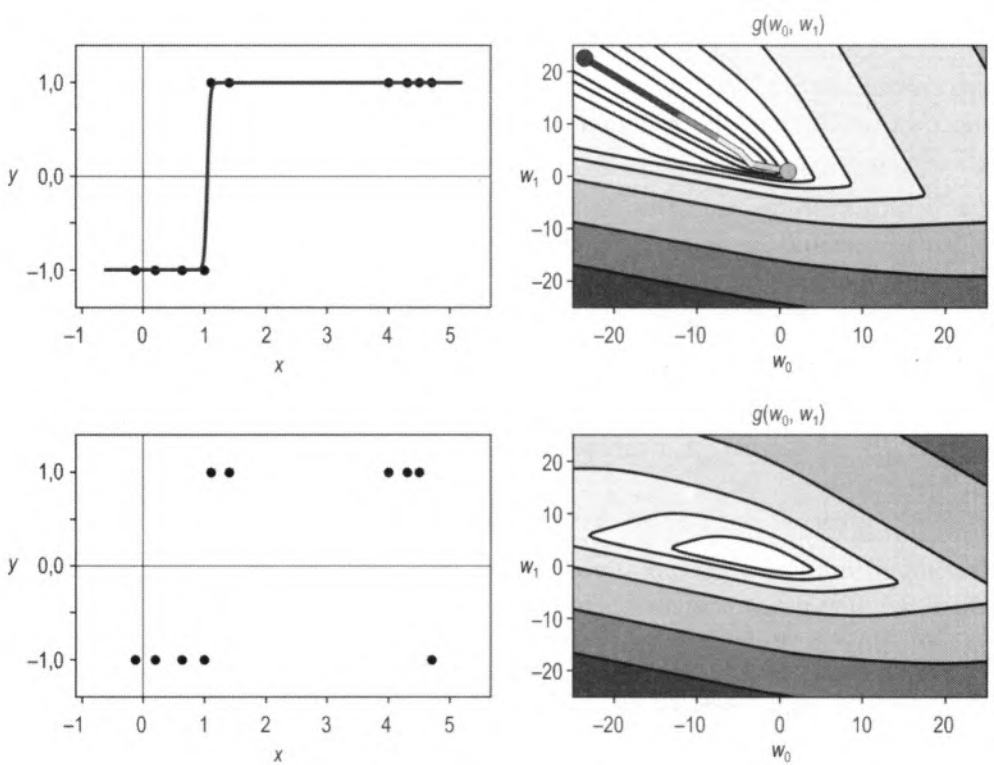


Рис. 6.13. Иллюстрация примера 6.6

6.4.5. Нормализация признаковых весов

Как предотвратить эту потенциальную проблему — весов, усваиваемых посредством локальной оптимизации (и метода Ньютона в частности), взлетающих к бесконечности, — при задействовании softmax/перекрестной энтропии с линейно разделяемыми данными? Простой подход может состоять в более тщательном задействовании наших схем локальной оптимизации: надо делать меньше шагов и/или останавливать оптимизацию, в случае если магнитуда весов становится больше,

чем большая предопределенная константа. Еще один подход заключается в контроле магнитуды весов во время самой процедуры оптимизации. На жаргоне машинного обучения оба подхода обычно называются *стратегиями регуляризации* (подробно описанными в *разд. 11.6*). Первая стратегия (форма того, что называется ранней остановкой) очень проста и требует незначительных корректировок действий по локальной оптимизации, но второй подход обязывает нас дать некоторые объяснения.

Контроль магнитуды множества весов $\mathbf{w}$ означает, что мы хотим контролировать размер $N+1$ индивидуальных весов, которые оно содержит:

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_N \end{bmatrix}. \tag{6.38}$$

Мы можем сделать это, *непосредственно* контролируя размер только N этих весов, и это особенно удобно сделать, используя финальные N признаковых весов w_1, w_2, ..., w_N, поскольку они определяют *нормальный вектор* к линейной границе решения $\dot{\mathbf{x}}^T\mathbf{w}^0 = 0$. Для того чтобы облегчить введение следующих далее геометрических понятий, будем использовать нашу нотацию смещения/признаковых весов для $\mathbf{w}$, впервые введенную в *разд. 5.2.4*. Она предоставляет нам индивидуальную нотацию для смещения и признаковых весов в виде

$$\text{(смещение)}: b = w_0 \qquad \text{(признаковые веса)}: \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{6.39}$$

С помощью этой нотации мы можем выразить линейные границы решения следующим образом:

$$\dot{\mathbf{x}}^T\mathbf{w}^0 = b + \mathbf{x}^T\boldsymbol{\omega} = 0. \tag{6.40}$$

Для того чтобы понять, почему эта нотация полезна, сначала обратите внимание на то, как с геометрической точки зрения признаковые веса $\boldsymbol{\omega}$ определяют *нормальный вектор линейной границы решения. Нормальный вектор* к гиперплоскости (как и наша граница решения) всегда *перпендикулярен* ей (рис. 6.14). Мы всегда можем вычислить ошибку — так называемое знаковое расстояние — точки $\mathbf{x}_p$ до линейной границы решения в терминах нормального вектора $\boldsymbol{\omega}$.

Для того чтобы увидеть, как это возможно, представьте, что у нас есть точка $\mathbf{x}_p$, лежащая "выше" линейной границы решения на трансляте границы решения, где $b + \mathbf{x}^T\boldsymbol{\omega} = \beta > 0$, как показано на рис. 6.14 (аналогично можно рассмотреть случай, если $\mathbf{x}_p$ лежит "ниже" нее). Для того чтобы вычислить расстояние от $\mathbf{x}_p$ до границы решения, представьте, что мы знаем расположение ее *вертикальной проекции* на

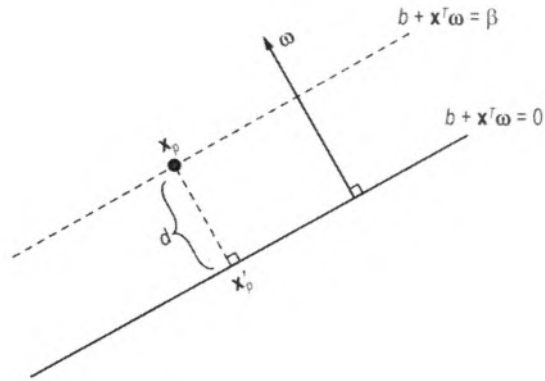


Рис. 6.14. Линейная граница решения, записанная как $b + \mathbf{x}^T\boldsymbol{\omega} = 0$, имеет нормальный вектор $\boldsymbol{\omega}$, определяемый его признаковыми весами. Для того чтобы вычислить знаковое расстояние точки $\mathbf{x}_p$ до границы, мы отмечаем трансляцию границы решения, проходящей через эту точку, как $b + \mathbf{x}^T\boldsymbol{\omega} = \beta$, и строим проекцию точки на границу решения как $\mathbf{x}'_p$

границу решения, которую мы будем называть $\mathbf{x}'_p$. Для того чтобы вычислить желаемую ошибку, мы хотим вычислить знаковое расстояние между $\mathbf{x}_p$ и его вертикальной проекцией, т. е. длину вектора $\mathbf{x}'_p - \mathbf{x}_p$, умноженную на знак β, который здесь равен $+1$, т. к. мы исходим из того, что точка лежит выше границы решения, т. е. $d = \|\mathbf{x}'_p - \mathbf{x}_p\|_2 \operatorname{sign}(\beta) = \|\mathbf{x}'_p - \mathbf{x}_p\|_2$. Теперь, поскольку этот *вектор также перпендикулярен* границе решения (и поэтому параллелен нормальному вектору $\boldsymbol{\omega}$), правило *внутреннего произведения* (*см. разд. П3.2*) дает

$$\left(\mathbf{x}'_p - \mathbf{x}_p\right)^T \boldsymbol{\omega} = -\|\mathbf{x}'_p - \mathbf{x}_p\|_2 \|\boldsymbol{\omega}\|_2 = -d\|\boldsymbol{\omega}\|_2. \tag{6.41}$$

Теперь, если мы возьмем разницу между нашей границей решения и ее трансляцией, вычисленную соответственно в $\mathbf{x}'_p$ и $\mathbf{x}_p$, то получим упрощение:

$$0 - \beta = \left(b + \left(\mathbf{x}'_p\right)^T \boldsymbol{\omega}\right) - \left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right) = \left(\mathbf{x}'_p - \mathbf{x}_p\right)^T \boldsymbol{\omega}. \tag{6.42}$$

Поскольку обе формулы равны $\left(\mathbf{x}'_p - \mathbf{x}_p\right)^T \boldsymbol{\omega}$, мы можем приравнять их друг другу, получив

$$d\|\boldsymbol{\omega}\|_2 = \beta, \tag{6.43}$$

или, другими словами, что знаковое расстояние d от $\mathbf{x}_p$ до границы решения равно

$$d = \frac{\beta}{\|\boldsymbol{\omega}\|_2} = \frac{b + \mathbf{x}_p^T\boldsymbol{\omega}}{\|\boldsymbol{\omega}\|_2}. \tag{6.44}$$

Обратите внимание, что здесь нам не нужно беспокоиться о делении на ноль, т. к. если бы все признаковые веса $\boldsymbol{\omega}$ были равны нулю, то из этого следовало бы, что и смещение было бы равно нулю ($b = 0$) тоже, и у нас вообще нет границы решения.

Кроме того, из этого анализа вытекает, что если признаковые веса имеют *единичную длину* в виде $\|\boldsymbol{\omega}\|_2 = 1$, то знаковое расстояние d от $\mathbf{x}_p$ до границы решения задается *просто ее вычислением* $b + \mathbf{x}_p^T\boldsymbol{\omega}$. Наконец, если бы точка $\mathbf{x}_p$ лежала ниже границы решения и $\beta < 0$, то ничего в окончательных формулах, выведенных выше, не изменилось бы.

Отметим это расстояние от точки до границы решения на рис. 6.15; здесь входная размерность равна $N = 3$, а граница решения является истинной гиперплоскостью.

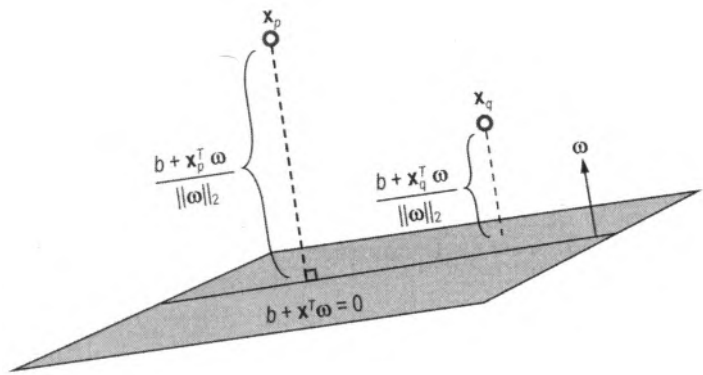


Рис. 6.15. Визуальное представление расстояния до гиперплоскости $b + \mathbf{x}^T\boldsymbol{\omega}$ из двух точек $\mathbf{x}_p$ и $\mathbf{x}_q$, лежащих над ней

Напомним, что мы можем прошкалировать любую линейную границу решения на ненулевой скаляр C, и она по-прежнему будет определять ту же самую гиперплоскость. Поэтому, если, в частности, мы умножим на $C = 1/\|\boldsymbol{\omega}\|_2$, то получим

$$\frac{b + \mathbf{x}^T\boldsymbol{\omega}}{\|\boldsymbol{\omega}\|_2} = \frac{b}{\|\boldsymbol{\omega}\|_2} + \mathbf{x}^T\frac{\boldsymbol{\omega}}{\|\boldsymbol{\omega}\|_2} = 0. \tag{6.45}$$

Мы не меняем характер нашей границы решения, и теперь наши признаковые веса имеют единичную длину в виде $\left\|\boldsymbol{\omega}/\|\boldsymbol{\omega}\|_2\right\|_2 = 1$. Другими словами, независимо от того, насколько велики были наши веса $\mathbf{w}$ изначально, мы всегда можем их единообразно нормализовать, разделив на магнитуду $\boldsymbol{\omega}$.

6.4.6. Регуляризирование двухклассовой классификации

Описанная выше схема нормализации особенно полезна в контексте подчеркнутой выше технической проблемы, связанной с мягким максимумом/перекрестной энтропией, поскольку ясно, что граница решения, которая полностью разделяет два класса данных, *может быть нормализована по признаковым весам*, чтобы не дать ее весам расти слишком много (и расходиться до бесконечности). Конечно, мы не хотим ждать выполнения этой нормализации до того, как наша локальная оптими-

зация будет закончена, поскольку предотвратить магнитуду весов от потенциального расхождения можно только *во время* оптимизации. Мы можем достичь этого, *ограничив* функцию мягкого максимума/перекрестной энтропии так, чтобы признаковые веса всегда имели длину 1, т. е. $\|\boldsymbol{\omega}\|_2 = 1$. Формально эта задача минимизации (при использовании softmax) может быть сформулирована следующим образом:

$$\begin{aligned} &\underset{b,\boldsymbol{\omega}}{\text{минимизировать}} \quad \frac{1}{P}\sum_{p=1}^{P}\log\left(1+e^{-y_p\left(b+\mathbf{x}_p^T\boldsymbol{\omega}\right)}\right) \\ &\text{с учетом, что} \qquad \|\boldsymbol{\omega}\|_2^2 = 1. \end{aligned} \tag{6.46}$$

Решая эту *ограниченную* версию стоимости мягкого максимума, мы по-прежнему можем усваивать границу решения, которая полностью разделяет два класса данных, но мы избегаем расхождения в магнитуде весов, поддерживая их магнитуду *нормализованной по признаковым весам*. Задача в такой формулировке действительно может быть решена с помощью простых расширений методов локальной оптимизации, подробно описанных в *главах 2–4*. Однако более популярный подход в сообществе машинного обучения заключается в том, чтобы "ослабить" эту ограниченную формулировку и вместо этого вычислить тесно связанную с ней неограниченную *регуляризованную* версию функции мягкого максимума. Эта упрощенная форма задачи состоит в минимизировании стоимостной функции, которая представляет собой линейную комбинацию нашей первоначальной функции мягкого максимума и магнитуды признаковых весов

$$g(b,\boldsymbol{\omega}) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1+e^{-y_p\left(b+\mathbf{x}_p^T\boldsymbol{\omega}\right)}\right) + \lambda\|\boldsymbol{\omega}\|_2^2, \tag{6.47}$$

которую мы можем минимизировать, используя любую из знакомых нам схем локальной оптимизации. Здесь член $\|\boldsymbol{\omega}\|_2^2$ называется *регуляризатором*, а параметр $\lambda \geq 0$ — *регуляризационным параметром*. Параметр λ используется для уравновешивания силы давления на тот или иной член при минимизировании их суммы. При минимизировании первого члена (нашей функции мягкого максимума) мы по-прежнему стремимся усвоить превосходную линейную границу решения. Кроме того, минимизируя второй член (магнитуду признаковых весов), мы стимулируем усвоение *малых весов*. Это предотвращает расхождение их магнитуды, т. к. если их размер все-таки начинает расти, то вся наша стоимостная функция из-за этого "страдает" и становится большой. По этой причине на практике значение λ в типичной ситуации выбирают малым (и положительным), хотя некоторая тонкая настройка может оказаться полезной.

Пример 6.7. Регуляризованная стоимость мягкого максимума

Давайте повторим эксперимент, описанный в примере 6.6, но добавим в функцию мягкого максимума регуляризатор с $\lambda = 10^{-3}$, как в уравнении (6.47). На рис. 6.16, *справа* показан контурный график регуляризованной стоимости, и мы видим, что ее глобальный минимум больше не лежит в бесконечности. При этом мы по-прежнему

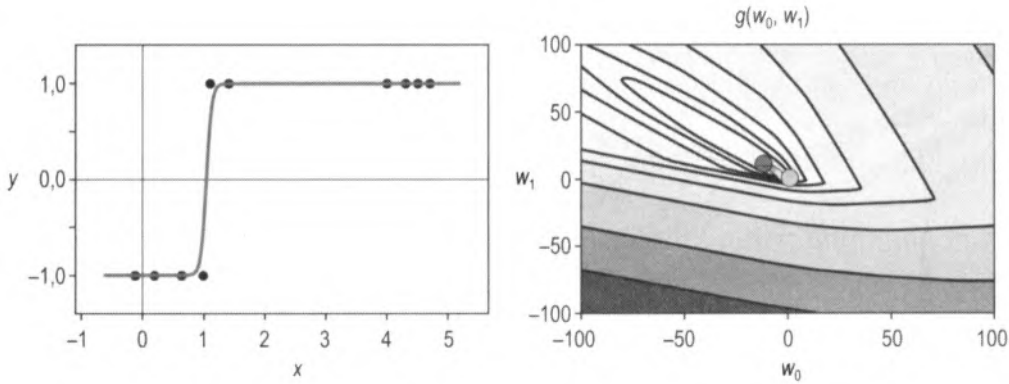


Рис. 6.16. Иллюстрация примера 6.7

усваиваем совершенную границу решения, как показано на рис. 6.16, *слева*, путем плотной подгонки функции $\tanh(\cdot)$.

6.5. Метод опорных векторов

В этом разделе мы опишем *метод опорных векторов* (support vector machines, SVM) [18]. Этот подход дает интересное теоретическое представление о процессе двухклассовой классификации, в особенности в условиях допущения, что данные являются полностью линейно разделяемыми. Однако мы увидим, что в более реалистичном сценарии, когда данные полностью разделить невозможно, подход на основе метода опорных векторов не обеспечивает усвоенной границы решения, которая существенно отличается от тех, что обеспечиваются логистической регрессией или перцептроном.

6.5.1. Маржевой перцептрон

Предположим еще раз, что у нас есть двухклассовый классификационный тренировочный набор данных из P точек $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ с метками $y_p \in \{-1, +1\}$. Также предположим, что мы имеем дело с двухклассовым набором данных, который полностью линейно разделяем с известной линейной границей решения $\dot{\mathbf{x}}^T\mathbf{w} = 0$ (рис. 6.17).

Эта разделяющая гиперплоскость создает буфер между двумя классами, ограниченный двумя равносдвинутыми (равноудаленными) версиями самого себя: одна версия лежит на положительной стороне разделителя и лишь касается класса, имеющего метки $y_p = +1$ (окрашенные в красный цвет), принимая форму $\dot{\mathbf{x}}^T\mathbf{w} = +1$, а другая, лежащая на отрицательной стороне от него, лишь касается класса с метками $y_p = -1$ (окрашенные в синий цвет), принимая форму $\dot{\mathbf{x}}^T\mathbf{w} = -1$. Трансляции выше и ниже разделяющей гиперплоскости в более общем смысле определяются соответственно как $\dot{\mathbf{x}}^T\mathbf{w} = +\beta$ и $\dot{\mathbf{x}}^T\mathbf{w} = -\beta$, где $\beta > 0$. Однако, разделив β в обоих

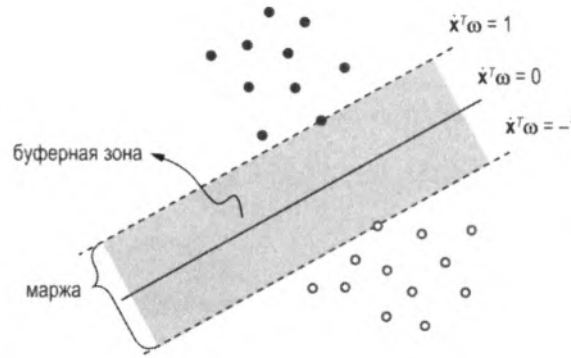


Рис. 6.17. Для линейно разделяемых данных ширина буферной зоны (окрашенной в серый цвет), ограниченной двумя равнорасположенными транслятами разделяющей гиперплоскости, которые лишь касаются каждого соответствующего класса, определяет маржу (или зазор, полосу) этой разделяющей гиперплоскости

уравнениях и переназначив переменные как $\mathbf{w} \leftarrow \mathbf{w}/\beta$, мы можем исключить избыточный параметр β и получить две трансляции в указанной форме $\dot{\mathbf{x}}^T\mathbf{w} = \pm 1$.

Тот факт, что все точки в классе $+1$ лежат строго на $\dot{\mathbf{x}}^T\mathbf{w} = +1$ или на положительной стороне от этой гиперплоскости, а все точки в классе -1 лежат строго на или на отрицательной стороне от $\dot{\mathbf{x}}^T\mathbf{w} = -1$, можно формально записать в виде следующих ниже условий:

$$\begin{aligned} \dot{\mathbf{x}}_p^T\mathbf{w} &\geq 1, \quad \text{если } y_p = +1; \\ \dot{\mathbf{x}}_p^T\mathbf{w} &\leq -1, \text{ если } y_p = -1, \end{aligned} \tag{6.48}$$

что является обобщением условий, приведших к стоимости перцептрона в уравнении (6.30).

Мы можем совместить эти условия в одно утверждение, умножив каждое из них на соответствующие значения меток, получив одно-единственное неравенство $y_p\dot{\mathbf{x}}_p^T\mathbf{w} \geq 1$, которое можно эквивалентно записать как поточечную функцию:

$$g_p(\mathbf{w}) = \max\left(0,\ 1 - y_p\dot{\mathbf{x}}_p^T\mathbf{w}\right) = 0. \tag{6.49}$$

Опять же, это значение всегда является неотрицательным. Взятие среднего арифметического всех P уравнений приведенной выше формы дает функцию маржевого персептрона (margin-perceptron):

$$g_p(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\max\left(0,\ 1 - y_p\dot{\mathbf{x}}_p^T\mathbf{w}\right). \tag{6.50}$$

Обратите внимание на поразительное сходство между первоначальной функцией перцептрона из предыдущего раздела и приведенной выше стоимостью маржевого персептрона: в каждом слагаемом мы просто наивно "добавили 1" в ненулевой вход функции max. Однако эта дополнительная 1 предотвращает обсуждавшуюся ранее

проблему тривиального нулевого решения, присущую первоначальному персептрону, которая здесь просто не возникает.

Если данные на самом деле полностью линейно разделяемы, то любая гиперплоскость, проходящая между двумя классами, будет иметь параметры $\mathbf{w}$, где $g(\mathbf{w}) = 0$. Однако маржевой персептрон по-прежнему является валидной стоимостной функцией, даже если данные невозможно линейно разделить. Единственная разница заключается в том, что с таким набором данных мы не можем заставить приведенные выше критерии соблюдаться для всех точек в наборе данных. Отсюда нарушение для p-й точки добавляет в стоимостную функцию положительное значение $1 - y_p \dot{\mathbf{x}}_p^T \mathbf{w}$.

6.5.2. Связь с функцией мягкого максимума

Как и в случае с персептроном, мы можем сгладить маржевой персептрон, заменив оператор max на softmax (*см. разд. 6.4.3*). Сделав это в одном из слагаемых маржевого персептрона, мы получаем родственное слагаемое

$$\text{soft}\left(0,\ 1 - y_p \dot{\mathbf{x}}_p^T \mathbf{w}\right) = \log\left(1 + e^{1 - y_p \dot{\mathbf{x}}_p^T \mathbf{w}}\right). \tag{6.51}$$

Сразу же отметим: если бы мы просуммировали все P, то мы могли бы сформировать стоимостную функцию наподобие мягкого максимума, которая близко совпадает с маржевым персептроном. Но обратите внимание, что в процедуре выведения приведенного выше маржевого персептрона использованная "1" в компоненте $1 - y_p \dot{\mathbf{x}}_p^T \mathbf{w}$ стоимости могла бы быть любым числом, которое мы пожелаем. И действительно, мы выбрали значение "1" из соображений удобства. Вместо этого мы могли бы выбрать любое значение $\varepsilon > 0$, в таком случае множество P условий, сформулированных в уравнении (6.48), будет эквивалентным образом сформулировано как

$$\max\left(0,\ \varepsilon - y_p \dot{\mathbf{x}}_p^T \mathbf{w}\right) = 0 \tag{6.52}$$

для всех p, и маржевой персептрон эквивалентным образом сформулирован как

$$g(\mathbf{w}) = \frac{1}{P} \sum_{p=1}^{P} \max\left(0,\ \varepsilon - y_p \dot{\mathbf{x}}_p^T \mathbf{w}\right), \tag{6.53}$$

и, наконец, softmax-версия одного слагаемого здесь будет

$$\text{soft}\left(0,\ \varepsilon - y_p \dot{\mathbf{x}}_p^T \mathbf{w}\right) = \log\left(1 + e^{\varepsilon - y_p \dot{\mathbf{x}}_p^T \mathbf{w}}\right). \tag{6.54}$$

Когда ε является совсем малым, у нас, разумеется, есть $\log\left(1 + e^{\varepsilon - y_p \dot{\mathbf{x}}_p^T \mathbf{w}}\right) \approx$ $\approx \log\left(1 + e^{-y_p \dot{\mathbf{x}}_p^T \mathbf{w}}\right)$, т. е. то же самое слагаемое, которое используется в функции мягкого максимума. Отсюда мы можем интерпретировать стоимостную функцию мягкого максимума как сглаженную версию нашей функции маржевого персептрона.

6.5.3. Границы решения с максимальной маржой

Когда два класса данных являются полностью линейно разделяемыми, данные полностью разделяются бесконечно большим числом гиперплоскостей. На рис. 6.18 показаны две такие гиперплоскости для прототипического полностью разделяемого набора данных. С учетом того, что оба классификатора (а также любая другая граница решения, полностью разделяющая данные) будут полностью классифицировать этот набор данных, можно ли хотя бы про один из них сказать, что он является "наилучшей" из всех возможных разделяющих гиперплоскостей?

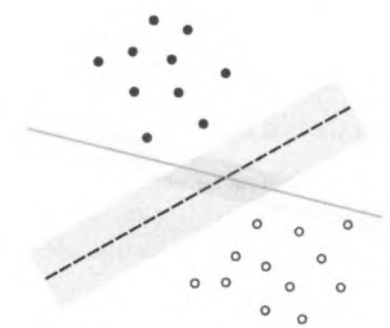


Рис. 6.18. Набор данных может полностью разделяться бесконечно большим числом линейных границ решения, как показано здесь, где две линейные границы решения показаны зеленым и черным цветом. Граница решения с максимальной маржой (черная штриховая линия) интуитивно является наилучшим выбором

Один из разумных стандартов для суждения о качестве этих гиперплоскостей основывается на длинах их маржей, т. е. расстоянии между равноотстоящими транслятами, которые только касаются каждого класса. Чем больше это расстояние, тем интуитивно лучше ассоциированная гиперплоскость разделяет все пространство с учетом конкретного распределения данных. Эта идея наглядно проиллюстрирована на рис. 6.18, на котором показаны два разделителя вместе с соответствующими маржами (зазорами). Оба разделителя проводят полное различие между двумя классами, зеленый разделитель (с меньшей маржой) делит пространство довольно неудобным образом, если учесть то, как данные распределены, и поэтому будет легче тяготеть к неправильной классификации будущих точек данных.

С другой стороны, черный разделитель (имеющий бóльшую маржу) делит пространство ровнее по отношению к конкретным данным и будет тяготеть к более точному классифицированию будущих точек.

В нашей попытке восстановить максимальную маржу, разделяющую границу решения, будет удобно использовать нашу индивидуальную нотацию для смещения и признаковых весов (используемую, например, в *разд. 6.4.5*):

$$(\text{смещение}):\ b = w_0 \qquad (\text{признаковые веса}):\ \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{6.55}$$

С помощью этой нотации мы можем выразить линейную границу решения как

$$\dot{\mathbf{x}}^T\mathbf{w} = b + \mathbf{x}^T\boldsymbol{\omega} = 0\,. \tag{6.56}$$

Для того чтобы отыскать разделяющую гиперплоскость с максимальной маржой, мы стремимся отыскать такой набор параметров, что участок, определяемый выражением $b + \mathbf{x}^T\boldsymbol{\omega} = \pm 1$, при котором каждый транслят лишь касается одного из двух классов, имеет максимально возможную маржу. Как показано на рис. 6.19, маржа может быть определена путем вычисления расстояния между любыми двумя точками (по одной от каждой транслированной гиперплоскости), лежащими на нормальном векторе $\boldsymbol{\omega}$. Обозначив через $\mathbf{x}_1$ и $\mathbf{x}_2$ точки на векторе $\boldsymbol{\omega}$, принадлежащие соответственно положительно и отрицательно транслированным гиперплоскостям, маржа вычисляется просто как длина отрезка прямой, соединяющего $\mathbf{x}_1$ и $\mathbf{x}_2$, т. е. $\|\mathbf{x}_1 - \mathbf{x}_2\|_2$.

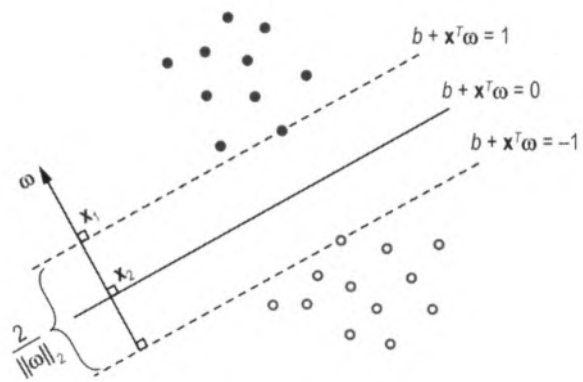


Рис. 6.19. Маржа разделяющей гиперплоскости может быть рассчитана путем измерения расстояния между двумя точками пересечения нормального вектора $\boldsymbol{\omega}$ и двумя равноудаленными трансляциями гиперплоскости. Можно показать, что это расстояние имеет значение $2/\|\boldsymbol{\omega}\|_2$

Маржа может быть записана гораздо удобнее, если взять разницу между двумя транслятами, вычисленными соответственно в $\mathbf{x}_1$ и $\mathbf{x}_2$, как

$$\left(w_0 + \mathbf{x}_1^T\mathbf{w}\right) - \left(w_0 + \mathbf{x}_2^T\mathbf{w}\right) = \left(\mathbf{x}_1 - \mathbf{x}_2\right)^T\boldsymbol{\omega} = 2\,. \tag{6.57}$$

Используя правило внутреннего произведения[8] и тот факт, что два вектора $\mathbf{x}_1 - \mathbf{x}_2$ и $\boldsymbol{\omega}$ параллельны друг другу, мы можем найти решение для маржи непосредственно в терминах $\boldsymbol{\omega}$ как

$$\|\mathbf{x}_1 - \mathbf{x}_2\|_2 = \frac{2}{\|\boldsymbol{\omega}\|_2}\,. \tag{6.58}$$

---

[8] Используя правило внутреннего произведения (*см. разд. П3.2*), из уравнения (6.57) мы имеем, что $2 = \left(\mathbf{x}_1 - \mathbf{x}_2\right)^T\boldsymbol{\omega} = \|\mathbf{x}_1 - \mathbf{x}_2\|_2\,\|\boldsymbol{\omega}\|_2$. Перестановка дает выражение, приведенное в уравнении (6.58).

Поэтому отыскание разделяющей гиперплоскости с максимальной маржой равносильно отысканию гиперплоскости с *наименьшим* возможным нормальным вектором $\boldsymbol{\omega}$.

6.5.4. Задачи о жесткомаржевых и мягкомаржевых SVM-машинах

Для того чтобы отыскать разделяющую гиперплоскость для данных с нормальным вектором минимальной длины, мы можем просто объединить наше желание минимизировать $\|\boldsymbol{\omega}\|_2^2$ с учетом ограничения (определяемого маржевым персептроном), что гиперплоскость полностью разделяет данные (задается описанным выше критерием маржи). Это приводит к так называемой задаче о *жесткомаржевых машинах* опорных векторов:

$$\begin{aligned} &\underset{b,\boldsymbol{\omega}}{\text{минимизировать}} \quad \|\boldsymbol{\omega}\|_2^2 \\ &\text{с учетом, что} \qquad \max\left(0, 1 - y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)\right) = 0, \quad p = 1, \ldots, P. \end{aligned} \tag{6.59}$$

Здесь ограничения гарантируют, что гиперплоскость, которую мы восстанавливаем, полностью разделяет данные. В то время как существуют алгоритмы *ограниченной оптимизации*, которые сконструированы для решения подобных задач, как указано выше (см., например, [14, 15, 19]), мы также можем решить жесткомаржевую задачу, *ослабив* ограничения и предложив неограниченную формулировку задачи (к которой мы можем применить знакомые алгоритмические методы минимизации). Именно такой подход к *регуляризации* был подробно описан в *разд. 6.4.6*. Для этого мы просто вводим ограничения, формируя единую стоимостную функцию

$$g(b, \boldsymbol{\omega}) = \frac{1}{P}\sum_{p=1}^{P}\max\left(0,\ 1 - y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)\right) + \lambda\|\boldsymbol{\omega}\|_2^2, \tag{6.60}$$

подлежащую минимизированию. Здесь параметр $\lambda \geq 0$ называется *штрафом* или *регуляризационным параметром* (как мы видели ранее в *разд. 6.4.6*). Когда λ устанавливается равным малому положительному значению, мы оказываем бóльшее "давление" на стоимостную функцию с целью обеспечения того, чтобы ограничения действительно соблюдались, и (теоретически) когда λ делается очень малым, приведенная выше формулировка совпадает с исходной ограниченной формой. По этой причине на практике λ часто устанавливают довольно малым (оптимальным значением является то, которое приводит к удовлетворению первоначальных ограничений уравнения (6.59)).

Эта *регуляризованная* форма стоимостной функции маржевого персептрона называется *стоимостью мягкомаржевых машин опорных векторов*[9].

---

[9] На практике используются и другие ослабления задачи жесткомаржевых SVM (см., например, [20]), однако они не имеют теоретического или практического преимущества перед представленной здесь [21, 22].

Пример 6.8. Граница решения на основе SVM

На рис. 6.20, *слева* представлены три границы, усваиваемые посредством минимизирования стоимости маржевого персептрона (показанной в уравнении (6.50)) трижды с разными случайными инициализациями. На рис. 6.20, *справа* продемонстрирована граница решения, обеспечиваемая надлежащим минимизированием стоимости мягкомаржевой SVM из уравнения (6.60) с регуляризационным параметром $\lambda = 10^{-3}$.

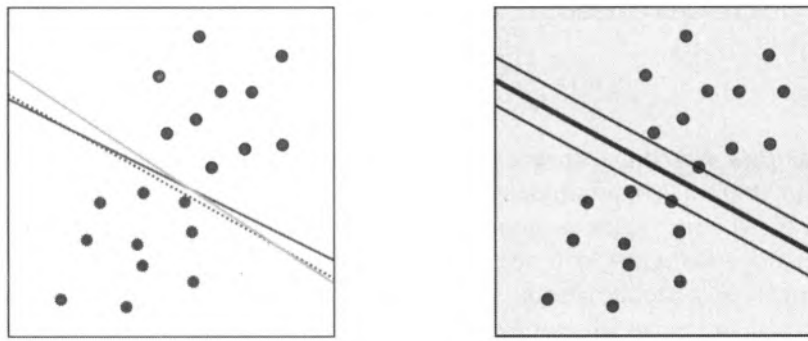


Рис. 6.20. Иллюстрация примера 6.8

Каждая граница, показанная на рисунке слева, полностью разделяет эти два класса, но граница решения на основе SVM (на рисунке справа) обеспечивает максимальную маржу. Обратите внимание, как на правом рисунке трансляты границы решения проходят через точки из обоих классов — равноудаленно от линейной SVM-границы решения. Эти точки называются *опорными векторами*, отсюда и название метода — машины опорных векторов (иногда также именуемые *опорно-векторными* машинами).

6.5.5. SVM-машины и шумные данные

Очень существенное практическое преимущество задачи о мягкомаржевых SVM в уравнении (6.60) заключается в том, что она позволяет нам справляться с шумными неполностью (линейно) разделяемыми данными, которые возникают гораздо чаще на практике, чем наборы полностью линейно разделяемых данных. В то время как "шум" делает невозможным удовлетворение по крайней мере одного из ограничений в жесткомаржевой задаче в уравнении (6.59) (и, следовательно, эта задача технически невозможна для решения), мягкомаржевое ослабление всегда может быть надлежаще минимизировано и поэтому гораздо чаще используется на практике.

6.6. Какой подход дает наилучшие результаты?

Как только мы отказываемся от сильного (и нереалистичного) допущения о полной (линейной) разделяемости, добавочная ценность "гиперплоскости с максимальной маржой", обеспечиваемая решением на основе SVM, исчезает, поскольку изначаль-

но *у нас больше нет маржи*. Отсюда следует, что для многих наборов данных на практике мягкомаржевая задача не обеспечивает решения, заметно отличающегося от персептрона или логистической регрессии. Действительно, с наборами данных, которые не являются линейно разделяемыми, оно часто возвращает *точно такое же* решение, которое предоставляется персептроном или логистической регрессией.

Кроме того, давайте возьмем задачу о мягкомаржевых SVM-машинах в уравнении (6.60) и *сгладим* часть стоимости, относящуюся к маржевому персептрону, с помощью *функции мягкого максимума*, как подробно описано в *разд. 6.5.2*. Это приведет к сглаженной стоимостной функции мягкомаржевых SVM-машин в форме

$$g(b, \boldsymbol{\omega}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)}\right) + \lambda\|\boldsymbol{\omega}\|_2^2 . \qquad (6.61)$$

Хотя эта форма интерпретируется через призму SVM-машин, она также может быть немедленно идентифицирована как регуляризованная стоимость мягкого максимума (т. е. как регуляризованный персептрон или логистическая регрессия[10]). Следовательно, мы видим, что все три метода линейной двухклассовой классификации, которые мы рассмотрели, — логистическая регрессия, персептрон и SVM-машины — очень тесно связаны, и поэтому они тяготеют к тому, чтобы давать похожие результаты на реалистичных (нелинейно разделяемых) наборах данных.

6.7. Категориальная стоимость перекрестной энтропии

В *разд. 6.2* и *6.3* мы видели, как два разных варианта значений меток, $y_p \in \{0, 1\}$ или $y_p \in \{-1, +1\}$, приводят к схожей двухклассовой классификации посредством минимизации стоимостной функции перекрестной энтропии/мягкого максимума. В каждом случае мы формировали стоимость *логарифмической ошибки* в расчете на точку данных, а их усреднение над всеми P точками данных обеспечивало надлежащую и выпуклую стоимостную функцию. Другими словами, числовые значения самих пар меток главным образом использовались только для *упрощения* выражения этих стоимостных функций. Получив шаблон выведения выпуклых стоимостных функций для логистической регрессии, имея любые два *числовых* значения меток $y_p \in \{a, b\}$, было бы легко вывести соответствующую выпуклую стоимостную функцию, основанную на логарифмической ошибке, такой как поточечная стоимость.

Однако истинный диапазон выбора значений меток шире, чем два *числовых* значения. В качестве меток мы можем использовать *любые два четко различимых объекта*, т. е. два неупорядоченных значения. Однако, независимо от того, как мы определяем наши метки, мы все равно в конечном итоге строим тот же самый

[10] Действительно, эта стоимость мягкомаржевых SVM-машин даже иногда упоминается как log-loss SVM, т. е. SVM-машины на основе логарифмической потери (см., например, [21]).

двухклассовый классификатор, который мы видели ранее — настраивая его веса путем минимизации знакомой стоимостной функции, такой как, например, стоимость перекрестной энтропии/мягкого максимума.

Для того чтобы прояснить этот момент, в этом кратком разделе мы покажем, как выводить ту же самую стоимостную функцию перекрестной энтропии, что и в *разд. 6.2*, задействуя *категориальные метки* вместо числовых. Это приводит к процедуре выведения так называемой *категориальной стоимостной функции перекрестной энтропии*, которая эквивалентна стоимости перекрестной энтропии.

6.7.1. Категориальные метки, кодированные с одним активным состоянием

Предположим, что мы начинаем с двухклассового классификационного набора данных $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ с N-мерным входом и преобразуем наши исходные числовые значения меток $y_p \in \{0, 1\}$ с помощью *векторов с одним активным состоянием* в форме

$$y_p = 0 \quad \longleftarrow \quad \mathbf{y}_p = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad y_p = 1 \quad \longleftarrow \quad \mathbf{y}_p = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{6.62}$$

Каждое векторное представление однозначно идентифицирует соответствующее ему значение метки, но теперь эти значения меток больше не являются упорядоченными числовыми значениями, и наш набор данных принимает форму $\left\{\left(\mathbf{x}_p, \mathbf{y}_p\right)\right\}_{p=1}^{P}$, где $\mathbf{y}_p$ определено, как указано выше. Однако наша цель здесь останется прежней: правильно настроить множество из $N+1$ весов $\mathbf{w}$ так, чтобы регрессировать входы в выходы нашего набора данных.

6.7.2. Выбор нелинейности

С этими новыми *категориальными метками* наша задача классификации — при ее рассмотрении в качестве задачи регрессии — является частным случаем *многовыходной регрессии* (*см. разд. 5.6*), в которой мы стремимся регрессировать N-мерный вход относительно двумерных категориальных меток, используя нелинейную функцию линейной комбинации $\dot{\mathbf{x}}_p^T\mathbf{w}$. Поскольку наши категориальные метки имеют длину два, нам нужно использовать нелинейную функцию этой линейной комбинации, которая также дает *два* выхода. Поскольку метки кодированы с одним активным состоянием и мы знакомы с сигмоидной функцией (*см. разд. 6.2.3*), то имеет смысл использовать следующую ниже нелинейную функцию каждой входной точки $\mathbf{x}_p$:

$$\boldsymbol{\sigma}_p = \begin{bmatrix} \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) \\ 1 - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) \end{bmatrix}. \tag{6.63}$$

Почему? Потому что предположим для конкретной точки, что $\mathbf{y}_p = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ и веса $\mathbf{w}$ настроены так, что $\sigma\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right) \approx 1$. По определению сигмоиды из этого следует, что $1 - \sigma\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right) \approx 0$ и для этой точки $\boldsymbol{\sigma}_p \approx \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \mathbf{y}_p$, что и является поистине нашим желанием. И, конечно, эта же идея справедлива и в том случае, если $\mathbf{y}_p = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

Отсюда при таком нелинейном преобразовании идеальная конфигурация наших весов $\mathbf{w}$ будет вынуждать

$$\boldsymbol{\sigma}_p \approx \mathbf{y}_p \tag{6.64}$$

соблюдаться для как можно большего числа точек.

6.7.3. Выбор стоимостной функции

Как и в случае с числовыми метками, здесь мы могли бы также вполне предложить стандартную поточечную стоимость, взятую из нашего опыта, подчерпнутого из регрессии, такую как, например, наименьшие квадраты

$$g_p(\mathbf{w}) = \left\| \boldsymbol{\sigma}_p - \mathbf{y}_p \right\|_2^2, \tag{6.65}$$

и минимизировать их среднее арифметическое над всеми P точками, чтобы настроить веса $\mathbf{w}$. Однако, как и в случае с числовыми метками, поскольку здесь наши *категориальные метки* принимают очень строгую двоичную форму, нам лучше задействовать логарифмическую ошибку (*см. разд. 6.2*), чтобы лучше простимулировать усвоение (производя выпуклую стоимостную функцию). Обозначив через $\log \boldsymbol{\sigma}_p$ вектор, образованный взятием $\log(\cdot)$ каждого элемента $\boldsymbol{\sigma}_p$, получим логарифмическую ошибку в форме

$$g_p(\mathbf{w}) = -\mathbf{y}_p^T \log \boldsymbol{\sigma}_p = -y_{p,1} \log\left(\sigma\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right)\right) - y_{p,2} \log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right)\right), \tag{6.66}$$

где $\mathbf{y}_p = \begin{bmatrix} y_{p,1} \\ y_{p,2} \end{bmatrix}$. Взятие среднего значения этих P поточечных стоимостей дает так называемую категориальную *стоимостную функцию перекрестной энтропии* для двухклассовой классификации. Здесь слово "категориальная" относится к тому факту, что наши метки представляют собой категориальные (т. е. неупорядоченные) векторы, кодированные с одним активным состоянием.

Однако нетрудно увидеть, что стоимостная функция является именно той логарифмической ошибкой, которую мы нашли в *разд. 6.2*. Другими словами, записанная в терминах наших оригинальных *числовых* меток, указанная выше поточечная стоимость равна в точности

$$g_p(\mathbf{w}) = -y_p \log\left(\sigma\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right)\right) - \left(1 - y_p\right) \log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right)\right). \tag{6.67}$$

Следовательно, несмотря на то, что мы задействовали категориальные версии наших исходных числовых значений меток, стоимостная функция, которую мы минимизируем в конце, чтобы надлежаще настроить $\mathbf{w}$, точно такая же, как мы видели ранее — стоимость перекрестной энтропии/мягкого максимума, где задействовались числовые метки.

6.8. Метрики качества классификации

В этом разделе мы опишем простые метрики для суждения о качестве натренированной модели двухклассовой классификации, а также то, как делать предсказания, используя одну из них.

6.8.1. Выполнение предсказаний, используя натренированную модель

Если мы обозначим через $\mathbf{w}^\star$ оптимальное множество весов, найденное путем минимизирования классификационной стоимостной функции, по умолчанию задействуя значения меток $y_p \in \{-1, +1\}$, то заметим, что мы можем записать нашу полностью настроенную линейную модель как

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = \mathbf{x}^T\mathbf{w}^\star = w_0^\star + x_1 w_1^\star + x_2 w_2^\star + \ldots + x_N w_N^\star . \tag{6.68}$$

Эта полностью натренированная модель определяет оптимальную границу решения для тренировочного набора данных, которая принимает форму

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = 0 . \tag{6.69}$$

Для того чтобы предсказать метку y' входа $\mathbf{x}'$, мы обрабатываем эту модель с помощью соответствующего шага. Поскольку мы по умолчанию используем значения меток $y_p \in \{-1, +1\}$, эта шаговая функция удобно определяется знаковой

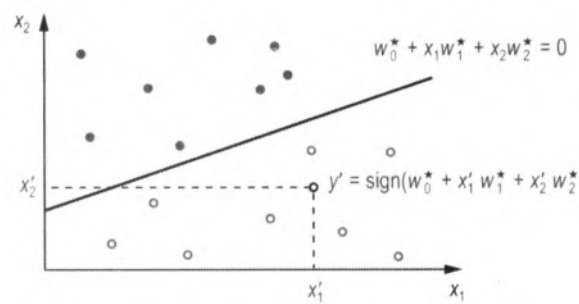


Рис. 6.21. После того как была усвоена граница решения для тренировочного набора данных с оптимальными параметрами $w_0^\star$ и $\mathbf{w}^\star$, метка y новой точки $\mathbf{x}$ может быть предсказана просто проверкой того, по какую сторону границы она лежит. Здесь точка $\mathbf{x}$ лежит ниже усвоенной гиперплоскости, и в результате ей дается метка $\text{sign}\left(\dot{\mathbf{x}}'^T\mathbf{w}^\star\right) = -1$

функцией $\text{sign}(\cdot)$ (как подробно описано в *разд. 6.3*), а предсказываемая метка для $\mathbf{x}'$ задается как

$$\text{sign}\left(\text{model}\left(\mathbf{x}', \mathbf{w}^\star\right)\right) = \mathbf{y}' . \tag{6.70}$$

Это вычисление, которое всегда будет принимать значения ± 1, если $\mathbf{x}'$ не лежит строго на границе решения (в этом случае мы задаем случайное значение из ± 1), просто показывает то, на какой стороне границы решения находится вход $\mathbf{x}'$. Если он лежит "выше" границы решения, то $y' = +1$, а если "ниже", то $y' = -1$. Это проиллюстрировано для набора данных на рис. 6.21.

6.8.2. Оценивание уверенности

После того как усвоена надлежащая граница решения, мы можем выразить *суждение* об уверенности в отношении нее в любой точке, основываясь на *расстоянии этой точки до границы решения*. Мы говорим, что наш классификатор дает нулевую уверенность в точках, лежащих вдоль самой границы решения, потому что граница не может точно сказать нам о том, к какому классу такие точки принадлежат (именно поэтому значение метки им задается случайно, в случае если нам когда-либо понадобится сделать там предсказание). Точно так же мы говорим, что *вблизи* границы решения мы *доверяем* предсказаниям классификатора *мало*. Почему? Представьте себе, что мы прикладываем небольшое возмущение к границе решения, слегка меняя ее местоположение. Некоторые точки, *очень близкие к исходной границе*, окажутся на противоположной стороне новой границы и, следовательно, будут иметь *другие предсказанные метки*. И наоборот, именно поэтому мы имеем высокую уверенность в предсказанных метках для точек, расположенных вдалеке от границы решения. Эти предсказанные метки никак не изменятся, в случае если внести небольшое изменение в расположение границы решения.

Понятие "уверенности" можно сделать точным и нормализованным для универсальной применимости путем измерения расстояния точки до границы посредством *сигмоидной функции* (*см. разд. 6.2.3*). В результате мы получим уверенность в том, что точка принадлежит классу $+1$.

Знаковое расстояние d точки до границы решения, обеспечиваемое нашей натренированной моделью, может быть вычислено (*см. разд. 6.4.5*) как

$$d = \frac{b^\star + \mathbf{x}_p^T \boldsymbol{\omega}^\star}{\left\| \boldsymbol{\omega}^\star \right\|_2}, \tag{6.71}$$

где

$$\text{(смещение)}: b^\star = w_0^\star \quad \text{(признаковые веса)}: \boldsymbol{\omega}^\star = \begin{bmatrix} w_1^\star \\ w_2^\star \\ \vdots \\ w_N^\star \end{bmatrix}. \tag{6.72}$$

Вычислив d с помощью *сигмоидной функции*, мы гладко сплющиваем ее в интервал [0; 1], получая балл "уверенности":

$$\text{уверенность в предсказанной метке для точки } \mathbf{x} = \sigma(d). \tag{6.73}$$

Когда это значение равно 0,5, точка лежит на самой границе. Если значение больше 0,5, то точка лежит на положительной стороне границы решения, и поэтому мы имеем бóльшую уверенность в том, что ее предсказанная метка равна +1. Когда значение меньше 0,5, точка лежит на *отрицательной* стороне классификатора, и поэтому у нас меньше уверенности в том, что она действительно имеет значение +1 метки. Поскольку нормализация с задействованием сигмоиды сжимает $(-\infty; +\infty)$ в интервал [0; 1], это значение уверенности часто интерпретируется как *вероятность*.

6.8.3. Суждение о качестве натренированной модели с помощью метрики точности

После того как мы успешно минимизировали стоимостную функцию для линейной двухклассовой классификации, определение качества нашей натренированной модели может оказаться деликатным делом. Простейшая метрика для суждения о качестве полностью натренированной модели состоит в подсчете *числа неправильных классифицирований*, которые она формирует в нашем тренировочном наборе данных. Указанная метрика представляет собой сырое число точек $\mathbf{x}_p$ тренировочных данных, истинная метка y_p которых предсказывается нашей натренированной моделью неправильно.

Для того чтобы сравнить предсказанную метку $\hat{y}_p = \text{sign}\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}^\star\right)\right)$ и истинную метку y_p, можно применить индикаторную функцию $\mathcal{I}(\cdot)$ и вычислить

$$\mathcal{I}\left(\hat{y}_p, y_p\right) = \begin{cases} 0, & \text{если } \hat{y}_p = y_p; \\ 1, & \text{если } \hat{y}_p \neq y_p. \end{cases} \tag{6.74}$$

Суммирование всех P точек дает общее число неправильных классифицирований в нашей натренированной модели:

$$\text{число неправильных классифицирований} = \sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p, y_p\right). \tag{6.75}$$

Используя эту формулу, мы также можем вычислить *точность* $\mathcal{A}$ натренированной модели. Это просто процент тренировочного набора данных, метки которого предсказаны моделью правильно, т. е.

$$\mathcal{A} = 1 - \frac{1}{P}\sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p, y_p\right). \tag{6.76}$$

Точность варьируется от 0 (ни одна точка не классифицирована правильно) до 1 (все точки классифицированы правильно).

Пример 6.9. Сравнение историй стоимостной функции и неправильных классифицирований

Наши классификационные стоимостные функции в конечном счете основаны на гладких аппроксимациях дискретной шаговой функции (как подробно описано в *разд. 6.2* и *6.3*). Это именно та функция, которую мы действительно хотим использовать, т. е. функция, с помощью которой мы на самом деле хотим настроить параметры нашей модели. Однако, поскольку мы не можем оптимизировать эту параметризованную шаговую функцию непосредственно, мы соглашаемся на ее гладкую аппроксимацию. Последствия этого практического выбора видны, когда мы сравниваем историю стоимостной функции из прогона градиентного спуска с соответствующим числом неправильных классифицирований, измеряемых на каждом шаге прогона. На рис. 6.22 иллюстрируется такое сравнение с помощью набора данных, показанного на рис. 6.10. Фактически мы демонстрируем результаты трех независимых прогонов градиентного спуска, причем история неправильных классифицирований показана на графике слева, а соответствующие истории стоимости мягкого максимума — справа. Каждый прогон кодирован цветом, чтобы отличить его от других прогонов.

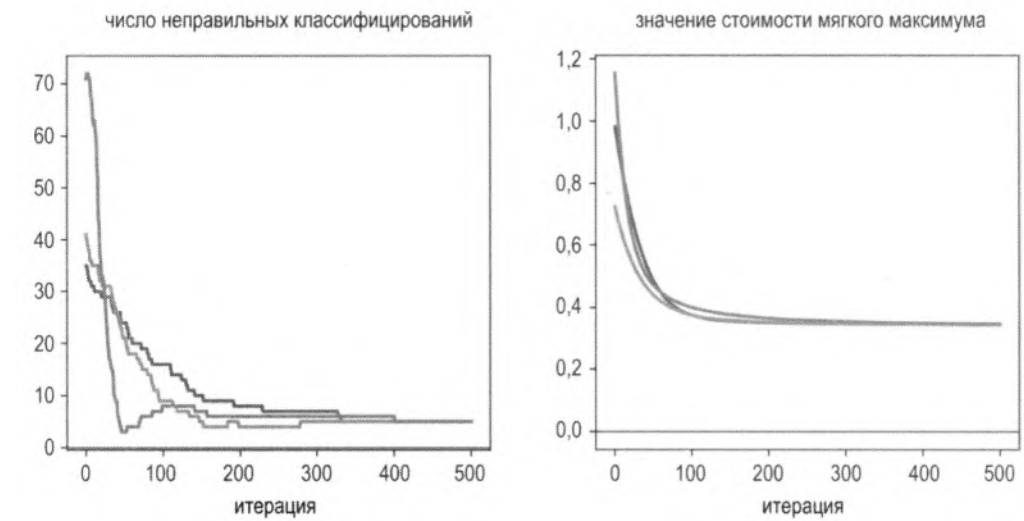


Рис. 6.22. Иллюстрация примера 6.9

Сравнивая графики на рис. 6.22, *слева* и *справа*, мы видим, что число неправильных классифицирований и вычислений функции мягкого максимума на каждом шаге градиентного спуска повторяют друг друга не полностью. То есть это не тот случай, когда при уменьшении значения стоимостной функции то же самое происходит и с числом неправильных классифицирований.

Это происходит потому, что наша функция мягкого максимума является лишь аппроксимацией истинной величины, которую мы хотели бы минимизировать.

Представленный простой пример имеет чрезвычайно важное практическое значение: после выполнения локальной оптимизации с целью минимизирования стоимо-

стной функции для двухклассовой классификации наилучший шаг (тот, который обеспечивает наилучший результат классификации) и соответствующие веса связаны с наименьшим числом *неправильных классифицирований* (либо, что аналогично, с самой *высокой точностью*), а не с наименьшим значением стоимостной функции.

6.8.4. Суждение о качестве натренированной модели с помощью метрики сбалансированной точности

Точность классифицирования — это превосходная базовая мера результативности натренированного классификатора. Однако в некоторых сценариях использование метрики точности может нарисовать неполную картину того, насколько хорошо мы действительно решили задачу классификации. Например, когда набор данных состоит из *сильно несбалансированных классов*, т. е. когда набор данных содержит гораздо больше примеров одного класса, чем другого, "точность" натренированной модели теряет свою ценность как метрика качества. Это обусловлено тем, что, когда один класс значительно превосходит другой в наборе данных, значение точности, близкое к 1, может ввести в заблуждение. Например, если один класс составляет 95% всех точек данных, то наивный классификатор, который слепо назначает метку мажоритарного класса (т. е. класса большинства) *каждой тренировочной точке*, достигает точности 95%. Но здесь неправильное классифицирование с точностью 5% сводится к *неправильному классифицированию всего класса данных целиком*.

Эта идея "жертвовать" членами меньшего класса путем неправильного их классифицирования (вместо членов из мажоритарного класса) — в зависимости от приложения — очень нежелательна. Например, проводя различие между здоровыми людьми и страдающими редким заболеванием (миноритарный класс), скорее можно неправильно классифицировать здорового человека (члена мажоритарного класса) и назначить ему дополнительное тестирование, чем упустить возможность правильно определить того, кто действительно болен. А вот еще пример: при решении задачи обнаружения финансового мошенничества (рассмотренной в примере 1.9) можно в большей степени согласиться с неправильно классифицированными допустимыми транзакциями, чем с необнаруженными мошенническими, поскольку первые, как правило, можно легко устранить путем подробной инспекции человеком (например, предложив клиенту выполнить перепроверку на возможное наличие мошеннических транзакций).

Такого рода сценарии указывают на проблему с использованием *точности* в качестве надлежащей метрики для диагностики результативности классификатора на наборах данных с сильно несбалансированными классами: поскольку она одинаково взвешивает *неправильные классифицирования из обоих классов*, она не может передать то, насколько хорошо натренированная модель работает с каждым классом данных в отдельности. Это приводит к потенциальной возможности получить высокую результативность на очень крупном классе данных, маскируя низкую результативность на очень малом. Самый простой способ улучшить метрику точности — принять во внимание эту потенциальную проблему и вместо вычисления точности над *обоими классами вместе* вычислить точность по *каждому классу в отдельности и усреднить результаты*.

Если мы обозначим *индексы* этих точек с метками $y_p = +1$ и $y_p = -1$ соответственно через Ω_{+1} и Ω_{-1}, то можем вычислить число неправильных классифицирований на каждом классе в отдельности (задействуя введенную выше нотацию и индикаторную функцию) как

$$\begin{aligned} &\text{число неправильных классифицирований на классе } +1 = \sum_{p\in\Omega_{+1}} \mathcal{I}\left(\hat{y}_p, y_p\right); \\ &\text{число неправильных классифицирований на классе } -1 = \sum_{p\in\Omega_{-1}} \mathcal{I}\left(\hat{y}_p, y_p\right). \end{aligned} \tag{6.77}$$

Точность на каждом классе в отдельности затем можно вычислить схожим образом как (обозначив точность на классе $+1$ и -1 в отдельности соответственно как $\mathcal{A}_{+1}$ и $\mathcal{A}_{-1}$):

$$\begin{aligned} \mathcal{A}_{+1} &= 1 - \frac{1}{|\Omega_{+1}|}\sum_{p\in\Omega_{+1}} \mathcal{I}\left(\hat{y}_p, y_p\right); \\ \mathcal{A}_{-1} &= 1 - \frac{1}{|\Omega_{-1}|}\sum_{p\in\Omega_{-1}} \mathcal{I}\left(\hat{y}_p, y_p\right). \end{aligned} \tag{6.78}$$

Отметим здесь, что $|\Omega_{+1}|$ и $|\Omega_{-1}|$ обозначают число точек, принадлежащих соответственно классу $+1$ и -1. Затем мы можем объединить эти две метрики в одно значение, взяв их *среднее значение*. Эта комбинированная метрика называется *сбалансированной точностью* (которую мы обозначаем как $\mathcal{A}_{\text{баланс}}$):

$$\mathcal{A}_{\text{баланс}} = \frac{\mathcal{A}_{+1} + \mathcal{A}_{-1}}{2}. \tag{6.79}$$

Обратите внимание, что если оба класса имеют равное представление, то сбалансированная точность сводится к значению совокупной точности $\mathcal{A}$.

Метрика сбалансированной точности варьируется от 0 до 1. Когда она равна 0, ни одна точка не классифицирована правильно, а когда оба класса классифицированы совершенно, $\mathcal{A}_{\text{баланс}} = 1$. Значения метрики посередине 0 и 1 измеряют то, насколько хорошо — в среднем — каждый класс классифицируется в отдельности. Если, например, один класс данных классифицирован абсолютно правильно, а другой — абсолютно неправильно (как в нашем воображаемом сценарии, где мы имеем несбалансированный набор данных с 95-процентной принадлежностью к одному классу и 5-процентной принадлежностью к другому и где мы просто классифицировали все пространство как мажоритарный класс), то $\mathcal{A}_{\text{баланс}} = 0{,}5$.

Таким образом, сбалансированная точность — это простая метрика, помогающая нам понять, "вела ли себя плохо" наша усвоенная модель на сильно несбалансированных наборах данных (см. пример 6.12) или нет. Для того чтобы *улучшить поведение* нашей усвоенной модели в таких случаях, мы должны скорректировать способ, которым мы выполняем двухклассовое классифицирование. Один из популярных способов сделать это — так называемая взвешенная классификация (*см. разд. 6.9*).

6.8.5. Матрица путаницы и дополнительные метрики качества

Дополнительные метрики для суждения о качестве натренированной модели для двухклассовой классификации можно сформировать с помощью *матрицы путаницы* (рис. 6.23). Матрица путаницы — это простая поисковая таблица, в которой результаты классифицирования разбиваются на фактическое (по строкам) и предсказываемое (по столбцам) членство в классе. Здесь мы обозначаем через A число точек данных, фактическая метка которых, $+1$, идентична метке, назначаемой им натренированным классификатором.



Рис. 6.23. Матрица путаницы может использоваться для получения дополнительных метрик качества для двухклассовой классификации

Другой диагональный элемент D схожим образом определяется как число точек данных, чей предсказанный класс, -1, равен их фактическому классу. Смещенные по диагонали элементы, обозначенные буквами B и C, представляют собой два типа ошибок классификации, при которых фактические и предсказанные метки не совпадают друг с другом. На практике мы хотим, чтобы эти две величины были как можно меньше.

Наша метрика *точности* может быть выражена в терминах, показанных на рисунке значений матрицы путаницы, как

$$\mathcal{A} = \frac{A + D}{A + B + C + D}, \tag{6.80}$$

и наша точность на каждом отдельном классе схожим образом, как:

$$\begin{aligned} \mathcal{A}_{+1} &= \frac{A}{A + B}; \\ \mathcal{A}_{-1} &= \frac{D}{C + D}. \end{aligned} \tag{6.81}$$

На жаргоне машинного обучения эти индивидуальные метрики точности часто называются соответственно *чувствительностью* ($\mathcal{A}_{+1}$) и *специфичностью* ($\mathcal{A}_{-1}$).

Метрика *сбалансированной точности* также может быть выражена как

$$\mathcal{A}_{\text{баланс}} = \frac{1}{2}\frac{A}{A + B} + \frac{1}{2}\frac{D}{C + D}. \tag{6.82}$$

Пример 6.10. Обнаружение спама

В этом примере мы выполняем двухклассовое классифицирование на популярном наборе данных для обнаружения спама (см. пример 1.9), взятом из [23]. Этот набор данных состоит из $P = 4601$ точки данных, 1813 спамных писем и 2788 неспамных писем, причем каждая точка данных состоит из различных входных признаков электронного письма (подробнее описанных в примере 9.2), а также двоичной метки, указывающей, является ли электронное письмо спамом или нет. Правильно минимизируя функцию мягкого максимума, мы можем достичь 93-процентной точности над всем набором данных наряду со следующей ниже матрицей путаницы:

		Предсказано	
		не-спам	спам
Фактически	не-спам	285	15
	спам	234	466

Пример 6.11. Кредитная проверка

В этом примере мы экзаменуем двухклассовый классификационный набор данных, состоящий из $P = 1000$ экземпляров, каждый из которых представляет собой набор статистик, извлеченных из заявок на получение кредита в немецкий банк (взято из [24]). С каждым входом связана соответствующая метка: *хороший* кредитный риск (700 примеров), либо *плохой* (300 примеров), по определению специалистов из финансово-кредитного отдела. В процессе усвоения классификатора для этого набора данных мы создаем автоматический инструмент оценки кредитного риска, который способен помогать принимать решения о том, являются ли будущие заявители надежными кандидатами на получение кредитов или нет.

Двадцатимерные ($N = 20$) входные признаки в этом наборе данных включают: баланс текущего счета физического лица в банке (признак 1), продолжительность (в месяцах) предыдущего кредита в банке (признак 2), платежный статус любого предыдущего кредита, полученного в банке (признак 3), и текущую стоимость его сбережений/акций (признак 6). Надлежаще минимизировав стоимость персептрона, мы можем достичь 75-процентной точности над всем набором данных наряду со следующей ниже матрицей путаницы:

		Предсказано	
		плохой	хороший
Фактически	плохой	285	15
	хороший	234	466

6.9. Взвешенная двухклассовая классификация

Поскольку наши стоимостные функции двухклассовой классификации *суммируются над отдельными точками*, мы можем — как мы сделали с регрессией в *разд. 5.5* — взвешивать отдельные точки, чтобы преувеличивать или преуменьшать их важность для классификационной модели. Это называется *взвешенной классификацией*. Указанная идея часто реализуется при работе с *сильно несбалансированными* двухклассовыми наборами данных (*см. разд. 6.8.4*).

6.9.1. Взвешенная двухклассовая классификация

Как и в случае с регрессией из *разд. 5.5*, стоимостные функции для взвешенной классификации естественным образом возникают из-за повторяющихся точек в наборе данных. Например, с помощью наборов метаданных (скажем, данных переписи) нередко можно получить повторяющиеся точки данных из-за того, что несколько человек дают одинаковые ответы при опросе.

Для рис. 6.24 взят стандартный набор данных переписи и построен график его подмножества вдоль одного входного признака. Принимая в расчет только один признак, в итоге мы получаем многочисленные записи одной и той же точки данных, которые показаны визуально посредством радиуса каждой точки (чем больше раз указанная точка данных появляется в наборе данных, тем больший радиус кружка на графике). Эти точки данных не следует отбрасывать — они не возникли из-за какой-то ошибки в процессе сбора данных. Они представляют собой истинный набор данных.

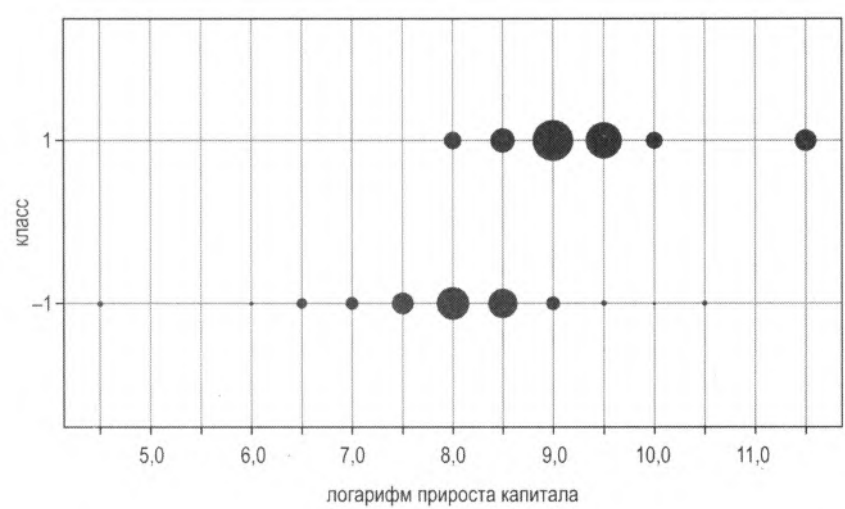


Рис. 6.24. Пример повторяющихся записей в наборе метаданных. Здесь построен график одного входного признака этого набора данных вместе с метками для каждой точки. В рассматриваемом срезе данных имеется много дубликатов, которые выражены визуально радиусом каждой точки (чем больше радиус, тем больше дубликатов этой точки существует в наборе данных)

Как и в случае с регрессионной стоимостью, если мы рассмотрим любую функцию двухклассовой классификации, то она "сколлапсируется", а слагаемые, содержащие идентичные точки, естественным образом объединятся во взвешенные члены. Это можно увидеть, выполнив то же самое простое упражнение, которое использовалось в *разд. 6.24* с целью иллюстрации этого факта для регрессии. Это приводит к понятию *взвешивания* двухклассовых стоимостных функций, таких как, например, взвешенная стоимость мягкого максимума, которую мы приводим ниже, задействуя обобщенную model-нотацию, использованную в *разд. 5.4*, для представления нашей линейной модели:

$$g(\mathbf{w}) == \frac{1}{\beta_1 + \beta_2 + ... + \beta_P} \sum_{p=1}^{P} \beta_p \log\left(1 + e^{-y_p \text{model}(x_p, \mathbf{w})}\right). \tag{6.83}$$

Здесь значения β_1, β_2, ..., β_P — это фиксированные *поточечные веса*. То есть уникальная точка (x_p, y_p) в наборе данных имеет вес $\beta_p = 1$, тогда как при повторе этой точки в наборе данных R раз один ее экземпляр будет иметь вес $\beta_p = R$, а другие — вес $\beta_p = 0$. Поскольку эти веса фиксированы (т. е. они не являются параметрами, которые нужно настраивать, подобно $\mathbf{w}$), мы можем минимизировать стоимость взвешенной классификации точно так же, как и любую другую, с помощью схемы локальной оптимизации, такой как градиентный спуск или метод Ньютона.

6.9.2. Взвешивание точек по уровню уверенности

Как и в случае с регрессией (*см. разд. 5.5*), мы можем думать о *присвоении* фиксированных весовых значений в уравнении (6.83) на основе нашей "уверенности" в легитимности точки данных. Если мы считаем, что точка является очень надежной, мы можем установить соответствующий ей вес β_p ближе к 1, и чем ненадежнее мы считаем точку, тем меньше мы устанавливаем β_p в интервале $0 \leq \beta_p \leq 1$, где из $\beta_p = 0$ следует, что мы не доверяем точке вообще. При выборе этих весов мы, конечно, задаем то, насколько важной является каждая точка данных в процессе тренировки модели.

6.9.3. Устранение классовых дисбалансов

Взвешенная классификация — в том виде, в котором было описано выше, — часто используется для работы с несбалансированными наборами данных. Такие наборы данных содержат гораздо больше примеров одного класса данных, чем другого. И с ними часто легко достичь высокой точности, неправильно классифицируя точки из меньшего класса (*см. разд. 6.8.4*).

Одним из способов решения этой проблемы является использование стоимости взвешенной классификации с целью изменения поведения усвоенного классификатора таким образом, чтобы он взвешивал точки в меньшем классе больше, а точки в большем классе меньше. Для получения такого результата принято назначать такие

веса *обратно пропорционально числу членов каждого класса*. В результате члены мажоритарного и миноритарного классов взвешиваются так, что (в совокупности) каждый вносит равный вклад во взвешенную классификацию.

То есть если мы обозначим через Ω_{+1} и Ω_{-1} индексные множества для точек соответственно в классах $+1$ и -1, то сначала отметим, что $P = |\Omega_{+1}| + |\Omega_{-1}|$. Затем, обозначив через β_{+1} и β_{-1} вес каждого члена класса соответственно $+1$ и -1, мы можем установить эти классовые веса обратно пропорциональными числу точек в каждом классе как:

$$\beta_{+1} \propto \frac{1}{|\Omega_{+1}|}; \quad \beta_{-1} \propto \frac{1}{|\Omega_{-1}|}. \tag{6.84}$$

Пример 6.12. Классовый дисбаланс и взвешенная классификация

На рис. 6.25, *слева* показан набор данных с серьезным классовым дисбалансом. Здесь также построена линейная граница решения, полученная в результате минимизирования функции softmax для этого набора данных с использованием пяти шагов метода Ньютона и окрашиванием каждого участка пространства в зависимости от того, как этот натренированный классификатор помечает точки. Здесь всего три (в общей сложности 55) неправильно классифицированные точки (одна синяя и две красных — что дает точность, близкую к 95%); однако те точки, которые классифицированы неправильно, составляют почти половину миноритарного (красного) класса. Хотя данный факт не отражается в грубой метрике неправильного классифицирования или точности, это отражается в сбалансированной точности (*см. разд. 6.8.4*), которая значительно ниже и составляет около 79%.

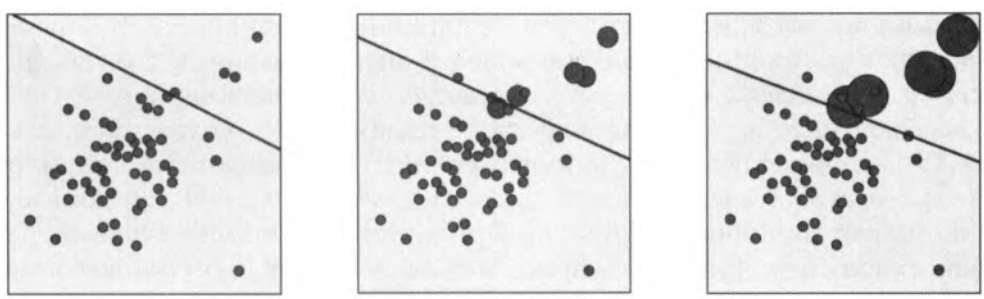


Рис. 6.25. Иллюстрация примера 6.12

На рис. 6.25, *в центре* и *справа* показан результат увеличения весов каждого члена миноритарного класса с $\beta = 1$ до $\beta = 5$ (в центре) и $\beta = 10$ (справа). Эти веса визуально обозначены на рисунке увеличением радиуса точек пропорционально используемому значению β (поэтому их радиус увеличивается слева направо). Также на рисунках в центре и справа отображен результат надлежащего минимизирования

взвешенной функции мягкого максимума из уравнения (6.83) с использованием той же самой процедуры оптимизации (т. е. пяти шагов метода Ньютона). Поскольку значение β для миноритарного класса возрастает, мы поощряем меньшее число неправильных классифицирований его членов (здесь за счет дополнительных неправильных классификацирований мажоритарного класса). На рис. 6.25, *справа* (где $\beta = 10$) мы имеем еще одно неправильное классифицирование, чем в исходном прогоне с точностью 93%. Однако, если исходить из допущения, что неправильное классифицирование членов миноритарного класса гораздо опаснее, чем неправильное классифицирование членов мажоритарного класса, то здесь компромисс является вполне оправданным, поскольку никакие члены миноритарного класса не классифицируются неправильно. Кроме того, мы достигаем значительно *улучшенного балла сбалансированной точности*, равного 96%, по сравнению с 79%, достигнутыми при первоначальном (невзвешенном) прогоне.

6.10. Заключение

В этой главе мы подробно рассмотрели задачу линейной двухклассовой классификации, в которой стремимся проводить автоматическое различие между разными типами четко различимых объектов с использованием линейной границы решения. Мы увидели широкий спектр точек зрения на обрамление классификации в вычислительный каркас, причем каждая из этих точек зрения проливает уникальный свет на сам процесс.

Мы начали с обсуждения регрессионного взгляда на классификацию в *разд. 6.2* и *6.3*, где была описана логистическая регрессия, а затем подробно описали способы рассмотрения задачи "сверху" с точки зрения *персептрона* и *опорно-векторных машин*, приведенных соответственно в *разд. 6.4* и *6.5*. Хотя эти несопоставимые точки зрения различаются в том, как они объясняют происхождение двухклассовой классификации, мы видели, как все они естественным образом приводят к минимизации одного и того же вида стоимостной функции (например, функции *перекрестной энтропийной* или *мягкого максимума*) с использованием одной и той же линейной модели. Эта унифицирующая реализация — обсуждаемая далее в *разд. 7.4* — помогает объяснить, почему (в среднем) эти различные точки зрения на двухклассовую классификацию приводят к сходным результатам на практике (на основе метрик, введенных в *разд. 6.8*). Практически это делает различные точки зрения, стоимостные функции и маркировочные схемы для двухклассовой классификации (включая категориальные маркировочные схемы, как подробно описано в *разд. 6.8*) существенно взаимозаменяемыми, допуская одну большую гибкость в концептуализации и конструировании двухклассовых классификаторов. В последующих главах мы будем в значительной степени опираться на эту идею при наслоении новых концепций, для простоты вводя их в контексте одной точки зрения (например, персептрона) на двухклассовую классификацию, в то же время неявно признавая, что такие новые концепции автоматически применимы и ко всем другим точкам зрения.

6.11. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:
github.com/jermwatt/machine_learning_refined.

6.1. Имплементирование сигмоидальной функции наименьших квадратов

Повторить эксперимент, описанный в примере 6.3, кодируя стоимостную функцию наименьших квадратов, показанную в уравнении (6.9), и алгоритм нормализованного градиентного спуска, подробно описанный в *разд. П1.3*. От вас не требуется воспроизводить контурный график, показанный на рис. 6.5, *справа*, однако вы можете перепроверить, что ваша имплементация работает надлежаще, воспроизведя окончательную подгонку, показанную на рис. 6.5, *слева*. В качестве альтернативы показать, что ваш окончательный результат дает нулевые неправильные классифицирования (*см. разд. 7.6.3*).

6.2. Показать эквивалентность логарифмической ошибки и поточечной стоимости перекрестной энтропии

Показать, что (со значениями меток $y_p \in \{0; 1\}$) логарифмическая ошибка из уравнения (6.10) эквивалентна поточечной стоимости перекрестной энтропии из уравнения (6.11).

6.3. Реализация функции перекрестной энтропии

Повторить эксперимент, описанный в примере 6.4, закодировав стоимостную функцию перекрестной энтропии, показанную в уравнении (6.12), как подробно описано в *разд. 6.2.7*. От вас не требуется воспроизводить контурный график, показанный на рис. 6.7, *справа*; однако вы можете убедиться, что ваша имплементация работает надлежаще, воспроизведя окончательную подгонку, показанную на рис. 6.7, *слева*. В качестве альтернативы показать, что ваш окончательный результат дает нулевые неправильные классифицирования (*см. разд. 7.6.3*).

6.4. Вычислить липшицеву константу стоимости перекрестной энтропии

Вычислить константу Липшица (*см. разд. П1.6.4*) стоимости перекрестной энтропии, показанной в уравнении (6.12).

6.5. Подтвердить расчеты градиента и гессиана

Подтвердить, что градиент и гессиан стоимости перекрестной энтропии показаны такими, какими они приведены в *разд. 6.2.7*.

6.6. Показать эквивалентность логарифмической ошибки и поточечной функции мягкого максимума

Показать, что (со значениями меток $y_p \in \{-1; +1\}$) логарифмическая ошибка из уравнения (6.22) эквивалентна поточечной функции мягкого максимума из уравнения (6.24).

6.7. Имплементирование стоимости мягкого максимума

Повторить эксперимент, описанный в примере 6.5, кодируя стоимостную функцию мягкого максимума, показанную в уравнении (6.25). От вас не требуется воспроизводить контурный график, приведенный на рис. 6.9, *справа*, однако вы можете перепроверить, что ваша имплементация работает надлежаще, воспроизведя окончательную подгонку, показанную на рис. 6.9, *слева*. В качестве альтернативы показать, что ваш окончательный результат дает нулевые неправильные классифицирования (*см. разд. 7.6.3*).

6.8. Реализация функции softmax с помощью логарифмической ошибки

Повторить эксперимент, описанный в *разд. 6.3.3* и показанный на рис. 6.10, используя логарифмическую ошибку, основываясь на функции мягкого максимума из уравнения (6.21), и любую схему локальной оптимизации, которую вы пожелаете. От вас не требуется воспроизводить графики, показанные на рисунке, чтобы подтвердить надлежащую работу вашей реализации, но вы должны получить как минимум пять неправильных классифицирований (*см. разд. 7.6.3*).

6.9. Использование градиентного спуска для минимизирования стоимости персептрона

Использовать схему стандартного градиентного спуска, чтобы минимизировать стоимостную функцию персептрона из уравнения (6.33) над набором данных, показанным на рис. 6.10. Сделать два прогона градиентного спуска, используя фиксированные значения длины шага $\alpha = 10^{-1}$ и $\alpha = 10^{-2}$, с 50 шагами каждый (и случайными инициализациями). Построить графики истории стоимостной функции и истории неправильного классифицирования (*см. разд. 7.6.3*) на каждом шаге прогона. Какой прогон достигает совершенной классификации первым?

6.10. Стоимость персептрона является выпуклой

Показать, что стоимость персептрона, приведенная в уравнении (6.33), является выпуклой, используя определение выпуклости нулевого порядка, описанное в упражнении 5.8.

6.11. Стоимость мягкого максимума является выпуклой

Показать, что стоимостная функция мягкого максимума, приведенная в уравнении (6.25), является выпуклой, проверив, что она удовлетворяет определению выпуклости второго порядка. Подсказка: гессиан, уже приведенный в уравнении (6.28), представляет собой взвешенную матрицу внешних произведений, аналогичную матрице, описанной в упражнении 4.2.

6.12. Регуляризированная функция мягкого максимума

Повторить эксперимент, описанный в примере 6.7 и показанный на рис. 6.16. От вас не требуется воспроизводить графики, приведенные на рисунке, чтобы подтвердить надлежащую работу вашей имплементации, но вы должны достичь результата, который имеет пять или меньше неправильных классифицирований.

6.13. Сравнение эффективности двухклассовых стоимостных функций (вариант I)

Сравнить эффективность стоимостных функций мягкого максимума и персептрона с точки зрения минимального числа неправильных классифицирований, которое каждый из них может достичь при надлежащей минимизации посредством градиентного спуска на наборе данных по раку молочной железы. Этот набор данных состоит из $P = 699$ точек данных, каждая точка имеет $N = 9$ входных атрибутов одного человека и выходную метку, указывающую на то, есть ли у конкретной пациентки рак молочной железы или нет. Вы должны достичь около 20 неправильных классифицирований с каждым методом.

6.14. Сравнение эффективности двухклассовых стоимостных функций (вариант II)

Сравнить эффективность стоимостных функций мягкого максимума и персептрона с точки зрения минимального числа неправильных классифицирований, которые каждый из них может достичь при надлежащей минимизации посредством градиентного спуска на наборе данных обнаружения спама, представленном в примере 6.10. Вы должны достичь по крайней мере 90-процентной точности с обоими методами. Из-за больших вариаций во входных значениях этого набора данных вы должны стандартно нормализовать входные признаки набора данных (*см. разд. 9.3*) перед началом оптимизации.

6.15. Кредитная проверка

Повторить эксперимент, описанный в примере 6.11. Используя оптимизатор по вашему выбору, постараться достичь результата, близкого к полученным. Обеспечить обязательную стандартную нормализацию входных признаков набора данных (*см. разд. 9.3*) перед оптимизацией.

6.16. Взвешенная классификация и сбалансированная точность

Повторить эксперимент, описанный в примере 6.12 и показанный на рис. 6.25. От вас не требуется воспроизводить графики, представленные на рисунке, чтобы подтвердить надлежащую работу вашей имплементации, но вы должны достичь аналогичных результатов.

7 Линейная многоклассовая классификация

7.1. Введение

На практике много задач классификации (например, распознавание лиц, жестов рук, произносимых фраз или слов и т. д.) имеют больше, чем просто два класса, которые мы хотим различать. Однако одна-единственная линейная граница решения естественным образом делит входное пространство только на два подпространства и поэтому принципиально недостаточна как механизм дифференциации между более чем двумя классами данных. В этой главе мы опишем, как обобщить то, что мы увидели в предыдущей главе, чтобы иметь возможность работать с этой многоклассовой окружающей обстановкой, именуемой линейной *многоклассовой классификацией*. Как и в предыдущих главах, мы опишем несколько точек зрения на многоклассовую классификацию, включая принцип *"один против всех"* и *многоклассовый перцептрон*, метрики качества и расширения базовых принципов (включая категориальную и взвешенную многоклассовую классификацию). Здесь мы также представим мини-пакетную оптимизацию — впервые подробно описанную в *разд. П1.5* в контексте первопорядковой оптимизации — через призму машинного обучения, подытожив ее использование для оптимизации как контролируемых, так и неконтролируемых учеников.

7.2. Многоклассовая классификация "один против всех"

В этом разделе мы приведем объяснение фундаментальной схемы многоклассовой классификации, именуемой *"один против всех"*, шаг за шагом используя один набор данных с тремя классами.

7.2.1. Нотация и моделирование

Многоклассовый классификационный набор данных $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ состоит из C четко различимых классов данных. Как и в случае с двумя классами, теоретически мы можем использовать *любое* множество из C четко различимых значений меток для различения классов. Для удобства здесь мы используем значения меток $y_p \in \{0, 1, ..., C-1\}$. В дальнейшем мы задействуем набор данных, показанный на рис. 7.1, который поможет нам вывести нашу фундаментальную схему многоклассовой классификации.

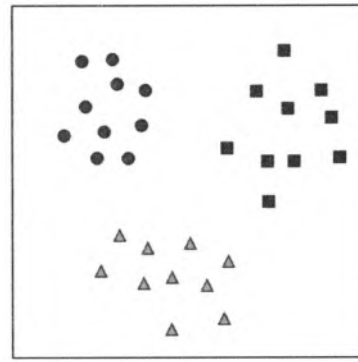


Рис. 7.1. Классификационный набор данных с $C = 3$ классами. Здесь точки со значением меток $y_p = 0$, $y_p = 1$ и $y_p = 2$ окрашены соответственно в синий, красный и зеленый цвета

7.2.2. Тренировка C классификаторов "один против всех"

Хорошим первым шагом в развитии многоклассовой классификации является упрощение задачи до того, с чем мы уже знакомы: *двухклассовой классификации*. Мы уже знаем, как отличить каждый класс наших данных от других $C-1$ классов. Такого рода задача двухклассовой классификации (рассмотренная в предыдущей главе) действительно является проще и примыкает к реальной задаче, которую мы хотим решить, т. е. усвоить классификатор, который может различать все классы одновременно. Для того чтобы решить эту задачу, мы усваиваем C двухклассовых классификаторов над всем набором данных, причем c-й классификатор натренирован отличать c-й класс от остальных данных и, следовательно, именуется классификатором *"один против остальных"* или *"один против всех"*.

Для того чтобы решить c-ю подзадачу двухклассовой классификации, мы просто назначаем временные метки $\tilde{y}_p$ всему набору данных, давая метки $+1$ c-му классу и метки -1 остальной части набора данных:

$$\tilde{y}_p = \begin{cases} +1, & \text{если } y_p = c; \\ -1, & \text{если } y_p \neq c, \end{cases} \tag{7.1}$$

где y_p — это исходная метка для p-й точки многоклассового набора данных. Затем мы выполняем избранную схему двухклассовой классификации (минимизируя любую классификационную функцию потерь, подробно описанную в предыдущей главе). Обозначив оптимальные веса, усвоенные c-м классификатором через $\mathbf{w}_c$, где

$$\mathbf{w}_c = \begin{bmatrix} w_{0,c} \\ w_{1,c} \\ w_{2,c} \\ \vdots \\ w_{N,c} \end{bmatrix}, \tag{7.2}$$

мы можем выразить соответствующую границу решения, связанную с c-й двухклассовой классификацией (отличающей класс c от всех других точек данных), просто как

$$\dot{\mathbf{x}}^T \mathbf{w}_c = 0. \tag{7.3}$$

На рис. 7.2 представлен результат решения этого набора подзадач на прототипическом наборе данных, изначально показанном на рис. 7.1. В этом случае мы усваиваем $C = 3$ четко различимых линейных двухклассовых классификатора, которые показаны в верхнем ряду рисунка. Каждая усвоенная граница решения иллюстрируется цветом отдельного класса, отличающимся от остальных данных, причем в каждом случае остальные данные окрашиваются в серый цвет. В нижнем ряду рисунка набор данных показан еще раз вместе со всеми тремя усвоенными двухклассовыми границами решения.

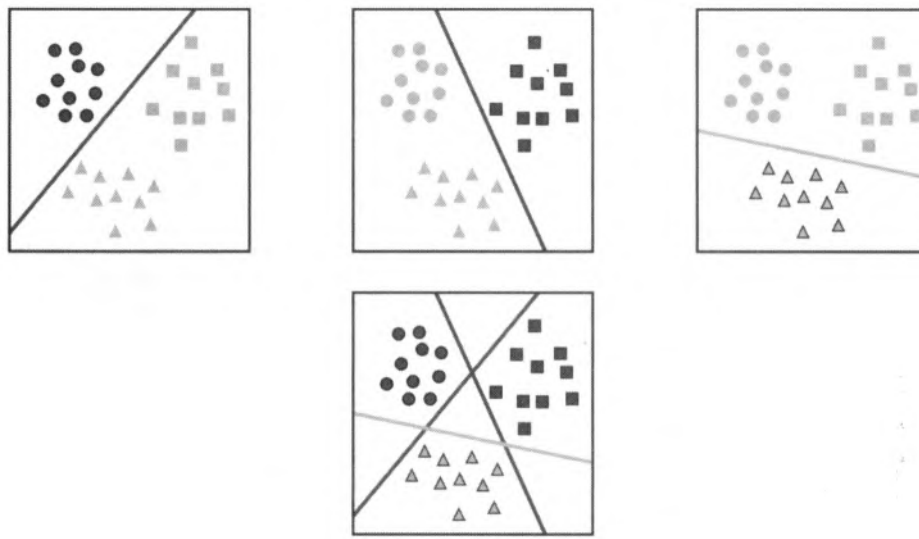


Рис. 7.2. Три линейных двухклассовых классификатора "один против всех", усвоенных на наборе данных, показанном на рис. 7.1 (*сверху*). Все три классификатора показаны поверх исходного набора данных (*внизу*)

Решив смежную задачу путем усвоения C классификаторов "один против всех", мы можем теперь спросить: существует ли возможность каким-то образом *объединить* эти двухклассовые классификаторы для решения исходной многоклассовой задачи? Как мы увидим, совершенно определенно — да.

Но сначала полезно разбить задачу на три части: во-первых, для точек, которые лежат на положительной стороне *одного* двухклассового классификатора; во-вторых, для точек, которые лежат на положительной стороне *более чем одного* классификатора; и в-третьих, для точек, которые не лежат на положительной стороне *ни одного* из классификаторов. Обратите внимание, что эти три случая исчерпывают все возможные способы расположения точки во входном пространстве по отношению к C двухклассовым классификаторам.

7.2.3. Случай 1. Назначение меток точкам на положительной стороне одного классификатора

Геометрически, точка **x**, лежащая на положительной стороне c-го классификатора, но на отрицательной стороне всех остальных, удовлетворяет следующим неравенствам: $\dot{\mathbf{x}}^T\mathbf{w}_c > 0$ и $\dot{\mathbf{x}}^T\mathbf{w}_j < 0$ для всех $j \neq c$. Сначала обратите внимание на то, что поскольку вычисления **x** всеми классификаторами являются отрицательными, за исключением c-го классификатора (результаты которых являются положительными), мы можем записать

$$\dot{\mathbf{x}}^T\mathbf{w}_c = \max_{j=0,\ldots,C-1} \dot{\mathbf{x}}^T\mathbf{w}_j \quad . \tag{7.4}$$

Далее, то, что c-й классификатор вычисляет **x** положительно, означает, что (с его точки зрения) **x** принадлежит классу c. Поскольку все остальные классификаторы оценивают **x** отрицательно, (с их точки зрения) **x** не принадлежит ни к одному классу $j \neq c$. В целом все C отдельных классификаторов находятся *в согласии* с тем, что **x** должна получить метку $y = c$. Следовательно, используя уравнение (7.4), мы можем записать метку y как

$$y = \arg\max_{j=0,\ldots,C-1} \dot{\mathbf{x}}^T\mathbf{w}_j \, . \tag{7.5}$$

На рис. 7.3 показан результат классифицирования всех точек в пространстве нашего набора данных, которые лежат на положительной стороне *одного* классификатора. Эти точки окрашены в соответствии с относящимся к ним классификатором. Обратите внимание, что на рис. 7.3 по-прежнему существуют участки, оставшиеся неокрашенными. К ним относятся участки, где точки находятся либо на положительной стороне более чем одного классификатора, либо на положительной стороне ни одного из классификаторов. Далее мы обсудим эти случаи.

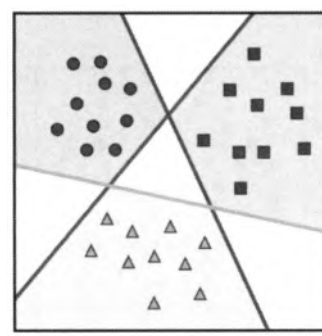


Рис. 7.3. Классификация "один против всех" точек, лежащих на положительной стороне одного классификатора

7.2.4. Случай 2. Назначение меток точкам на положительной стороне более чем одного классификатора

Когда точка **x** попадает на положительную сторону более чем одного классификатора, это означает, что, в отличие от предыдущего случая, более чем один классификатор будет претендовать на **x** как на свою собственную. В подобной ситуации и, как мы видели в *разд. 6.8.2*, мы можем рассматривать *знаковое расстояние* от точки **x** до двухклассовой границы решения как меру нашей уверенности в том, как точка должна быть помечена. Чем *дальше* точка находится на *положительной* стороне классификатора, тем *больше* у нас уверенности в том, что эта точка должна быть помечена +1. Интуитивно мы имеем простую геометрическую идею: чем больше расстояние от точки до границы, тем глубже она лежит в одном участке полупространства классификатора, и поэтому мы можем быть гораздо увереннее в ее классовой идентичности, чем точка ближе к границе. Еще один способ подумать — это представить, что произойдет, если мы внесем легкое возмущение в границу решения. Эти точки, первоначально близкие к его границе, могут оказаться на другой стороне пертурбированной гиперплоскости, поменяв классы, в то время как те точки, которые находятся дальше от границы, менее подвержены такому влиянию, и поэтому мы с самого начала можем быть увереннее в их классовой идентичности.

В настоящем контексте мы можем интуитивно расширить эту идею. Если точка лежит на положительной стороне многочисленных двухклассовых классификаторов, то нам следует назначить ей метку, соответствующую границе решения, от которой она находится *дальше* всего.

Давайте теперь посмотрим, как эта простая идея работает на нашем наборе данных. На рис. 7.4, *слева* и *в центре* показаны две примерные точки (нарисованные черным цветом) в участке нашего набора данных, которые лежат на положительной стороне более чем одного двухклассового классификатора. В каждом случае мы выделяем расстояние от новой точки до обеих границ решения пунктиром черного цвета, причем проекция точки на каждую границу решения показана как × того же цвета, что и соответствующий ей классификатор.

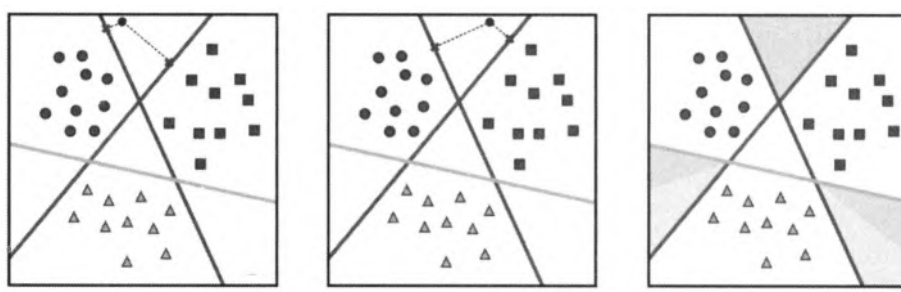


Рис. 7.4. Классификация "один против всех" точек, лежащих на положительной стороне более чем одного классификатора. Здесь точка (показанная черным цветом) лежит на положительной стороне как красного, так и синего классификаторов, причем ее расстояние до каждой границы решения отмечено пунктирными линиями, соединяющими ее с каждым классификатором

Начиная с точки, показанной на левой части рисунка, обратите внимание на то, что она лежит на положительной стороне как красного, так и синего классификаторов. Однако мы можем быть уверены в том, что точка должна принадлежать к синему классу, поскольку эта точка лежит на большем расстоянии от синей границы решения, чем красная (как можно судить, рассматривая длину пунктирных линий, исходящих от точки к каждой границе). По той же логике точка, показанная на среднем рисунке, лучше всего относится к красному классу, находясь на большем расстоянии от границы красного классификатора, чем от синего. Если мы продолжим рассуждать в том же ключе относительно каждой точки в этом участке (а также тех точек, которые находятся в двух других треугольных участках, где два или более классификатора являются положительными) и раскрасим каждую точку цветом соответствующего ей класса, то в итоге затеним каждый такой участок (рис. рис. 7.4, *справа*). Отрезки прямых, разделяющие каждую пару цветных участков на этой части рисунка (например, зеленые и розовые области правого нижнего треугольного участка), состоят из точек, лежащих на равном расстоянии (и на положительных сторонах) от двух классификаторов "один против всех". Другими словами, они являются частями нашей многоклассовой границы решения.

Разложив веса $\mathbf{w}_j$ нашего j-го классификатора путем отделения смещения от признаковых весов как

$$(\text{смещение}):\ b_j = w_{0,j} \quad (\text{признаковые веса}):\ \boldsymbol{\omega}_j = \begin{bmatrix} w_{1,j} \\ w_{2,j} \\ \vdots \\ w_{N,j} \end{bmatrix}, \tag{7.6}$$

мы получим возможность *записать расстояние* от точки $\mathbf{x}$ до ее границы решения:

$$\text{знаковое расстояние от } \mathbf{x} \text{ до } j\text{-й границы равно } \frac{\dot{\mathbf{x}}^T \mathbf{w}_j}{\|\boldsymbol{\omega}_j\|_2}. \tag{7.7}$$

Теперь, если мы *нормализуем* веса каждого линейного классификатора на длину его нормального вектора (содержащего все признаковые веса) как

$$\mathbf{w}_j \leftarrow \frac{\mathbf{w}_j}{\|\boldsymbol{\omega}_j\|_2}, \tag{7.8}$$

что мы и рассчитываем делать дальше, то это расстояние можно просто записать как сырое вычисление значения точки посредством границы решения:

$$\text{знаковое расстояние от } \mathbf{x} \text{ до } j\text{-й границы равно } \dot{\mathbf{x}}^T \mathbf{w}_j. \tag{7.9}$$

Для того чтобы отнести точку к одному из наших текущих участков, мы выискиваем классификатор, который *максимизирует* эту величину. Выражаясь алгебраически, метка y, назначенная этой точке, задается (после весовой нормализации) как

$$y = \underset{j=0,\ldots,C-1}{\arg\max}\ \dot{\mathbf{x}}^T \mathbf{w}_j. \tag{7.10}$$

Это точно такое же правило, которое мы нашли в уравнении (7.5) для участков пространства, где только один классификатор является положительным. Обратите внимание, что нормализирование весов каждого классификатора (по магнитуде их признаковых весов) не влияет на правильность этой схемы назначения меток точкам, лежащим на положительной стороне только одного классификатора, поскольку деление набора сырых вычисленных значений на положительное число не меняет математический *знак* этих сырых значений: единственный классификатор, который положительно оценивает точку, останется единственным классификатором, знаковое расстояние которого до рассматриваемой точки является положительным.

7.2.5. Случай 3. Назначение меток точкам на положительной стороне ни одного из классификаторов

Когда точка **x** не лежит на положительной стороне *ни одного* из наших C классификаторов "один против всех" (или, другими словами, лежит на *отрицательной* стороне их всех), то каждый из наших классификаторов обозначает ее как *не* относящуюся к соответствующему им классу. Таким образом, мы не можем утверждать (как это было в предыдущем случае), что один классификатор более уверен в ее классовой идентичности. Вместо этого мы можем задаться вопросом: какой классификатор *наименее* уверен в том, что **x** не принадлежит к его классу? Ответ — эта граница решения *не* та, от которой он наиболее далек, как это было раньше, а та, к которой он *ближе* всего. Здесь снова пригодится понятие *знакового расстояния до границы* решения; при этом отнесение точки **x** к границе решения, к которой она находится ближе всего, означает отнесение ее к границе, к которой она имеет наибольшее знаковое расстояние (т. к. теперь это знаковое расстояние, а также все другие знаковые расстояния являются отрицательными). Формально, опять же исходя из того, что веса каждого классификатора были нормализованы, мы можем написать

$$y = \arg\max_{j=0,\ldots,C-1} \dot{\mathbf{x}}^T \mathbf{w}_j . \tag{7.11}$$

На рис. 7.5, *слева* и *в центре* показаны два примера точек в участке нашего игрушечного набора данных, которые лежат на *отрицательной* стороне всех трех наших двухклассовых классификаторов. Это белый треугольный участок в середине данных. Снова мы выделяем расстояние от каждой точки до всех трех границ решения пунктиром черного цвета, причем проекция точки на каждую границу решения показана как × того же цвета, что и соответствующий ей классификатор.

Начнем с точки на левом рисунке, поскольку она лежит на отрицательной стороне всех трех классификаторов. Лучшее, что мы можем сделать, — это отнести ее к классу, к которому она ближе всего (или "меньше всего нарушает"), здесь синий класс. Схожим образом точка, показанная на рисунке *в центре*, может быть отнесена к зеленому классу, т. к. она находится ближе всего к его границе. Если мы повторим эту логику для каждой точки в участке и раскрасим каждую точку цветом соответствующего ей класса, то в итоге мы затеним этот центральный участок, как показано на рис. 7.5, *справа*. Отрезки прямой, разделяющие цветные участки в се-

редине этого рисунка, состоят из точек, расположенных на равном расстоянии (и на отрицательных сторонах) от всех классификаторов "один против всех". Другими словами, они являются частями нашей многоклассовой границы решения.

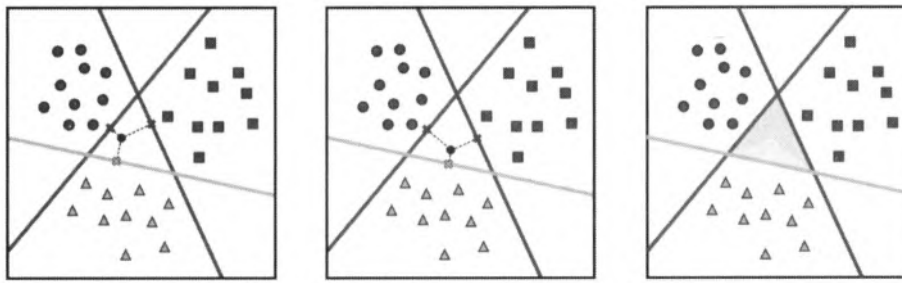


Рис. 7.5. Классификация "один против всех" точек, лежащих на отрицательной стороне всех классификаторов

7.2.6. Все случаи вместе

Теперь, разбив задачу комбинирования наших C двухклассовых классификаторов для выполнения многоклассовой классификации на три исчерпывающих случая, мы пришли к выводу о том, что для назначения метки y любой точке $\mathbf{x}$ мы можем использовать *одно-единственное* правило

$$y = \underset{j=0,\ldots,C-1}{\arg\max}\, \dot{\mathbf{x}}^T \mathbf{w}_j, \tag{7.12}$$

исходя из допущения, что (признаковые) веса каждого классификатора были нормализованы, как и в уравнении (7.8). Мы называем это правило *правилом слияния*, поскольку мы можем думать о нем как о правиле, которое объединяет наши C двухклассовых классификатора вместе, чтобы определить метку, которая должна быть назначена точке на основе понятия знакового расстояния до границы решения.

На рис. 7.6 показан результат применения правила слияния из уравнения (7.12) ко всему входному пространству нашего набора данных, который мы использовали на протяжении всего этого раздела. Оно включает участки пространства, где точки лежат: во-первых, на положительной стороне одного классификатора (см. рис. 7.3); во-вторых, на положительной стороне более чем одного классификатора (см. рис. 7.4, *справа*), и в-третьих, на положительной стороне ни одного из классификаторов (см. рис. 7.5, *справа*).

На левой панели рис. 7.6 мы показываем все пространство, окрашенное надлежащим образом в соответствии с правилом слияния, вместе с тремя исходными двухклассовыми классификаторами. На правой панели мы выделяем (черным цветом) отрезки прямой, которые определяют границы между участками пространства, занимаемыми каждым классом. Это действительно является нашим *многоклассовым классификатором* или *границей решения*, предоставляемой правилом слияния. В общем случае при использовании линейных двухклассовых классификаторов

граница, вытекающая из правила слияния, всегда будет *кусочно-линейной* (как в нашем простом примере здесь). Хотя правило слияния явно определяет эту кусочно-линейную границу, оно не дает нам для нее хорошей формулы замкнутой формы (как в случае с двухклассовыми вычислительными каркасами, такими как логистическая регрессия или SVM-машины).

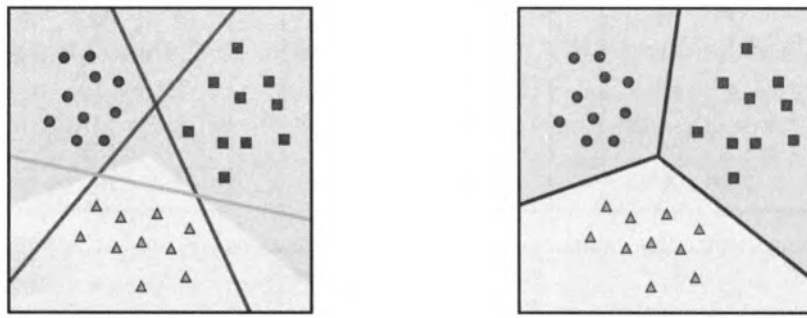


Рис. 7.6. Результат применения правила слияния из уравнения (7.12) к входному пространству нашего игрушечного набора данных с участками, окрашенными в соответствии с их предсказанной меткой наряду с исходными двухклассовыми границами решения (*слева*) и объединенной многоклассовой границей решения черным цветом (*справа*)

7.2.7. Алгоритм "один против всех"

Двухшаговый процесс, который мы сейчас видели (т. е. усваиваем C двухклассовых классификаторов "один против всех", а затем объединяем их с помощью правила слияния), обычно именуется *алгоритмом многоклассовой классификации "один против всех"*, или OvA-алгоритмом (One-versus-All). На практике часто встречаются имплементации алгоритма OvA, которые пропускают шаг нормализации, показанный в уравнении (7.8). Это теоретически может привести к низкой точности классификации из-за разных размеров нормальных векторов, создавая внешкальные показания расстояния до классификатора. Однако, поскольку часто каждый классификатор тренируется с использованием одной и той же схемы локальной оптимизации, результирующая магнитуда каждого натренированного нормального вектора может оказаться в итоге примерно одинаковой, что уменьшает разницу между нормализованным и ненормированным вычислением входных точек, выполняемым каждым классификатором.

Пример 7.1. Классификация набора данных с $C = 4$ классами с помощью алгоритма OvA

В этом примере мы применим алгоритм OvA для классифицирования игрушечного набора данных с $C = 4$ классами, показанными на рис. 7.7, *слева*. Здесь синие, красные, зеленые и желтые точки имеют значения меток соответственно 0, 1, 2 и 3. На рис. 7.7, *в центре* показано входное пространство, окрашенное в соответствии с правилом слияния, вместе с индивидуально усвоенными $C = 4$ двухклассовыми классификаторами. Обратите внимание, что ни один из этих двухклассовых клас-

сификаторов не отделяет полностью соответствующий ему класс от остальных данных. Тем не менее окончательная многоклассовая граница решения, показанная на рис. 7.7, *справа*, прекрасно различает эти четыре класса.

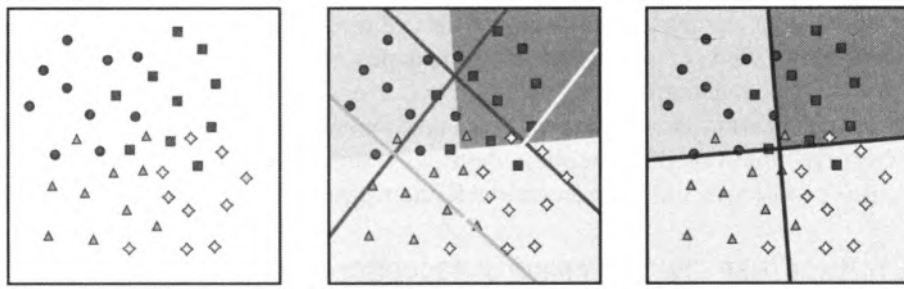


Рис. 7.7. Иллюстрация примера 7.1

Пример 7.2. Регрессионный взгляд на правило слияния

При выведении правила слияния в уравнении (7.12) мы рассматривали задачу многоклассовой классификации с точки зрения перцептрона (впервые приведенную в двухклассовом случае в *разд. 6.4*), имея в виду, что мы смотрели на наши данные только в пространстве входных признаков, окрашивая значение каждой метки вместо того, чтобы трактовать метки как выходную размерность и строить график данных в трех размерностях. Однако, если мы сделаем это, глядя на наши многоклассовые данные "со стороны" в том, что мы можем считать регрессионной точкой зрения, мы на самом деле можем рассматривать правило слияния как *многоуровневую шаговую функцию*. Это показано на рис. 7.8 для первичного набора данных, используемого в этом разделе. На левом рисунке показаны данные и правило слияния "сверху", тогда как на правом тот же самый расклад показан "сбоку", причем правило слияния изображено как дискретная шаговая функция.

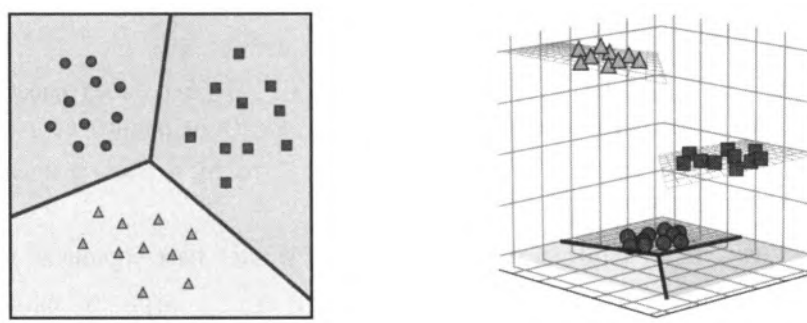


Рис. 7.8. Правило слияния для игрушечного классификационного набора данных с $C = 3$ классами, показанное с точки зрения перцептрона (*слева*) и с точки зрения регрессии (*справа*). Обратите внимание, что зазубренные края на некоторых шагах в правой части рисунка являются всего лишь артефактом механизма графопостроения, используемого для генерирования трехмерного графика. На самом деле края каждого шага являются гладкими, как объединенная граница решения, показанная во входном пространстве

7.3. Многоклассовая классификация и перцептрон

В этом разделе мы обсудим естественную альтернативу многоклассовой классификации "один против всех" (OvA), подробно описанной в предыдущем разделе. Вместо того чтобы сначала тренировать C двухклассовых классификаторов, а затем объединять их в *единую* границу решения (посредством правила слияния), мы тренируем все C классификаторов *одновременно*, чтобы удовлетворить правило слияния непосредственно. В частности, для достижения этого результата, который можно считать прямым обобщением *двухклассового перцептрона* (*см. разд. 6.4*), мы выводим стоимость многоклассового перцептрона.

7.3.1. Стоимостная функция многоклассового перцептрона

В предыдущем разделе о многоклассовой классификации OvA мы видели, как правило слияния из уравнения (7.12) определяет принадлежность к классу каждой точки $\mathbf{x}$ во входном пространстве задачи. Сюда, конечно же, входят все (входные) точки $\mathbf{x}_p$ из нашего тренировочного набора данных $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$. В идеале, при правильной настройке всех двухклассовых классификаторов мы хотели бы, чтобы правило слияния соблюдалось для как можно большего числа этих точек:

$$y = \arg\max_{j=0,\ldots,C-1} \dot{\mathbf{x}}_p^T \mathbf{w}_j . \tag{7.13}$$

Вместо того чтобы настраивать наши C двухклассовых классификатора один за другим и затем комбинировать их, мы можем усвоить веса всех C классификаторов *одновременно*, чтобы удовлетворять этому идеальному условию как можно чаще.

Для того чтобы начать строить надлежащую стоимостную функцию, минимизаторы которой удовлетворяют этому идеалу, сначала обратите внимание на то, что если уравнение (7.13) должно соблюдаться для нашей p-й точки, то мы можем сказать, что следующее ниже равенство также должно быть истинным:

$$\dot{\mathbf{x}}_p^T \mathbf{w}_{y_p} = \max_{j=0,\ldots,C-1} \dot{\mathbf{x}}_p^T \mathbf{w}_j . \tag{7.14}$$

Другими словами, в соответствии с уравнением (7.14) (знаковое) расстояние от точки $\mathbf{x}_p$ до ее классовой границы решения *больше* (или равно) ее расстоянию до любой другой двухклассовой границы решения. Это то, что мы в идеале хотим получить для всех наших тренировочных точек данных.

Вычитание $\dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}$ из правой части уравнения (7.14) дает нам хорошего кандидата для поточечной стоимостной функции, которая всегда неотрицательна и минимальна в нуле:

$$g_p(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}) = \left(\max_{j=0,\ldots,C-1} \dot{\mathbf{x}}_p^T \mathbf{w}_j \right) - \dot{\mathbf{x}}_p^T \mathbf{w}_{y_p} . \tag{7.15}$$

Обратите внимание, что если наши веса $\mathbf{w}_0, \ldots, \mathbf{w}_{C-1}$ установлены идеально, то $g_p(\mathbf{w}_0, \ldots, \mathbf{w}_{C-1})$ должна быть равна нулю для как можно большего числа точек.

Имея это в виду, мы можем сформировать стоимостную функцию, взяв среднее арифметическое значение поточечной стоимости в уравнении (7.15) над всем набором данных:

$$g(\mathbf{w}_0, ..., \mathbf{w}_{C-1}) = \frac{1}{P}\sum_{p=1}^{P}\left[\left(\max_{j=0,...,C-1} \dot{\mathbf{x}}_p^T \mathbf{w}_j\right) - \dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}\right]. \quad (7.16)$$

Эта стоимостная функция, которую мы далее будем называть *стоимостью многоклассового перцептрона*, обеспечивает способ одновременной настройки всех классификаторных весов для *восстановления* весов, которые удовлетворяют правилу слияния как нельзя лучше.

7.3.2. Минимизирование стоимости многоклассового перцептрона

Подобно своему двухклассовому аналогу, рассмотренному в *разд. 6.4*, многоклассовый перцептрон всегда является выпуклым независимо от задействованного набора данных (см. упражнения главы). Он также имеет тривиальное решение в нуле. То есть при $\mathbf{w}_j = 0$ для всех $j = 0, ..., C-1$ стоимость является минимальной. Этого нежелательного поведения часто можно избежать, инициализируя любую схему локальной оптимизации, используемую для ее минимизирования вдали от начала координат. Обратите внимание также на то, что мы ограничены использованием методов оптимизации нулевого и первого порядков для минимизирования стоимости многоклассового перцептрона, поскольку его вторая производная равна нулю (где бы она ни была определена).

Пример 7.3. Минимизирование многоклассового перцептрона

В этом примере мы минимизируем многоклассовый перцептрон из уравнения (7.16) над многоклассовым набором данных, первоначально показанным на рис. 7.1, минимизируя стоимость посредством процедуры стандартного градиентного спуска (*см. разд. 3.5*).

На рис. 7.9, *слева* построен график набора данных окончательной многоклассовой классификации над всем пространством и каждой отдельной границы решения "один против всех" (т. е. где $\dot{\mathbf{x}}^T \mathbf{w}_c = 0$ для $c = 0, 1, 2$). На рис. 7.9, *справа* показана объединенная многоклассовая граница решения, образованная путем слияния этих индивидуальных границ "один против всех" с помощью правила слияния. Обратите внимание, что на левой части рисунка ввиду того, что мы *не* тренировали каждый отдельный двухклассовый классификатор по принципу "один против всех" (как это было сделано в предыдущем разделе), каждый отдельный усвоенный двухклассовый классификатор показывает довольно слабую результативность в вычленении своего класса из остальных данных. Это совершенно нормально, поскольку именно слияние этих линейных классификаторов (посредством правила слияния) обеспечивает окончательную многоклассовую границу решения, изображенную на правой части рисунка, которая обеспечивает совершенную классификацию.

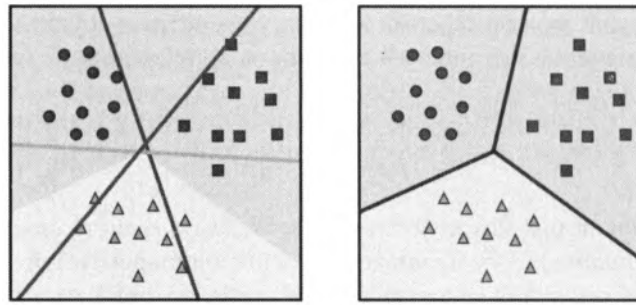


Рис. 7.9. Иллюстрация примера 7.3

7.3.3. Альтернативные формулировки стоимости многоклассового перцептрона

Стоимость многоклассового перцептрона из уравнения (7.16) также может быть выведена как *прямое обобщение* его двухклассовой версии, представленной в *разд. 6.4*. Используя следующее простое свойство функции max

$$\max\left(s_0, s_1, ..., s_{C-1}\right) - z = \max\left(s_0 - z, s_1 - z, ..., s_{C-1} - z\right), \tag{7.17}$$

где $s_0, s_1, ..., s_{C-1}$ и z — скалярные значения, мы можем записать каждое слагаемое в правой части уравнения (7.16) как

$$\max_{j=0,...,C-1} \dot{\mathbf{x}}_p^T \left(\mathbf{w}_j - \mathbf{w}_{y_p}\right). \tag{7.18}$$

Обратите внимание, что для $j = y_p$ мы имеем $\dot{\mathbf{x}}_p^T \left(\mathbf{w}_j - \mathbf{w}_{y_p}\right) = 0$. Это позволяет нам переписать величину в уравнении (7.18) эквивалентным образом как

$$\max_{\substack{j=0,...,C-1 \\ j \neq y_p}} \left(0,\ \dot{\mathbf{x}}_p^T \left(\mathbf{w}_j - \mathbf{w}_{y_p}\right)\right) \tag{7.19}$$

и, следовательно, всю стоимость многоклассового перцептрона как

$$g\left(\mathbf{w}_0, ..., \mathbf{w}_{C-1}\right) = \frac{1}{P} \sum_{p=1}^{P} \max_{\substack{j=0,...,C-1 \\ j \neq y_p}} \left(0,\ \dot{\mathbf{x}}_p^T \left(\mathbf{w}_j - \mathbf{w}_{y_p}\right)\right). \tag{7.20}$$

В этой форме легко увидеть, что при $C = 2$ стоимость многоклассового перцептрона сводится к двухклассовому варианту, подробно описанному в *разд. 6.4*.

7.3.4. Регуляризирование многоклассового перцептрона

При выведении правила слияния и последующей стоимостной функции многоклассового перцептрона мы приняли допущение о том, что нормальный вектор для каждого двухклассового классификатора имеет единичную длину, благодаря чему мы можем справедливо сравнивать (знаковое) расстояние каждого входа $\mathbf{x}_p$ с каж-

дой из наших двухклассовых границ решения "один против всех" (как впервые подробно описано в *разд. 7.2.4*). Это означает, что при минимизировании стоимости многоклассового перцептрона из уравнения (7.16) мы должны (по крайней мере формально) подчинить его ограничениям, что все эти нормальные векторы имеют единичную длину, в итоге получив задачу *ограниченной* минимизации

$$\begin{aligned} &\underset{b_0,\boldsymbol{\omega}_0,\ldots,b_{C-1},\boldsymbol{\omega}_{C-1}}{\text{минимизировать}} \quad \frac{1}{P}\sum_{p=1}^{P}\left[\left(\max_{j=0,\ldots,C-1} b_j + \mathbf{x}_p^T\boldsymbol{\omega}_c\right) - \left(b_{y_p} + \mathbf{x}_p^T\boldsymbol{\omega}_{y_p}\right)\right] \\ &\text{с учетом, что} \qquad \|\boldsymbol{\omega}_j\|_2^2 = 1, \quad j = 0,\ldots,C-1, \end{aligned} \tag{7.21}$$

где мы использовали нотацию смещения/признаковых весов, позволяющую нам разложить каждый весовой вектор $\mathbf{w}_j$, как показано в уравнении (7.6).

Хотя эта задача может быть решена в ее ограниченной форме, более широкое распространение (в сообществе машинного обучения) получило *ослабление* такой задачи (как мы видели ранее с двухклассовым перцептроном в *разд. 6.4.6* и машиной опорных векторов в *разд. 6.5*) и решение *регуляризованной* версии.

Хотя теоретически мы могли бы обеспечить отдельный штраф (или регуляризационный) параметр для каждого из C ограничений в уравнении (7.21), для простоты можно выбрать один регуляризационный параметр $\lambda \geq 0$, чтобы штрафовать магнитуду всех нормальных векторов одновременно. Благодаря этому нам нужно предоставить только одно регуляризационное значение вместо C отдельных регуляризационных параметров, в итоге получив регуляризированную версию задачи многоклассового перцептрона:

$$\underset{b_0,\boldsymbol{\omega}_0,\ldots,b_{C-1},\boldsymbol{\omega}_{C-1}}{\text{минимизировать}} \quad \frac{1}{P}\sum_{p=1}^{P}\left[\left(\max_{j=0,\ldots,C-1} b_j + \mathbf{x}_p^T\boldsymbol{\omega}_j\right) - \left(b_{y_p} + \mathbf{x}_p^T\boldsymbol{\omega}_{y_p}\right)\right] + \lambda\sum_{j=0}^{C-1}\|\boldsymbol{\omega}_j\|_2^2. \tag{7.22}$$

Эта регуляризированная форма не совсем совпадает с исходной ограниченной формулировкой, т. к. регуляризация всех нормальных векторов вместе не обязательно гарантирует, что $\|\boldsymbol{\omega}_j\|_2^2 = 1$ для всех j. Однако эта форма, как правило, заставляет магнитуду всех нормальных векторов "вести себя хорошо", например запрещая одному нормальному вектору (его магнитуде) бесконечно расти, тогда как другой сжимается почти до нуля. Подобные ситуации часто встречаются в машинном обучении, и нам приходится соглашаться на компромиссы, чтобы получить что-то, находящееся "достаточно близко" к оригиналу, коль скоро это хорошо работает на практике. Это действительно так, поскольку λ обычно устанавливается равным малому значению (например, 10^{-3} или меньше).

7.3.5. Стоимостная функция многоклассового мягкого максимума

Как и в случае с двухклассовым перцептроном (*см. разд. 6.4.3*), мы часто готовы пожертвовать малым количеством прецизионности моделирования, сформировав близко совпадающую более гладкую стоимостную функцию относительно той, которая у нас уже есть, чтобы облегчить оптимизацию или расширить оптимизационные инструменты, которые мы могли бы пустить в ход. Как и в случае с двухклас-

совым перцептроном, здесь мы также имеем возможность сгладить стоимость многоклассового перцептрона, задействуя функцию *мягкого максимума*.

Замена функции max в каждом слагаемом многоклассового перцептрона в уравнении (7.16) на его softmax-аппроксимацию из уравнения (6.34) дает следующую стоимостную функцию:

$$g\left(\mathbf{w}_0, ..., \mathbf{w}_{C-1}\right) = \frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{j=0}^{C-1} e^{\dot{\mathbf{x}}_p^T \mathbf{w}_j}\right) - \dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}\right]. \tag{7.23}$$

Она называется *стоимостной функцией многоклассового мягкого максимума*, потому что построена путем сглаживания многоклассового перцептрона с использованием функции мягкого максимума, а также потому что она может быть показана как прямое многоклассовое обобщение функции двухклассового мягкого максимума (см., например, уравнение (6.37)). Стоимостная функция многоклассового мягкого максимума в уравнении (7.23) также имеет много других названий: *стоимость многоклассовой перекрестной энтропии*, *стоимость softplus* и *многоклассовая логистическая стоимость*.

7.3.6. Минимизирование функции многоклассового мягкого максимума

Стоимостная функция многоклассового мягкого максимума не только является выпуклой (см. примеры в главе), но и (в отличие от многоклассового перцептрона) имеет бесконечно много гладких производных, что позволяет нам использовать методы второго порядка (в дополнение к методам нулевого и превого порядков), чтобы надлежаще ее минимизировать. Обратите также внимание на то, что она больше не имеет тривиального решения в нуле, аналогичного его двухклассовому мягко-максимумному аналогу, которое устраняет этот недостаток из двухклассового перцептрона.

Пример 7.4. Метод Ньютона, применяемый для минимизирования стоимости многоклассового мягкого максимума

На рис. 7.10 представлен результат применения метода Ньютона для минимизирования стоимости многоклассового мягкого максимума к игрушечному набору данных с $C = 4$ классами, впервые показанному на рис. 7.7.

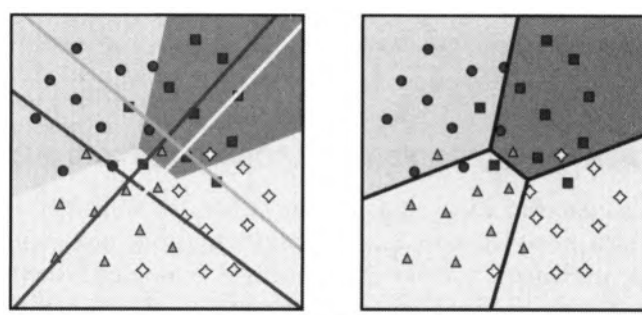


Рис. 7.10. Иллюстрация примера 7.4

7.3.7. Альтернативные формулировки многоклассового мягкого максимума

Сглаживание формулировки многоклассового перцептрона, приведенное в уравнении (7.16), путем замены функции max функцией мягкого максимума (softmax) дает эквивалентную, но иную форму многоклассового мягкого максимума в виде

$$g(\mathbf{w}_0, ..., \mathbf{w}_{C-1}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(\sum_{\substack{j=0 \\ j \neq y_p}}^{C-1} e^{\mathbf{x}_p^T(\mathbf{w}_j - \mathbf{w}_{y_p})}\right). \tag{7.24}$$

Визуально эта форма выглядит более похожей на стоимость двухклассового мягкого максимума и действительно сводится к ней при $C = 2$ и $y_p \in \{-1, +1\}$.

Эта стоимостная функция также называется *стоимостью многоклассовой перекрестной энтропии*, поскольку она также является естественным обобщением двухклассовой версии, рассмотренной в *разд. 6.2*. Для того чтобы убедиться, что это действительно так, сначала обратите внимание на то, что мы можем переписать p-е слагаемое стоимости многоклассового мягкого максимума в уравнении (7.24), используя тот факт, что $\log(e^s) = s$:

$$\log\left(\sum_{j=0}^{C-1} e^{\mathbf{x}_p^T\mathbf{w}_j}\right) - \mathbf{x}_p^T\mathbf{w}_{y_p} = \log\left(\sum_{c=0}^{C-1} e^{\mathbf{x}_p^T\mathbf{w}_c}\right) - \log\left(e^{\mathbf{x}_p^T\mathbf{w}_{y_p}}\right). \tag{7.25}$$

Далее мы можем использовать свойство логарифма $\log(s) - \log(t) = \log(s/t)$, чтобы переписать уравнение (7.25) как

$$\log\left(\sum_{j=0}^{C-1} e^{\mathbf{x}_p^T\mathbf{w}_j}\right) - \log e^{\mathbf{x}_p^T\mathbf{w}_{y_p}} = \log\left(\frac{\sum_{j=0}^{C-1} e^{\mathbf{x}_p^T\mathbf{w}_j}}{e^{\mathbf{x}_p^T\mathbf{w}_{y_p}}}\right). \tag{7.26}$$

Наконец, поскольку $\log(s) = -\log(1/s)$, получаем

$$\log\left(\frac{\sum_{j=0}^{C-1} e^{\mathbf{x}_p^T\mathbf{w}_j}}{e^{\mathbf{x}_p^T\mathbf{w}_{y_p}}}\right) = -\log\left(\frac{e^{\mathbf{x}_p^T\mathbf{w}_{y_p}}}{\sum_{j=0}^{C-1} e^{\mathbf{x}_p^T\mathbf{w}_j}}\right). \tag{7.27}$$

В целом мы можем выразить многоклассовый мягкий максимум из уравнения (7.23) эквивалентным образом как

$$g(\mathbf{w}_0, ..., \mathbf{w}_{C-1}) = -\frac{1}{P}\sum_{p=1}^{P} \frac{e^{\mathbf{x}_p^T\mathbf{w}_{y_p}}}{\sum_{j=0}^{C-1} e^{\mathbf{x}_p^T\mathbf{w}_j}}. \tag{7.28}$$

Визуально эта формула более похожа на стоимость двухклассовой перекрестной энтропии из уравнения (6.12) и действительно сводится к ней довольно простым образом при $C = 2$ и $y_p \in \{0, 1\}$.

7.3.8. Регуляризация и многоклассовый мягкий максимум

Как и в случае со стоимостью многоклассового перцептрона (*см. разд. 7.3.4*), общепринято *регуляризировать* многоклассовый мягкий максимум как

$$\frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{j=0}^{C-1}e^{b_j+\mathbf{x}_p^T\boldsymbol{\omega}_j}\right)-\left(b_{y_p}+\mathbf{x}_p^T\boldsymbol{\omega}_{y_p}\right)\right]+\lambda\sum_{j=0}^{C-1}\left\|\boldsymbol{\omega}_j\right\|_2^2, \tag{7.29}$$

где мы использовали нотацию смещения/признаковых весов, позволяющую нам разложить каждый весовой вектор $\mathbf{w}_c$, как показано в уравнении (7.6). Регуляризация также помогает предотвратить расхождение методов локальной оптимизации, таких как метод Ньютона (который предпринимает большие шаги), во время работы с полностью разделяемыми данными (*см. разд. 6.4*).

7.3.9. Имплементация на Python

Для имплементации любой из многоклассовых стоимостных функций, подробно описанных в этом разделе, сначала полезно переписать нашу модель, используя матричную нотацию, впервые представленную в *разд. 5.6*. Другими словами, мы сначала укладываем веса из наших C классификаторов в один $(N+1)\times C$-массив:

$$\mathbf{W}=\begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & \cdots & w_{0,C-1} \\ w_{1,0} & w_{1,1} & w_{1,2} & \cdots & w_{1,C-1} \\ w_{2,0} & w_{1,2} & w_{2,2} & \cdots & w_{2,C-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{N,0} & w_{N,1} & w_{N,2} & \cdots & w_{N,C-1} \end{bmatrix}. \tag{7.30}$$

Здесь смещение и нормальный вектор c-го классификатора были уложены один поверх другого и составляют c-й столбец массива. Расширим также нашу model-нотацию, чтобы обозначить все вычисления наших C отдельных линейных моделей вместе через

$$\text{model}(\mathbf{x},\ \mathbf{W})=\dot{\mathbf{x}}^T\mathbf{W}=\begin{bmatrix}\dot{\mathbf{x}}^T\mathbf{w}_0 & \dot{\mathbf{x}}^T\mathbf{w}_1 & \cdots & \dot{\mathbf{x}}^T\mathbf{w}_{C-1}\end{bmatrix}. \tag{7.31}$$

Это точно *такая же* сжатая линейная модель, которую мы использовали для имплементирования многовыходной регрессии в *разд. 5.6.3*, которую мы повторим ниже.

```
# вычислить C линейных комбинаций входных точек, по одной на классификатор
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

С помощью этой model-нотации нам удобнее имплементировать по существу любую формулу, выведенную из правила слияния, как, например, многоклассовый перцептрон. Мы можем записать само правило слияния из уравнения (7.12) эквивалентным образом:

$$y=\max\left[\text{model}(\mathbf{x},\ \mathbf{W})\right]. \tag{7.32}$$

Точно так же мы можем компактно записать p-е слагаемое многоклассового перцептрона как

$$\left(\max_{c=0,\ldots,C-1} \dot{\mathbf{x}}_p^T \mathbf{w}_c\right) - \dot{\mathbf{x}}_p^T \mathbf{w}_{y_p} = \max\left[\text{model}\left(\mathbf{x}_p, \mathbf{W}\right)\right] - \text{model}\left(\mathbf{x}_p, \mathbf{W}\right)_{y_p}, \qquad (7.33)$$

где член $\text{model}\left(\mathbf{x}_p, \mathbf{W}\right)_{y_p}$ относится к y_p-й записи члена $\text{model}\left(\mathbf{x}_p, \mathbf{W}\right)$.

Python-код часто работает намного быстрее, когда циклы `for` — или эквивалентая операции включения в список — пишутся эквивалентно с использованием матрично-векторных операций библиотеки NumPy (эта тема постоянно присутствовала в наших имплементациях, начиная с линейной регрессии в *разд. 5.2.4*).

Ниже приведен пример имплементации многоклассового перцептрона из уравнения (7.16), которая принимает представленную выше функцию `model`. Обратите внимание, что операция `np.linalg.fro` обозначает матричную *норму Фробениуса* (*см. разд. П3.5.3*).

Совершенно аналогичным образом можно имплементировать многоклассовый мягкий максимум из уравнения (7.23).

```
# многоклассовый перцептрон
lam = 10**-5  # наш регуляризационный параметр
def multiclass_perceptron(w):
    # предвычислить предсказания на всех точках
    all_evals = model(x,w)

    # максимум по всем точкам данных
    a = np.max(all_evals,axis = 0)

    # стоимость в компактной форме с помощью операции транслирования numpy
    b = all_evals[y.astype(int).flatten(),np.arange(np.size(y))]
    cost = np.sum(a - b)

    # добавить регуляризатор
    cost = cost + lam*np.linalg.norm(w[1:,:],'fro')**2

    # вернуть среднее
    return cost/float(np.size(y))
```

Наконец, обратите внимание, что поскольку любая стоимостная функция для многоклассовой классификации, описанная в этом разделе, принимает *матрицу* параметров, при использовании библиотеки autograd в рамках вашего процесса оптимизации бывает очень удобно сначала *сгладить* выбранную вами стоимость — как описано в *разд. П2.10.3* — перед минимизацией. Это позволяет избежать необходимости явно перебирать веса в вашей процедуре локальной оптимизации, позволяя вам напрямую задействовать базовые Python-имплементации, например градиентный спуск (*см. разд. 3.5.4*) и метод Ньютона (*см. разд. 4.3.6*) без изменений.

7.4. Какой подход дает наилучшие результаты?

В предыдущих двух разделах мы видели два фундаментальных подхода к линейной многоклассовой классификации: один против всех (OvA) и многоклассовый перцептрон/многоклассовый мягкий максимум. Оба подхода широко используются на практике и часто (в зависимости от набора данных) дают сходные результаты (см., например, [25, 26]). Однако последний подход (по крайней мере в принципе) способен обеспечить более высокую точность на более широком диапазоне наборов данных. Это связано с тем, что в случае OvA мы решаем последовательность из C двухклассовых подзадач (по одной на класс), настраивая веса наших классификаторов *независимо* друг от друга. Только после этого мы объединяем все классификаторы вместе, чтобы сформировать правило слияния. Таким образом, веса, которые мы усваиваем, удовлетворяют правилу слияния *косвенно*. С другой стороны, в случае минимизации на основе стоимостной функции многоклассового перцептрона или мягкого максимума мы настраиваем все параметры всех C классификаторов *одновременно*, чтобы удовлетворить правилу слияния над нашим тренировочным набором данных непосредственно. Эта совместная минимизация позволяет проявляться потенциально ценным взаимодействиям между двухклассовыми классификаторами при настройке их весов, которые не могут происходить в подходе на основе алгоритма OvA.

Пример 7.5. Сравнение OvA и многоклассового перцептрона

Проиллюстрируем принципиальное превосходство подхода на основе многоклассового перцептрона над OvA, используя игрушечный набор данных с $C = 5$ классами, показанный на рис. 7.11, *слева*, где точки, окрашенные в красный (квадратики), синий (кружочки), зеленый (треугольники), желтый (ромбики) и фиолетовый (звездочки) цвета, соответственно имеют значения меток $y_p = 0$, 1, 2, 3 и 4.

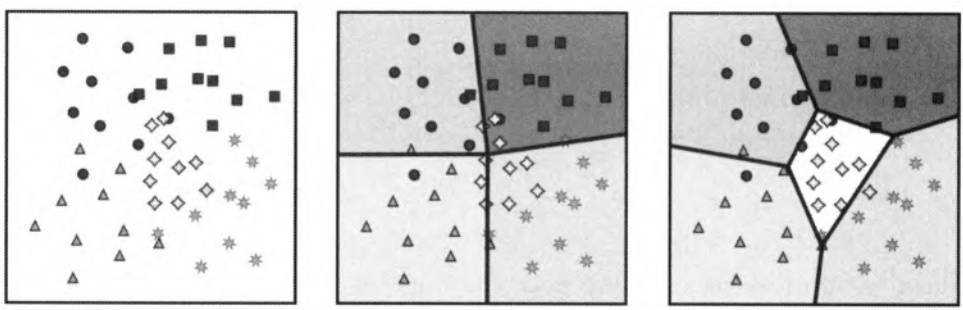


Рис. 7.11. Иллюстрация примера 7.5

Вообразите на мгновение, как подход на основе OvA будет работать с точки зрения класса, окрашенного в желтый цвет, сосредоточенного в середине набора данных, в особенности как будет решена подзадача, в которой мы различаем членов этого класса и всех остальных. Поскольку этот класс данных окружен членами других классов, а членов желтого класса меньше, чем всех остальных классов, вместе взя-

тых, оптимальное линейное классификационное правило для этой подзадачи состоит в том, чтобы классифицировать все точки как не желтые (или, другими словами, неправильно классифицировать весь желтый класс целиком). Из этого следует, что линейная граница решения будет лежать вне диапазона показанных точек, причем все точки в тренировочных данных будут лежать на ее отрицательной стороне. Поскольку веса границы решения, связанной с классом желтого цвета, настраиваются исключительно на основе этой подзадачи, это приведет к тому, что весь желтый класс будет неправильно классифицирован в окончательном решении OvA, предоставленном правилом слияния, как показано на рис. 7.11, *в центре*.

С другой стороны, если задействовать подход на основе стоимостной функции многоклассового перцептрона или мягкого максимума, то мы не пропустим этот класс, т. к. все $C = 5$ двухклассовых классификаторов усваиваются *одновременно*, что приводит к окончательной объединенной границе решения, которая намного превосходит ту, что обеспечивает алгоритм OvA, как показано на рис. 7.11, *справа*. Мы неправильно классифицируем гораздо меньше точек и, в частности, не классифицируем неправильно весь желтый класс данных.

7.5. Стоимостная функция категориальной перекрестной энтропии

В предыдущих разделах мы по умолчанию задействовали числовые значения меток $y_p \in \{0, 1, ..., C-1\}$. Однако, как и в случае двухклассовой классификации (*см. разд. 6.7*), выбор значений меток при многоклассовой классификации также носит произвольный характер. Независимо от того, как мы определяем значения меток, мы все равно получаем точно такую же многоклассовую схему, которую мы видели в *разд. 7.3*, и стоимостные функции, такие как многоклассовый мягкий максимум.

В этом разделе мы увидим, как использовать *категориальные* метки, т. е. метки, не имеющие внутреннего числового порядка, для выполнения многоклассовой классификации. Эта точка зрения вводит понятие *дискретного распределения вероятностей*, а также понятие *стоимости категориальной перекрестной энтропии*, которая (как мы увидим) полностью эквивалентна стоимостной функции многоклассового мягкого максимума/многоклассовой перекрестной энтропии, которую мы видели в *разд. 7.3*.

7.5.1. Дискретные распределения вероятностей

Предположим, что вы опросили десять друзей или коллег по работе, задавая вопрос о том, есть ли у них дома кошка или собака. Из этой группы вы выяснили, что три человека не владели ни кошкой, ни собакой, один человек владел кошкой, а у шести человек были собаки. Построив соответствующий вектор $\mathbf{s} = [3 \quad 1 \quad 6]$ данных из этого ответа на опрос, $\mathbf{s}$ можно визуально представить в виде *гистограммы*, где значение каждой записи представлено вертикальным столбиком, высота которого

пропорциональна соответствующему ему значению. Гистограмма этого конкретного вектора показана на рис. 7.12, *слева*.

Довольно часто принято *нормализовать* векторы данных, подобные этому, чтобы их можно было интерпретировать как *дискретное распределение вероятностей*. Для этого нормализация должна быть выполнена таким образом, чтобы, во-первых, числовой порядок его значений (от наименьшего к наибольшему) был сохранен, во-вторых, его значения были неотрицательными, и в-третьих, его значения в сумме составляли ровно 1. Для вектора всех неотрицательных записей это можно сделать, просто разделив его на сумму его значений. Поскольку сумма (неотрицательных) значений примера, который мы используем здесь, равна $3+1+6=10$, это потребует деления всех записей **s** на 10:

$$\mathbf{s} = \begin{bmatrix} 0{,}3 & 0{,}1 & 0{,}6 \end{bmatrix}. \tag{7.34}$$

Нормализованная гистограмма (иногда именуемая функцией массы вероятности) показана на рис. 7.12, *справа*.



Рис. 7.12. Визуальное представление вектора выборочных данных $\mathbf{s} = [3\ 1\ 6]$, полученного в результате опроса владельцев домашних животных и представленного в виде гистограммы (*слева*). Нормализация этого вектора путем деления его на сумму его входов позволяет рассматривать его как дискретное распределение вероятностей (*справа*)

7.5.2. Экспоненциальная нормализация

Такого рода нормализация может быть выполнена в общем случае для вектора $\mathbf{s} = [s_0\ s_1\ \ldots\ s_{C-1}]$ любой длины C с потенциально отрицательными элементами. Экспоненциирование[1] каждого элемента в **s** дает вектор

$$\begin{bmatrix} e^{s_0} & e^{s_1} & \ldots & e^{s_{C-1}} \end{bmatrix}, \tag{7.35}$$

где все записи теперь гарантированно являются неотрицательными. Обратите также внимание на то, что экспоненциация поддерживает порядок значений в **s** от малого до большого[2].

---

[1] То есть потенцирование, или возведение в степень с некоторым основанием (действие, обратное логарифмированию по этому основанию). — *Прим. пер.*

[2] Это обусловлено тем, что экспоненциальная функция $e^{(\cdot)}$ всегда монотонно возрастает.

Если теперь мы разделим сумму этой экспоненциированной версии **s** из каждой ее записи

$$\sigma(\mathbf{s}) = \left[\frac{e^{s_0}}{\sum_{c=0}^{C-1} e^{s_c}} \quad \frac{e^{s_1}}{\sum_{c=0}^{C-1} e^{s_c}} \quad \cdots \quad \frac{e^{s_{C-1}}}{\sum_{c=0}^{C-1} e^{s_c}} \right], \tag{7.36}$$

то не только сохраним два вышеупомянутых свойства (т. е. неотрицательность и структуру числового порядка), но удовлетворим третье свойство допустимого дискретного распределения вероятностей: все записи теперь в сумме составляют 1.

Функция $\sigma(\cdot)$, определенная в уравнении (7.36), называется *нормализованной экспоненциальной функцией*[3]. Она часто используется для того, чтобы мы могли *интерпретировать* произвольный вектор **s** (возможно, содержащий как отрицательные, так и положительные записи) как дискретное распределение вероятностей (рис. 7.13), и ее можно рассматривать как обобщение *логистической сигмоиды*, введенной в *разд. 6.2.3*.

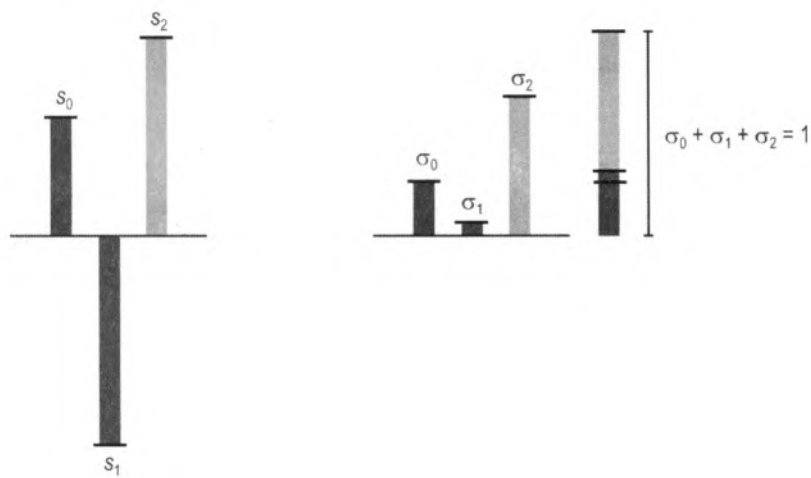


Рис. 7.13. Вектор длины $C = 3$, показанный в виде гистограммы (*слева*). Взяв нормализованный экспоненциал этого вектора, как определено в уравнении (7.36), мы получаем новый вектор всех неотрицательных записей, числовой порядок которых сохраняется и суммарное значение которого равно 1 (*справа*)

[3] В контексте нейронных сетей эта функция иногда называется *активацией softmax*, или *мягкомаксимумной активацией*. Это соглашение об именовании является неудачным, поскольку нормализованный экспоненциал не является мягкой версией функции max, как правильно именованная функция softmax, подробно описанная в *разд. 7.2.5*, с которой ее не следует путать. Хотя она представляет собой преобразование, которое действительно сохраняет индекс самой большой записи в ее входных данных, она также *не* является мягкой версией функции argmax, как это иногда ошибочно утверждается.

7.5.3. Экспоненциально нормализованные знаковые расстояния

В предыдущих двух разделах мы полагались на *знаковое расстояние* точки $\mathbf{x}$ до каждой из C отдельных границ решения (или чего-то очень близкого к ней, если мы не нормализуем признаковые веса), чтобы надлежаще определить принадлежность к классу. Это было кодифицировано непосредственно в правиле слияния (приведенном, например, в уравнении (7.12)).

Для заданной конфигурации наших весов для всех C двухклассовых классификаторов значение точки $\mathbf{x}$, вычисляемое через все границы решения, порождает C величин знакового расстояния

$$\mathbf{s} = \begin{bmatrix} \dot{\mathbf{x}}^T\mathbf{w}_0 & \dot{\mathbf{x}}^T\mathbf{w}_1 & \ldots & \dot{\mathbf{x}}^T\mathbf{w}_{C-1} \end{bmatrix}, \tag{7.37}$$

которые мы можем рассматривать как *гистограмму*.

Поскольку — как мы видели выше — нормализованная экспоненциальная функция предопределяет числовой порядок, мы можем также рассмотреть экспоненциально нормализованное знаковое расстояние при определении надлежащей принадлежности к классу. Обозначив через $\sigma(\cdot)$ нормализованный экспоненциал, наша обобщенная гистограмма знаковых расстояний становится такой:

$$\sigma(\mathbf{s}) = \left[\frac{e^{\dot{\mathbf{x}}^T\mathbf{w}_0}}{\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}^T\mathbf{w}_c}} \quad \frac{e^{\dot{\mathbf{x}}^T\mathbf{w}_1}}{\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}^T\mathbf{w}_c}} \quad \cdots \quad \frac{e^{\dot{\mathbf{x}}^T\mathbf{w}_{C-1}}}{\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}^T\mathbf{w}_c}} \right]. \tag{7.38}$$

Преобразовав гистограмму величин знаковых расстояний, мы также получаем возможность рассматривать принадлежность к классу с *вероятностной* точки зрения. Например, если для некоторой конфигурации всего множества весов для некоторой точки $\mathbf{x}_p$ дано

$$\sigma(\mathbf{s}) = \begin{bmatrix} 0{,}1 & 0{,}7 & 0{,}2 \end{bmatrix}, \tag{7.39}$$

то, хотя мы по-прежнему будем *назначать метку, основываясь на правиле слияния* (см. уравнение (7.12)) — здесь назначая метке $y_p = 1$, т. к. вторая запись 0,7 этого вектора является наибольшей, — мы также можем выразить уверенность в том, что "$y_p = 1$ с вероятностью 70%".

Пример 7.6. Знаковые расстояния как распределение вероятностей

На рис. 7.14 представлен прототипный набор данных с $C = 3$ классами (показанный на рис. 7.1) и визуализированы оба вектора знаковых расстояний в виде гистограмм для нескольких точек (*слева*), а также их нормализованные экспоненциальные версии (*справа*). Обратите внимание, что самая большая положительная запись в каждой исходной гистограмме, показанной на левом рисунке, остается самой большой положительной записью в нормализованной версии, показанной на правом рисунке.

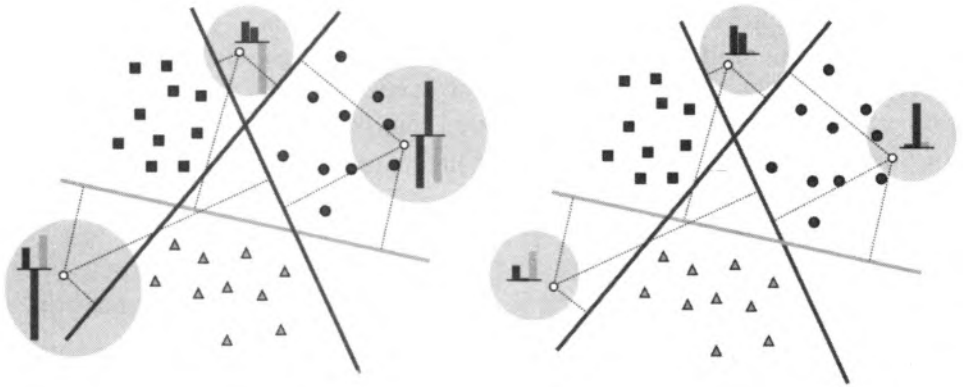


Рис. 7.14. Визуализация гистограммы величин знаковых расстояний трех образцовых точек в наборе данных с $C = 3$ классами (*слева*). Экспоненциально нормализованные показания знаковых расстояний визуализируются в виде гистограмм для тех же самых трех классов (*справа*)

7.5.4. Категориальная классификация и стоимость категориальной перекрестной энтропии

Предположим, что мы начинаем с многоклассового классификационного набора данных $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ с N-мерным входом и преобразовываем наши числовые значения меток $y_p \in \{0, 1, ..., C-1\}$ с помощью *векторов, кодированных с одним активным состоянием*, в форме

$$\begin{aligned} y_p = 0 \quad &\longleftarrow \quad \mathbf{y}_p = [1 \;\; 0 \;\; \cdots \;\; 0 \;\; 0]; \\ y_p = 1 \quad &\longleftarrow \quad \mathbf{y}_p = [0 \;\; 1 \;\; \cdots \;\; 0 \;\; 0]; \\ &\vdots \\ y_p = C-1 \quad &\longleftarrow \quad \mathbf{y}_p = [0 \;\; 0 \;\; \cdots \;\; 0 \;\; 1]. \end{aligned} \tag{7.40}$$

Здесь каждая кодированная таким образом *категориальная метка* является вектором длины C и содержит одни нули, кроме 1 в индексе, равном значению y_p (обратите внимание, что первая запись в этом векторе имеет индекс 0, а последняя — $C-1$).

Каждое векторное представление однозначно идентифицирует соответствующее ему значение метки, но теперь наши значения меток больше не являются упорядоченными числовыми значениями, а наш набор данных принимает форму $\{(\mathbf{x}_p, \mathbf{y}_p)\}_{p=1}^{P}$, где $\mathbf{y}_p$ — это определенные выше метки, кодированные с одним активным состоянием. Наша цель, однако, остается прежней: надлежаще настроить веса наших C двухклассовых классификаторов "один против всех", чтобы усвоить наилучшее соответствие между N-мерным входом и C-мерным выходом нашего тренировочного набора данных.

С векторнозначным выходом вместо скалярных числовых значений мы можем сформулировать многоклассовую классификацию как пример *многовыходной регрессии* (*см. разд. 5.6*). Другими словами, обозначив через $\mathbf{W}$ весовую матрицу размера $(N+1)\times C$ для всех C классификаторов (*см. разд. 7.3.9*), мы можем нацелиться на настройку $\mathbf{W}$ таким образом, чтобы приближенная линейная связь соблюдалась для всех наших точек:

$$\dot{\mathbf{x}}_p^T\mathbf{W} \approx \mathbf{y}_p. \tag{7.41}$$

Однако, поскольку теперь наш выход $\mathbf{y}_p$ состоит не из непрерывных значений, а из векторов, кодированных с одним активным состоянием, линейная связь вообще не очень хорошо представляла бы такие векторы. Левосторонние записи для данного p могут быть неотрицательными, меньше нуля или больше единицы и т. д. Однако взятие *нормализованной экспоненциальной* трансформанты нашей линейной модели нормализует его таким образом (заставляя все его записи быть неотрицательными и иметь в сумме ровно единицу), что мы можем разумно предложить настроить $\mathbf{W}$ так, чтобы

$$\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{W}\right) \approx \mathbf{y}_p \tag{7.42}$$

соблюдалось как можно плотнее над нашим тренировочным набором данных (где $\sigma(\cdot)$ — это нормализованный экспоненциал). Интерпретируя $\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{W}\right)$ как дискретное распределение вероятностей, мы можем сказать, что хотим настроить веса нашей модели так, чтобы это распределение полностью концентрировалось на индексе y_p, т. е. единственной ненулевой записи выхода $\mathbf{y}_p$, кодированного с одним активным состоянием.

Для того чтобы усвоить наши веса надлежащим образом, мы могли бы задействовать стандартную поточечную регрессионную функцию потерь, такую как наименьшие квадраты (*см. разд. 5.2*):

$$g_p(\mathbf{W}) = \left\|\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{W}\right) - \mathbf{y}_p\right\|_2^2. \tag{7.43}$$

Однако, как мы обсуждали в *разд. 6.2*, более подходящей поточечной функцией при работе с двоичным выходом является логарифмическая ошибка, поскольку она сильнее штрафует ошибки меньше единицы в таких случаях. Здесь логарифмическая ошибка нормализованного экспоненциала $\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{W}\right)$ и $\mathbf{y}_p$ может быть записана как

$$g_p(\mathbf{W}) = -\sum_{c=0}^{C} y_{p,c} \log\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{W}\right)_c, \tag{7.44}$$

где $y_{p,c}$ — c-я запись метки $\mathbf{y}_p$, кодированной с одним активным состоянием. Обратите внимание, что эта формула значительно упрощает задачу, т. к. $\mathbf{y}_p$ является вектором с одним активным состоянием, следовательно, все слагаемые, кроме одного, в правой части приведенной выше формуле равны нулю. Это как раз пер-

воначальная целочисленная метка c точки $\mathbf{x}_p$, отсюда c-й индекс выхода $\mathbf{y}_p$ равен единице, т. е. $y_{p,c} = 1$. Это означает, что вышеизложенное упрощается до

$$g_p(\mathbf{W}) = -\log \sigma\left(\dot{\mathbf{x}}_p^T \mathbf{W}\right)_{y_p} \tag{7.45}$$

и из определения нормализованного экспоненциала это в точности равно

$$g_p(\mathbf{W}) = -\log\left(\frac{e^{\dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}}}{\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}_p^T \mathbf{w}_c}}\right). \tag{7.46}$$

Если затем мы сформируем стоимостную функцию, взяв среднее арифметическое из приведенных выше значений над всеми P тренировочными точками данных, то получим

$$g(\mathbf{W}) = -\frac{1}{P}\sum_{p=1}^{P} \log\left(\frac{e^{\dot{\mathbf{x}}_p^T \mathbf{w}_{y_p}}}{\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}_p^T \mathbf{w}_c}}\right). \tag{7.47}$$

Это именно та форма стандартной стоимостной функции многоклассовой перекрестной энтропии/мягкого максимума, которую мы видели в *разд. 7.3.7* в уравнении (7.28), где мы использовали числовые значения меток $y_p \in \{0, 1, ..., C-1\}$.

7.6. Метрики качества классификации

В этом разделе мы опишем простые метрики для суждения о качестве натренированной многоклассовой классификационной модели, а также то, как делать предсказания с помощью одной из них.

7.6.1. Выполнение предсказаний с использованием натренированной модели

Если мы обозначим оптимальное множество весов для c-го двухклассового классификатора "один против всех" как $\mathbf{w}_c^\star$ — найденное путем минимизирования многоклассовой классификационной стоимостной функции из *разд. 7.3* либо путем выполнения алгоритма OvA, как подробно описано в *разд. 7.2*, то для предсказания метки y' входа $\mathbf{x}'$ мы задействуем *правило слияния* в виде

$$y' = \underset{c=0,\ldots,C-1}{\arg\max}\, \mathbf{x}'^T \mathbf{w}_c^\star, \tag{7.48}$$

где любой точке, лежащей точно на границе решения, должна быть назначена метка случайным образом, основываясь на индексе тех классификаторов, которые обеспечивают максимальное вычисляемое значение. Действительно, этот набор точек образует многоклассовую границу решения — проиллюстрированную черным цветом для игрушечного набора данных с $C = 3$ классами на рис. 7.15, *слева*, — где участки на этом изображении были окрашены на основе предсказанной

метки, получаемой в результате вычисления значения каждой точки во входном пространстве с помощью правила слияния. Правая панель этого изображения показывает тот же самый набор данных и индивидуальные границы "один против всех" (где каждая из них окрашена в зависимости от класса, который она отличает от остальных данных) и для точки $\mathbf{x}'$ изображает ее значение, вычисленное правилом слияния, как максимальное расстояние до каждой границы "один против всех" (т. е. примерно то, что вычисляется благодаря оцениванию правила слияния — *см. разд. 7.2.4*).

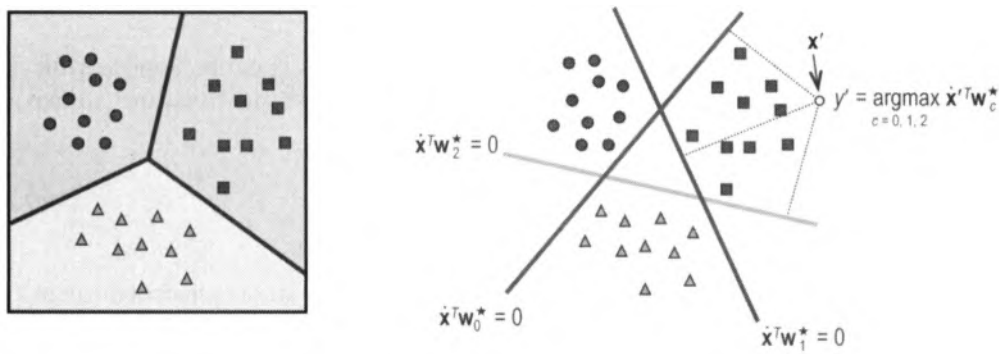


Рис. 7.15. Многоклассовая граница решения на основе набора данных с $C = 3$ классами (показана черным цветом вместе с участками, окрашенными в соответствии с предсказанием, предоставляемым правилом слияния; *слева*). Тот же самый набор данных с каждой границей решения "один против всех" (каждая из них окрашена в зависимости от класса, который она отличает от остальных данных; *справа*), которые — при их объединении посредством правила слияния — обеспечивают многоклассовую границу решения, показанную на рисунке слева. Здесь метка y' новой точки $\mathbf{x}'$, показанная в виде полого круга в правом верхнем углу, предсказывается посредством правила слияния, которое измеряет максимальное расстояние от этой точки до каждой границы решения "один против всех"

7.6.2. Оценивание уверенности

После того как будет усвоена надлежащая граница решения, мы сможем описать *уверенность*, которую мы имеем в любой точке, основываясь *на расстоянии точки до границы решения*, точно так же, как мы можем это сделать с двухклассовыми данными (*см. разд. 6.8.2*). Выражаясь конкретнее, мы можем использовать его экспоненциально нормализованное расстояние для оценивания нашей уверенности в предсказании, как описано в *разд. 7.5.3*.

7.6.3. Суждение о качестве натренированной модели с помощью метрики точности

Для того чтобы подсчитать число неправильных классифицирований, которые натренированный многоклассовый классификатор формирует над нашим трениро-

вочным набором данных, мы просто берем сырое количество тренировочных точек данных $\mathbf{x}_p$, истинная метка y_p которых предсказана *неправильно*. Для того чтобы сравнить предсказанную метку $\hat{y}_p = \underset{j=0,\ldots,C-1}{\arg\max}\, \mathbf{x}_p^T \mathbf{w}_j^\star$ точки $\mathbf{x}_p$ и ее истинную метку y_p, мы можем применить индикаторную функцию $\mathcal{I}(\cdot)$ и вычислить

$$\mathcal{I}\left(\hat{y}_p, y_p\right) = \begin{cases} 0, & \text{если } \hat{y}_p = y_p; \\ 1, & \text{если } \hat{y}_p \neq y_p. \end{cases} \tag{7.49}$$

Взятие суммы вышеуказанного над всеми P точками дает общее число неправильных классифицирований, предсказанных нашей натренированной моделью

$$\text{число неправильных классифицирований} = \sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p, y_p\right). \tag{7.50}$$

Используя его, мы также можем вычислить точность $\mathcal{A}$ натренированной модели. Это просто нормализованное число тренировочных точек данных, метки которых правильно предсказаны моделью:

$$\mathcal{A} = 1 - \frac{1}{P}\sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p, y_p\right). \tag{7.51}$$

Точность варьируется от 0 (ни одна точка не классифицирована правильно) до 1 (все точки классифицированы правильно).

Пример 7.7. Сравнение стоимостной функции и подсчет стоимостных значений

На рис. 7.16 мы сравниваем число неправильных классифицирований со значением функции многоклассового мягкого максимума при классифицировании набора данных с $C = 5$ классами, показанного на рис. 7.11, на протяжении трех прогонов стандартного градиентного спуска с использованием параметра длины шага $\alpha = 10^{-2}$ для всех трех прогонов.

Сравнивая левую и правую части рисунка, мы видим, что число неправильных классифицирований и значений мягкого максимума на каждом шаге градиентного спуска повторяют друг друга не полностью. То есть это не тот случай, когда при уменьшении значения стоимостной функции то же самое происходит и с числом неправильных классифицирований (очень похоже на двухклассовый случай). Это происходит потому, что наша стоимость мягкого максимума является лишь аппроксимацией истинной величины, которую мы хотели бы минимизировать, т. е. число неправильных классифицирований.

Приведенный выше простой пример имеет чрезвычайно важное практическое значение: в то время как мы настраиваем веса нашей модели, минимизируя соответствующую многоклассовую стоимость, после минимизации наилучший набор весов из прогона ассоциируется с наименьшим числом *неправильных классифицирований* (или схожим образом с самой высокой точностью), а *не* с наименьшим значением стоимостной функции.

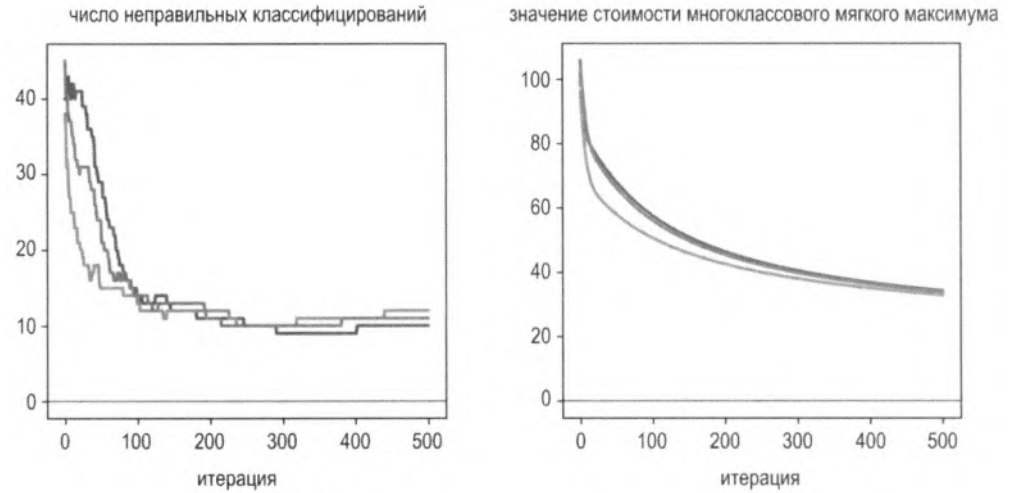


Рис. 7.16. Иллюстрация примера 7.7

7.6.4. Продвинутые метрики качества для работы с несбалансированными классами

Продвинутые метрики качества, которые мы видели в случае двухклассовой классификации для решения серьезных проблем классового дисбаланса, в том числе сбалансированная точность (*см. разд. 6.8.4*), а также дополнительные метрики, определяемые матрицей путаницы (*см. разд. 6.8.5*), могут быть непосредственно расширены для решения проблем классового дисбаланса в многоклассовом контексте. Эти прямые расширения разведываются далее в упражнениях этой главы.

Пример 7.8. Матрица путаницы для игрушечного набора данных

На рис. 7.17, *слева* показан результат полностью настроенного многоклассового классификатора, натренированного на наборе данных, впервые приведенном на рис. 7.7. *Матрица путаницы*, соответствующая этому классификатору, представлена в правой части рисунка. (i, j)-й элемент этой матрицы подсчитывает число тренировочных точек данных, имеющих *истинную* метку $y = i$ и *предсказанную* метку $\hat{y} = j$.

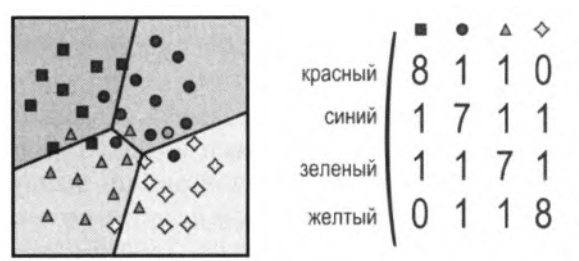


Рис. 7.17. Иллюстрация примера 7.9

Пример 7.9. Набор данных Iris

В этом примере мы разведываем применение линейной многоклассовой классификации к популярному набору данных Iris, взятому из работы [27]. Этот набор данных состоит из набора $P = 150$ статистических измерений на $C = 3$ видах цветов рода многолетних корневищных растений семейства *ирисовые* (виды *ирис щетинистый* (Iris Setosa), *ирис разноцветный* (Iris Versicolor) и *ирис вергинский* (Iris Virginica)), причем в данных представлено по 50 представителей каждого вида цветов. Каждая входная точка состоит из $N = 4$ признаков, включая длину и ширину чашелистика, а также длину и ширину лепестка (все в сантиметрах).

Минимизировав стоимость многоклассового мягкого максимума, мы можем легко достичь почти совершенной классификации этого набора данных, в результате получив два неправильных классифицирования и следующую ниже матрицу путаницы.

		Предсказано		
		щетинистый	разноцветный	вергинский
Фактически	щетинистый	50	0	0
	разноцветный	0	49	1
	вергинский	0	1	49

7.7. Взвешенная многоклассовая классификация

Взвешенная многоклассовая классификация возникает точно по тем же причинам, которые описаны для двухклассовой классификации в *разд. 6.9*; т. е. как способ включения понятия уверенности в точки данных и для решения серьезных классовых дисбалансов. Можно легко вывести взвешенные версии стоимостных функций многоклассового перцептрона/мягкого максимума, которые полностью отражают двухклассовые аналоги, подробно описанные в ранее в этой главе (см. упражнение 7.10).

7.8. Стохастическое и мини-пакетное обучение

В *разд. П1.5* описано простое расширение алгоритма стандартного градиентного спуска (*см. главу 3*), именуемое *мини-пакетным градиентным спуском*. Этот подход призван ускорить минимизацию стоимостных функций, состоящих из суммы (или среднего значения) P членов:

$$g(\mathbf{w}) = \sum_{p=1}^{P} g_p(\mathbf{w}), \tag{7.52}$$

где g_1, g_2, ..., g_P — функции одного и того же рода, которые принимают одинаковые параметры.

Как мы уже видели, каждая стоимостная функция обучения с учителем выглядит примерно так — включая те, которые используются для регрессии, двухклассовой и многоклассовой классификации. Каждый g_p — это то, что мы обычно называем (по)точечной стоимостью, которая измеряет ошибку конкретной модели в p-й точке набора данных. Например, в случае со стоимостью наименьших квадратов мы видели в *разд. 5.2*, что поточечная стоимость принимала форму $g_p(\mathbf{w}) = \left(\dot{\mathbf{x}}_p^T\mathbf{w} - y_p\right)^2$, с двухклассовым мягким максимумом (*см. разд. 6.4.3*) она принимала форму $g_p(\mathbf{w}) = -\log\left(\sigma\left(y_p\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right)$, а с мультиклассовым softmax (в *разд. 7.3.5*) — форму $g_p(\mathbf{W}) = \left[\log\left(\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}_p^T\mathbf{w}_c}\right) - \dot{\mathbf{x}}_p^T\mathbf{w}_{y_p}\right]$. В более общем случае, как мы и увидим, двигаясь вперед (например, в случае линейного автоэнкодера, описанного в *разд. 8.3*), *каждая стоимостная функция машинного обучения* принимает такую форму (поскольку они всегда разлагаются над своими тренировочными данными), где g_p — это поточечная стоимость p-й точки набора данных. Благодаря этому мы можем применять мини-пакетную оптимизацию непосредственно во время точной настройки их параметров.

7.8.1. Мини-пакетная оптимизация и онлайновое (потоковое) обучение

Как подробно описано в *разд. П1.5*, суть мини-пакетной идеи состоит в том, чтобы минимизировать такую стоимость *последовательно* малыми мини-пакетами ее слагаемых, по одному мини-пакету слагаемых за раз, в отличие от шага стандартной локальной оптимизации, когда минимизируется весь набор слагаемых сразу. Теперь, поскольку слагаемые стоимости машинного обучения по своей сути привязаны к тренировочным точкам данных, в контексте машинного обучения мы можем рассуждать о мини-пакетной оптимизации *эквивалентно* в терминах *мини-пакетов тренировочных данных*. Отсюда, в отличие от стандартной (так называемой *полнопакетной*) локальной оптимизации, которая выполняет отдельные шаги путем одновременного обхода всего *тренировочного* набора данных, мини-пакетный подход заставляет нас выполнять меньшие шаги, выполняющие обход тренировочных данных *последовательно* (при этом один полный обход данных называется *эпохой*). Эта интерпретация мини-пакетной оптимизации машинным обучением схематически проиллюстрирована на рис. 7.18.

Как и в случае с обобщенными функциями стоимости, мини-пакетное обучение с учителем часто значительно ускоряет минимизацию стоимостных функций машинного обучения (и следовательно, происходит соответствующее усвоение параметров, т. е. обучение) и наиболее часто сочетается с градиентным спуском (*см. разд. 3.5*) или одним из его усовершенствованных аналогов (*см. разд. П1.4*).

Это особенно верно при работе с *очень крупными наборами данных*, т. е. когда P велико (см., например, [21, 28]). С очень крупными наборами данных мини-пакетный подход также помогает ограничить объем активной памяти, потребляе-

мой при хранении данных, загружая — на каждом шаге в мини-пакетной эпохе — *только те данные, которые включены в текущий мини-пакет*. Мини-пакетный подход может также применяться (или интерпретироваться) в качестве так называемого метода *обучения без учителя* (потокового), при котором данные фактически поступают малыми мини-пакетами и непосредственно используются для обновления параметров ассоциированной модели.

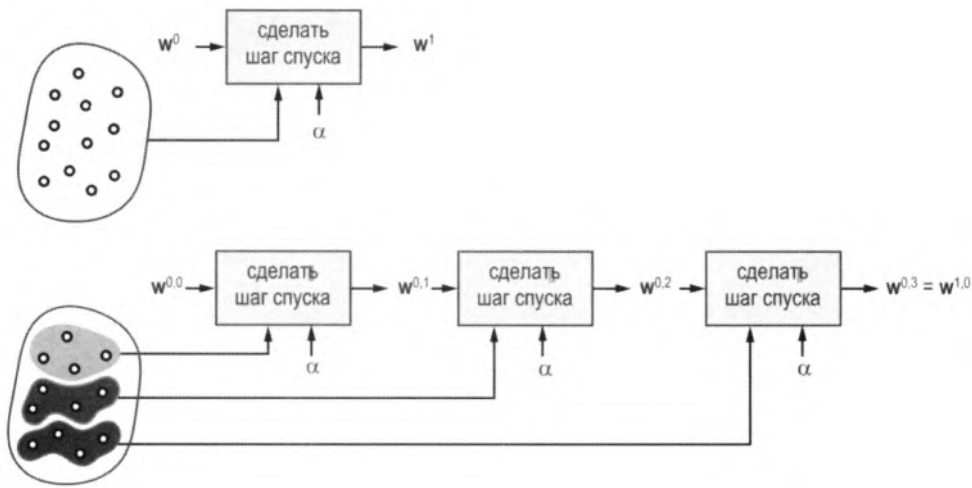


Рис. 7.18. Схематическое сравнение первой итерации (*сверху*) полного пакетного и (*снизу*) стохастического градиентного спуска через призму машинного обучения с использованием (ради простоты) малого набора точек. В полнопакетном обходе мы делаем шаг во всех точках одновременно, тогда как в мини-пакетном подходе мы проходим эти точки последовательно, как если бы мы получали данные онлайновым (потоковым) способом (*снизу*)

.

Пример 7.10. Распознавание рукописных цифр

Выполним распознавание рукописных цифр (как показано в примере 1.10) посредством многоклассовой классификации. На рис. 7.19 проиллюстрировано ускоренное схождение мини-пакетного градиентного спуска по сравнению с методом стандартного градиентного спуска с использованием стоимости многоклассового мягкого максимума и $P = 50\,000$ случайно отобранных тренировочных точек из набора данных MNIST [29] — популярной коллекции рукописных изображений, подобных тем, что показаны на рис. 1.13. В частности, приведено сравнение первых 10 шагов/эпох обоих методов с использованием пакета размером 200 для мини-пакетной версии и одинаковой длины шага для обоих прогонов, где мы видим, что мини-пакетный прогон резко ускоряет минимизацию как с точки зрения стоимостной функции (см. рис. 7.19, *слева*), так и с точки зрения числа неправильных классифицирований (см. рис. 7.19, *справа*).

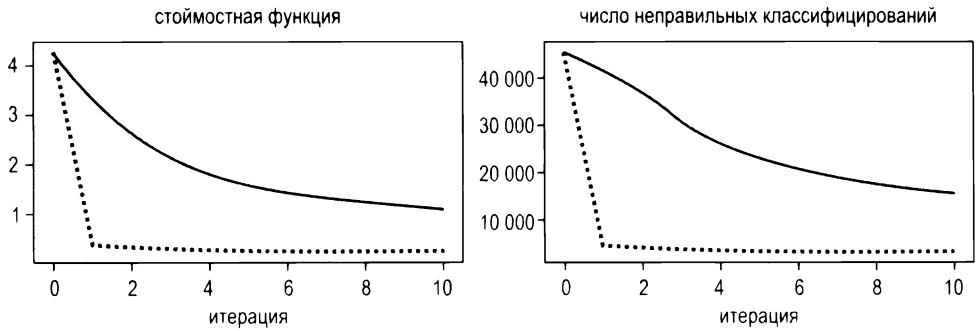


Рис. 7.19. Иллюстрация примера 7.10. Сравнение прогрессии, достигнутой стандартным (сплошная черная линия) и мини-пакетным (пунктирная сиреневая линия) градиентным спуском над тренировочным набором данных MNIST, с точки зрения значения стоимостной функции (*слева*) и числа неправильных классифицирований (*справа*). Мини-пакетный подход обеспечивает значительно более быструю прогрессию, чем стандартный градиентный спуск в обоих метриках

7.9. Заключение

В этой главе мы перешли от двухклассовой линейной классификации к многоклассовой, подробно описав ряд точек зрения на отражение нашего обсуждения в предыдущей главе.

Мы начали с подхода *"один против всех"* (OvA), описанного в *разд. 7.2*, который включает в себя разумное объединение результатов нескольких двухклассовых классификаторов "один против всех". В *разд. 7.3* мы увидели, как можно натренировать этот набор двухклассовых классификаторов *одновременно*, минимизируя стоимостные функции *многоклассового перцептрона* или *перекрестной энтропии/мягкого максимума*. Этот одновременный подход — по крайней мере в принципе — может обеспечить более высокую результативность по сравнению с OvA, но на практике оба подхода часто приводят к аналогичным результатам (*см. разд. 7.4*). Далее в *разд. 7.5* мы обсудили произвольную природу выбора значений меток, подробно описав наиболее произвольный выбор: категориальные метки. Затем в *разд. 7.6* были подробно рассмотрены многоклассовые метрики — естественные расширения указанных метрик были впервые описаны для двухклассовых задач в *разд. 6.8*. Наконец, в *разд. 7.8* мы обсудили мини-пакетную оптимизацию в контексте машинного обучения (указанная тема будет подхвачена далее в *разд. П1.5*).

7.10. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:

github.com/jermwatt/machine_learning_refined.

7.1. Псевдокод классификации "один против всех"

Написать псевдокод, который подытоживает алгоритм "один против всех" (OvA), описанный в *разд. 7.2*.

7.2. Классификация "один против всех"

Повторить эксперимент, описанный в примере 7.1. От вас не требуется воспроизводить иллюстрации на рис. 7.7, однако вы должны убедиться, что ваша натренированная модель достигает похожего результата (с точки зрения числа неправильных классифицирований), как показано здесь (имея менее десяти неправильных классифицирований).

7.3. Многоклассовый перцептрон

Повторить эксперимент, описанный в примере 7.3. От вас не требуется воспроизводить иллюстрации на рис. 7.9, однако вы должны убедиться, что ваша натренированная модель достигает нулевого числа неправильных классифицирований. Вы можете использовать Python-имплементацию стоимости многоклассового перцептрона (*см. разд. 7.3.9*).

7.4. Многоклассовый и двухклассовый перцептроны

Завершить аргумент, начатый в *разд. 7.3.3*, чтобы показать, что стоимость многоклассового перцептрона в уравнении (7.16) сводится к стоимости двухклассового перцептрона в уравнении (6.33).

7.5. Многоклассовый мягкий максимум

Повторить эксперимент, описанный в примере 7.4, используя любой метод локальной оптимизации. От вас не требуется воспроизводить иллюстрации на рис. 7.10, однако вы должны убедиться, что ваша натренированная модель достигает малого числа неправильных классифицирований (десять или меньше). Вы можете использовать Python-имплементацию в качестве основы для имплементации стоимости многоклассового мягкого максимума (*см. разд. 7.3.9*).

7.6. Показать, что мультиклассовый softmax сводится к двухклассовому softmax при $C = 2$

Завершить аргумент, начатый в *разд. 7.3.7*, чтобы показать, что стоимостная функция многоклассового softmax в уравнении (7.23) сводится к стоимостной функции двухклассового softmax в уравнении (6.34).

7.7. Ручные расчеты со стоимостью многоклассового мягкого максимума

Показать, что гессиан стоимостной функции многоклассового мягкого максимума можно вычислить поблочно для $s \neq c$ как

$$\nabla^2_{\mathbf{w}_c \mathbf{w}_s} g = \sum_{p=1}^{P} \frac{e^{\mathbf{x}_p^T \mathbf{w}_c + \mathbf{x}_p^T \mathbf{w}_s}}{\left(\sum_{d=1}^{C} e^{\mathbf{x}_p^T \mathbf{w}_d} \right)^2} \mathbf{x}_p \mathbf{x}_p^T \tag{7.53}$$

и для $s = c$ как

$$\nabla^2_{\mathbf{w}_c \mathbf{w}_c} g = \sum_{p=1}^{P} \frac{e^{\mathbf{x}_p^T \mathbf{w}_c}}{\sum_{d=1}^{C} e^{\mathbf{x}_p^T \mathbf{w}_d}} \left(1 - \frac{e^{\mathbf{x}_p^T \mathbf{w}_c}}{\sum_{d=1}^{C} e^{\mathbf{x}_p^T \mathbf{w}_d}} \right) \mathbf{x}_p \mathbf{x}_p^T . \tag{7.54}$$

7.8. Стоимости многоклассового перцептрона и мягкого максимума являются выпуклыми

Показать, что стоимостные функции многоклассового перцептрона и мягкого максимума всегда являются выпуклыми (независимо от набора данных). Для этого можно использовать, например, определение выпуклости нулевого порядка (см. упражнение 5.8).

7.9. Сбалансированная точность в многоклассовой окружающей обстановке

Распространить понятие сбалансированной точности, подробно описанное в контексте двухклассовой классификации (*см. разд. 6.8.4*), на многоклассовую окружающую обстановку. В частности, привести уравнение для сбалансированной точности в контексте многоклассовой классификации, аналогичное уравнению (6.79) для двухклассового случая.

7.10. Взвешенный многоклассовый мягкий максимум

Общая взвешенная версия двухклассового softmax приведена в уравнении (6.83). Как будет выглядеть аналогичная взвешенная стоимостная функция многоклассового мягкого максимума? Если мы устанавливаем веса стоимостной функции для решения классового дисбаланса (как подробно описано для двухклассового случая в *разд. 6.9.3*), то как мы должны устанавливать весовые значения?

7.11. Распознавание рукописных цифр

Повторить эксперимент из примера 7.10, имплементировав мини-пакетный градиентный спуск. Вы, возможно, не получите точно такие же результаты, как показано на рис. 7.19, основываясь на вашей имплементации, инициализации алгоритма и т. д. Однако вы должны воссоздать общий результат.

8 Линейное обучение без учителя

8.1. Введение

В этой главе мы обсудим несколько полезных приемов для *обучения без учителя*, широко представленных в *разд. 1.3*, которые предназначены для редукции размерности конкретного набора данных путем разумного редуцирования числа входных признаков или точек данных. Эти приемы можно задействовать в качестве шага предобработки в рамках обучения с учителем, позволяя им масштабироваться до более крупных наборов данных или для анализа данных с периодическим участием человека. Мы начинаем эту главу с рассмотрения понятия *охватывающего множества* (spanning set) из векторной алгебры, а затем подробно опишем линейный *автокодировщик*, наш фундаментальный инструмент редукции размерности признаков, и обратимся к тесно связанной теме *анализа главных компонент*. Затем мы обсудим *алгоритм кластеризации методом K средних*, рекомендательные системы и закончим обсуждением общих *задач матричной факторизации*.

8.2. Фиксированные охватывающие множества, ортонормальность и проекции

В этом разделе мы рассмотрим рудиментарные понятия линейной алгебры, которые имеют решающее значение для понимания методов обучения без учителя. Заинтересованный читатель, нуждающийся в обновлении своих знаний об основных векторных и матричных операциях (которые имеют решающее значение для понимания представленных здесь концепций), может найти обзор таких тем в *разд. П3.2* и *П3.3*.

8.2.1. Нотация

Как мы видели в предыдущих главах, данные, связанные с *контролируемыми* задачами регрессии и классификации, всегда поступают в виде пар "вход — выход". Такая дихотомия отсутствует в задачах обучения без учителя, в которых типичный набор данных записывается просто как множество из P (входных) точек

$$\{\mathbf{x}_1,\ \mathbf{x}_2,\ ...,\ \mathbf{x}_P\} \tag{8.1}$$

или $\{\mathbf{x}_p\}_{p=1}^{P}$ для краткости, все из которых располагаются в одном и том же N-мерном пространстве. В оставшейся части этого раздела мы будем считать, что

наш набор данных был *среднецентрированным*: путем простой и полностью обратимой операции, которая предполагает вычитание среднего значения набора данных вдоль каждой входной размерности так, чтобы она пересекала начало координат.

Рассуждая о точках в многомерном векторном пространстве, мы можем представить их либо как графические *точки* (рис. 8.1, *слева*), либо как графические стрелки, идущие от начала координат (см. рис. 8.1, *в центре*). Первый рисунок отражает наше представление о регрессионных и классификационных данных до текущего момента, поскольку просто эстетичнее представить линейную регрессию как подгонку прямой к *разбросу графических точек*, а *не коллекции стрелок*. Однако эквивалентный последний взгляд (т. е. взгляд на многомерные точки в качестве стрелок) является традиционным способом изображения векторов, например, в любом стандартном учебнике по линейной алгебре. При обсуждении методов обучения без учителя часто полезно визуализировать точки в N-мерном пространстве, используя оба этих общепринятых способа: некоторые в виде *графических точек*, а некоторые в виде *графических стрелок* (рис. 8.1, *справа*). Векторы, нарисованные в виде стрелок, являются конкретными точками данных, часто именуемыми *базисом* или *охватывающим множеством*[1] *векторов*, над которыми мы стремимся представить любую другую точку данных в пространстве.

Условно мы обозначаем охватывающее множество через

$$\{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_K\} \tag{8.2}$$

или для краткости $\{\mathbf{c}_k\}_{k=1}^{K}$.

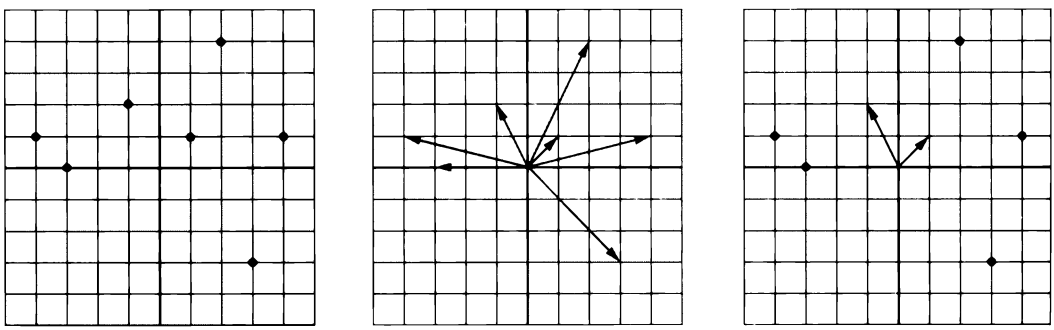


Рис. 8.1. Двумерные точки данных, изображенные в виде графических точек (*слева*), графических стрелок (*в центре*) и смеси того и другого (*справа*). Те, что показаны стрелками справа, представляют собой *базис* или *охватывающее множество*, над которым мы стремимся представить каждую точку данных во всем пространстве

[1] Также генерирующим множеством (spanning set). См. **https://en.wikipedia.org/wiki/Linear_span**, а также **https://ru.qwe.wiki/wiki/Linear_span**. — *Прим. пер.*

8.2.2. Совершенное представление данных с использованием фиксированных охватывающих множеств

Считается, что охватывающее множество способно совершенно представлять все P наших точек данных, если мы можем выразить каждую точку $\mathbf{x}_p$ как некоторую линейную комбинацию членов нашего охватывающего множества в виде

$$\sum_{k=1}^{K} \mathbf{c}_k w_{p,k} = \mathbf{x}_p, \quad p = 1, \ ..., \ P. \tag{8.3}$$

Вообще говоря, два простых условия, в случае если они удовлетворяются охватывающим множеством векторов $\{\mathbf{c}_k\}_{k=1}^{K}$, гарантируют, что все P равенств в уравнении (8.3) будут соблюдаться независимо от используемого набора данных $\{\mathbf{x}_p\}_{p=1}^{P}$: во-первых, $K = N$, или, другими словами, число охватывающих векторов совпадает с размерностью данных[2], и во-вторых, все охватывающие векторы линейно независимы[3]. Для такого охватывающего множества уравнение (8.3) можно записать компактнее:

$$\mathbf{C}\mathbf{w}_p = \mathbf{x}_p, \quad p = 1, \ ..., \ P, \tag{8.4}$$

где охватывающая матрица $\mathbf{C}$ формируется путем укладки охватывающих векторов по столбцам

$$\mathbf{C} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{c}_1 & \mathbf{c}_2 & & \mathbf{c}_N \\ | & | & \cdots & | \end{bmatrix}, \tag{8.5}$$

а линейно-комбинационные веса укладываются в столбцовые векторы $\mathbf{w}_p$:

$$\mathbf{w}_p = \begin{bmatrix} w_{p,1} \\ w_{p,2} \\ \vdots \\ w_{p,N} \end{bmatrix} \tag{8.6}$$

для всех $p = 1, \ ..., \ P$.

Для того чтобы настроить веса в каждом $\mathbf{w}_p$, мы можем сформировать соответствующую стоимостную функцию наименьших квадратов (как мы делали несколько раз ранее, например, в случае с линейной регрессией в *разд. 5.2*), которая при минимизировании заставляет равенства в уравнении (8.4) соблюдаться:

$$g(\mathbf{w}_1, ..., \mathbf{w}_P) = \frac{1}{P}\sum_{p=1}^{P} \left\| \mathbf{C}\mathbf{w}_p - \mathbf{x}_p \right\|_2^2. \tag{8.7}$$

---

[2] В противном случае, если $K < N$, некоторые части пространства, определенно, будут находиться вне досягаемости охватывающих векторов.

[3] *См. разд. П3.2.4*, если вы не знакомы с понятием линейной независимости.

Эту стоимостную функцию наименьших квадратов можно минимизировать с помощью любого метода локальной оптимизации. В частности, мы можем использовать первопорядковое условие (*см. разд. 3.2*) в каждом весовом векторе $\mathbf{w}_p$ *независимо*, причем соответствующая первопорядковая система принимает форму

$$\mathbf{C}^T\mathbf{C}\mathbf{w}_p = \mathbf{C}^T\mathbf{x}_p. \tag{8.8}$$

Она представляет собой симметричную линейную систему $N \times N$, которую можно легко решить численно (см., например, *разд. 3.2.2* и пример 3.6). После ее решения оптимально настроенный весовой вектор $\mathbf{w}_p^\star$ для точки $\mathbf{x}_p$ часто называется *кодировкой* точки над охватывающей матрицей $\mathbf{C}$. Фактическая линейная комбинация охватывающих векторов для каждой точки $\mathbf{x}_p$ (т. е. $\mathbf{C}\mathbf{w}_p^\star$) также называется *декодировкой* точки.

Пример 8.1. Кодировка данных

На рис. 8.2, *слева* показан двумерный ($N = 2$)набор данных, центрированный в начале координат вместе с охватывающей матрицей $\mathbf{C} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$, показанной в виде двух красных стрелок. Минимизируя стоимостную функцию наименьших квадратов из уравнения (8.7), на рис. 8.2, *справа* представлена *кодированная* версия этих данных, построенная на графике в новом пространстве, координатные оси которого теперь совпадают с двумя охватывающими векторами.

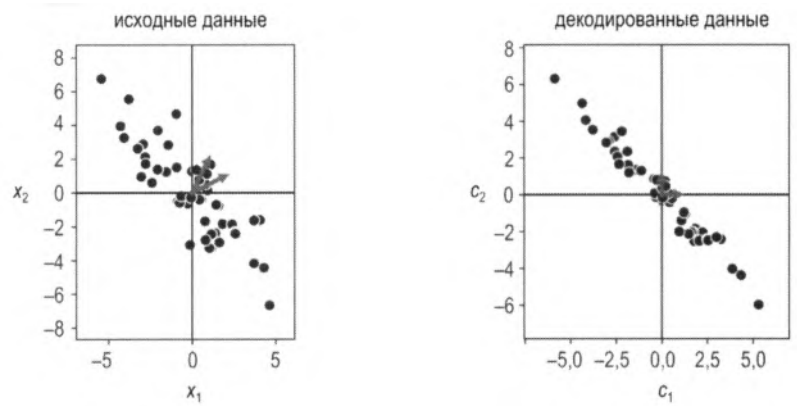


Рис. 8.2. Иллюстрация примера 8.1. Набор данных (*слева*) с охватывающими векторами, показанными красными стрелками, и его кодировкой (*справа*)

8.2.3. Совершенное представление данных с использованием фиксированных ортонормальных охватывающих множеств

Ортонормальный базис, или ортонормальное охватывающее множество, — это совершенно особый вид охватывающего множества, элементы которого, во-первых,

имеют единичную длину, и во-вторых, перпендикулярны или ортогональны друг другу. Алгебраически это означает, что векторы, принадлежащие ортонормальному базису, удовлетворяют условию

$$\mathbf{c}_i^T \mathbf{c}_j = \begin{cases} 1, & \text{если } i = j; \\ 0, & \text{если } i \neq j, \end{cases} \tag{8.9}$$

которое может быть выражено эквивалентным образом, но компактнее, в терминах охватывающей матрицы $\mathbf{C}$, как

$$\mathbf{C}^T \mathbf{C} = \mathbf{I}_{N \times N}. \tag{8.10}$$

Вследствие этого очень специального свойства ортонормальных охватывающих множеств мы можем найти решение для идеальных весов $\mathbf{w}_p$ (или *кодировки*) точки $\mathbf{x}_p$ сразу же, т. к. первопорядковое решение в уравнении (8.8) упрощается до

$$\mathbf{w}_p = \mathbf{C}^T \mathbf{x}_p. \tag{8.11}$$

Другими словами, кодировка обходится в огромной степени дешевле, когда наше охватывающее множество является ортонормальным, т. к. не осталось системы уравнений, для которой требуется найти решение, и мы можем получить кодировку каждой точки данных непосредственно простым матрично-векторным умножением.

Подставив эту форму кодировки в множество равенств в уравнении (8.4), получим

$$\mathbf{C}\mathbf{C}^T \mathbf{x}_p = \mathbf{x}_p, \quad p = 1, \ ..., \ P. \tag{8.12}$$

Мы называем эту формулу формулой автокодировщика, поскольку она выражает то, как точка $\mathbf{x}_p$ сначала *кодируется* (посредством $\mathbf{w}_p = \mathbf{C}^T \mathbf{x}_p$), а затем *декодируется* обратно в себя ($\mathbf{C}\mathbf{w}_p = \mathbf{C}\mathbf{C}^T \mathbf{x}_p$). Это обусловлено тем, что для ортонормальных охватывающих множеств мы также имеем $\mathbf{C}^T \mathbf{C} = \mathbf{I}_{N \times N}$, и два преобразования, которые мы применяем к данным (*кодировочное* преобразование $\mathbf{C}^T$ и *декодировочное* преобразование $\mathbf{C}$), являются обратными операциями.

8.2.4. Несовершенное представление данных с использованием фиксированных охватывающих множеств

Для того чтобы иметь возможность представлять данные совершенным образом, в двух предыдущих подразделах мы исходили из того, что число линейно независимых охватывающих векторов K и окружающая входная размерность N были идентичны. Когда $K < N$, мы больше не можем точно представлять все возможные точки данных во входном пространстве. Вместо этого мы можем только надеяться *аппроксимировать*, насколько это возможно, наш набор данных как

$$\mathbf{C}\mathbf{w}_p \approx \mathbf{x}_p, \quad p = 1, \ ..., \ P. \tag{8.13}$$

Данная формула аналогична уравнению (8.4), за исключением того, что теперь $\mathbf{C}$ и $\mathbf{w}_p$ являются соответственно $N \times K$-матрицей и столбцовым $K \times 1$-вектором, где $K < N$.

Для того чтобы усвоить надлежащие кодировки для наших данных, мы по-прежнему стремимся минимизировать стоимость наименьших квадратов в уравнении (8.7) и по-прежнему можем использовать систему первого порядка для отыскания независимого решения для каждого $\mathbf{w}_p$, как в уравнении (8.8). Геометрически, при решении стоимости наименьших квадратов мы стремимся отыскать наилучшее K-мерное подпространство, на которое можно спроецировать наши точки данных, как показано на рис. 8.3. Когда *кодировка* $\mathbf{w}_p$ оптимально вычисляется для точки $\mathbf{x}_p$, ее *декодировка* $\mathbf{C}\mathbf{w}_p$ является как раз той проекцией $\mathbf{x}_p$ на подпространство, которое охватывается матрицей $\mathbf{C}$. Это называется *проекцией*, потому что, как показано на рисунке, представление задается путем проецирования или отбрасывания $\mathbf{x}_p$ перпендикулярно на подпространство.

Подобно тому, что мы видели в предыдущем подразделе, если наше охватывающее множество из K элементов является ортонормальным, то соответствующая формула для каждого кодировочного вектора $\mathbf{w}_p$ упрощается до уравнения (8.11), а формула *автокодировщика*, показанная ранее в уравнении (8.12), становится такой:

$$\mathbf{C}\mathbf{C}^T\mathbf{x}_p \approx \mathbf{x}_p, \quad p = 1, ..., P. \tag{8.14}$$

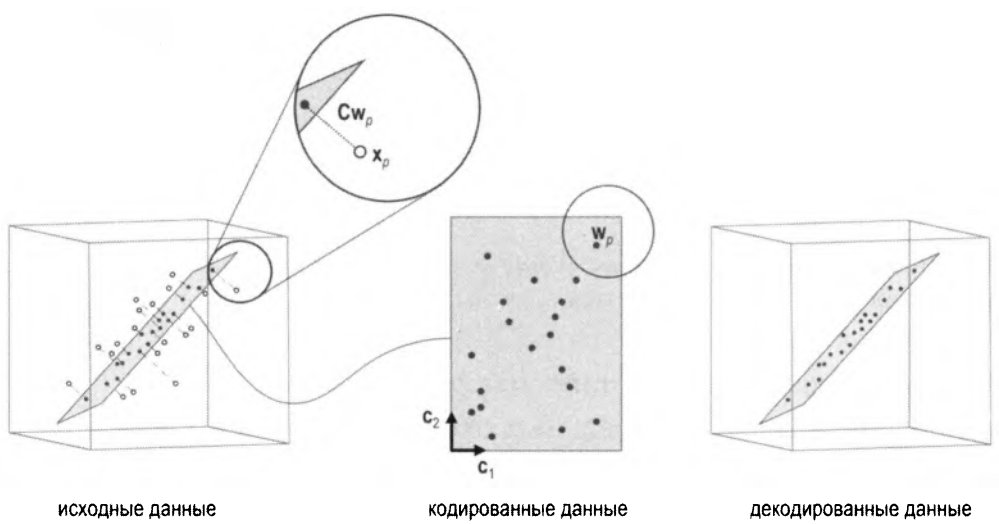


Рис. 8.3. Множество точек $\mathbf{x}_p$ в $N = 3$ размерностях вместе с линейным подпространством, охватываемым $K = 2$ векторами $\mathbf{c}_1$ и $\mathbf{c}_2$, окрашенными в красный цвет (*слева*). Кодировочное пространство, охватываемое $\mathbf{c}_1$ и $\mathbf{c}_2$, где располагаются наши кодированные векторы $\mathbf{w}_p$ (*в центре*). Проецируемые или декодированные версии каждой точки $\mathbf{x}_p$ показаны в подпространстве, охватываемом $\mathbf{C} = [\mathbf{c}_1 \ \ \mathbf{c}_2]$. Декодированная версия исходной точки $\mathbf{x}_p$ выражается как $\mathbf{C}\mathbf{w}_p$ (*справа*)

Другими словами, поскольку $K < N$, преобразования кодировки $\mathbf{C}^T$ и декодировки $\mathbf{C}$ уже не являются полностью обратными операциями друг друга.

8.3. Линейный автокодировщик и анализ главных компонент

Самый фундаментальный метод обучения без учителя, именуемый *анализом главных компонент* (principal component analysis, PCA), следует непосредственно из нашего обсуждения в предыдущем разделе относительно представлений фиксированных охватывающих множеств с одной важной оговоркой: вместо того, чтобы просто усваивать надлежащие веса с целью наилучшим образом представлять входные данные над конкретным *фиксированным* охватывающим множеством, мы также *усваиваем* надлежащее охватывающее множество.

8.3.1. Усвоение надлежащих охватывающих множеств

Представьте, что мы вернулись к предыдущему разделу, но вместо того, чтобы исходить из данности нам $K \leq N$ *фиксированных* охватывающих векторов, над которыми можно представить наши среднецентрированные входные данные как

$$\mathbf{C}\mathbf{w}_p \approx \mathbf{x}_p, \qquad p = 1, \ ..., \ P, \tag{8.15}$$

мы стремились *усвоить* наилучшие охватывающие векторы, чтобы сделать эту аппроксимацию как можно более плотной. Для этого мы могли бы просто добавить $\mathbf{C}$ в набор параметров нашей функции наименьших квадратов из уравнения (8.7), получив

$$g(\mathbf{w}_1, ..., \mathbf{w}_P, \mathbf{C}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\mathbf{w}_p - \mathbf{x}_p\right\|_2^2, \tag{8.16}$$

которую мы могли бы затем минимизировать, чтобы усвоить наилучшее возможное множество весов с $\mathbf{w}_1$ по $\mathbf{w}_P$, а также охватывающую матрицу $\mathbf{C}$. Эта стоимостная функция наименьших квадратов, которая обычно является невыпуклой[4], может быть надлежаще минимизирована с помощью любого числа методов локальной оптимизации, включая градиентный спуск (*см. разд. 3.5*) и координатный спуск (*см. разд. 3.2.2*).

Пример 8.2. Усвоение надлежащего охватывающего множества посредством градиентного спуска

В этом примере мы используем градиентный спуск для минимизирования стоимости наименьших квадратов из уравнения (8.16), чтобы узнать наилучшее двумерное

[4] Тем не менее она является выпуклой в каждом $\mathbf{w}_p$, оставляя все другие веса и $\mathbf{C}$ фиксированными, и выпуклой в $\mathbf{C}$, оставляя все весовые векторы $\mathbf{w}_p$ фиксированными.

($K = 2$) подпространство для среднецентрированного трехмерного ($N = 3$) набора данных из $P = 100$ точек, показанного на рис. 8.4, *слева*.

В дополнение к исходным данным левая часть рисунка показывает усвоенные охватывающие векторы в виде красных стрелок и соответствующее подпространство, окрашенное в светло-розовый цвет. Это лучшее представление двумерного подпространства для входных данных. На рис. 8.4, *в центре* показаны соответствующие усвоенные кодировки $\mathbf{w}_p$ исходных входных данных $\mathbf{x}_p$ в пространстве, охватываемом двумя восстановленными охватывающими векторами. На правой части рисунка снова представлено исходное пространство данных, а также декодированные данные, т. е. проекция каждой исходной точки данных на наше усвоенное подпространство.

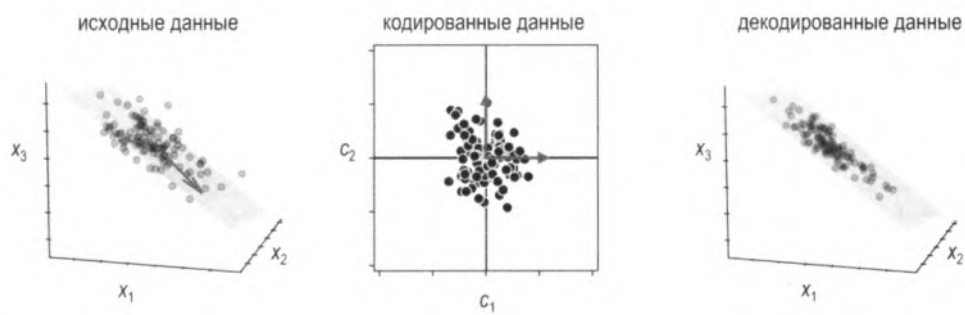


Рис. 8.4. Иллюстрация примера 8.2

8.3.2. Линейный автокодировщик

Как подробно описано в *разд. 8.2.4*, если наши K охватывающих векторов, конкатенированных по столбцам и формирующих таким образом охватывающую матрицу $\mathbf{C}$, являются ортонормальными, то кодирование каждой $\mathbf{x}_p$ может быть записано просто как $\mathbf{w}_p = \mathbf{C}^T\mathbf{x}_p$. Если мы подключим это простое решение для $\mathbf{w}_p$ в p-е слагаемое стоимости наименьших квадратов из уравнения (8.16), то получим стоимость, которая является функцией *только* $\mathbf{C}$:

$$g(\mathbf{C}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\mathbf{C}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2. \tag{8.17}$$

Мы можем думать об этих наименьших квадратах как об обеспечении того, чтобы формула автокодировщика, показанная в уравнении (8.14), соблюдалась при надлежащем минимизировании, и поэтому ее часто называют *линейным автокодировщиком*. Вместо того чтобы получить кодировочную/декодировочную схему для каждой точки данных, минимизируя эту стоимостную функцию, мы ее *усваиваем* (заучиваем).

Даже если мы пришли к линейному автокодировщику, приняв за основу, что наша охватывающая матрица $\mathbf{C}$ является ортонормальной, нам не нужно обязательно

ограничивать нашу минимизацию уравнения (8.17), чтобы обеспечить соблюдение этого условия, потому что, как показано в *разд. 8.9*, минимумы линейного автокодировщика *всегда* ортонормальны (*см. разд. 8.9.1*).

Пример 8.3. Усвоение линейного автокодировщика с использованием градиентного спуска

На рис. 8.5, *слева* показан среднецентрированный двумерный набор данных вместе с единственным охватывающим вектором (т. е. $K = 1$), *усвоенным* на данных путем минимизирования линейной стоимостной функции автокодировщика из уравнения (8.17) с использованием градиентного спуска. Оптимальный вектор показан красной стрелкой на левой части рисунка, соответствующие кодированные данные представлены в центре рисунка, а декодированные данные — на правой части вместе с оптимальным подпространством для данных (прямая), показанным красным цветом.

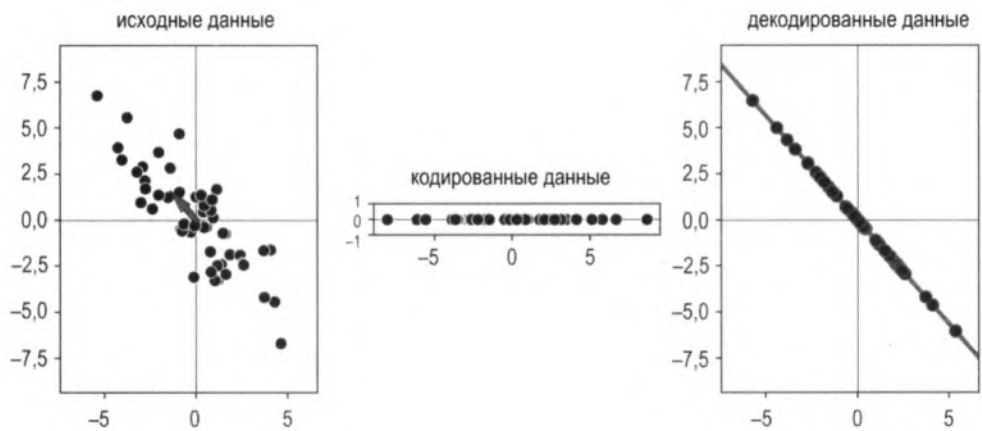


Рис. 8.5. Иллюстрация примера 8.3

8.3.3. Анализ главных компонент

Стоимость линейного автокодировщика в уравнении (8.17) может иметь много минимизаторов, из которых множество *главных компонент* является особенно важным. Охватывающее множество главных компонент всегда обеспечивает единообразный скелет для набора данных, причем его члены указывают на наибольшие направления *ортогональной дисперсии набора данных*. Задействование этого конкретного для линейного автокодировщика решения на практике часто называется *анализом главных компонент* (Principal Component Analysis, PCA).

Эта идея проиллюстрирована для двумерного ($N = 2$) набора данных на рис. 8.6, где общее эллиптическое распределение данных показано светло-серым цветом. Шкалированная версия первой главной компоненты этого набора данных (в виде длинной красной стрелки) указывает на направление, в котором набор данных наиболее разбросан, также именуемое его наибольшим направлением дисперсии. Шкалированная версия второй главной компоненты (короткая красная стрелка) указы-

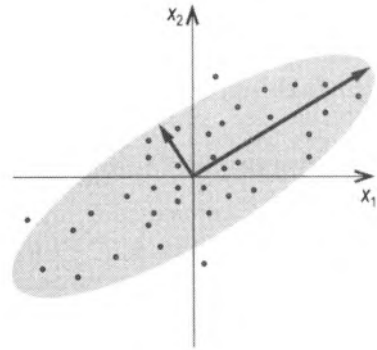


Рис. 8.6. Прототипический набор данных со шкалированными версиями его первой и второй главных компонент, показанными соответственно длинной и короткой красными стрелками

вает на следующее наиболее важное направление, в котором набор данных разбросан, *ортогонально* первому.

Как мы покажем в *разд. 8.9.2*, этот специальный ортонормальный минимизатор линейного автокодировщика задается собственными векторами так называемой *матрицы ковариаций* данных. Обозначив через $\mathbf{X}$ матрицу данных размера $N \times N$, состоящую из наших среднецентрированных входных точек, уложенных по столбцам

$$\mathbf{X} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{x}_1 & \mathbf{x}_2 & & \mathbf{x}_P \\ | & | & \cdots & | \end{bmatrix}, \tag{8.18}$$

можно определить матрицу ковариаций как $N \times N$-матрицу $\frac{1}{P}\mathbf{X}\mathbf{X}^T$. Обозначив собственное разложение (*см. разд. П3.4*) матрицы ковариаций через

$$\frac{1}{P}\mathbf{X}\mathbf{X}^T = \mathbf{V}\mathbf{D}\mathbf{V}^T, \tag{8.19}$$

можно задать главные компоненты ортонормальными собственными векторами в $\mathbf{V}$, а дисперсию в каждом направлении (главной компоненты) — в точности соответствующими неотрицательными собственными значениями в диагональной матрице $\mathbf{D}$.

Пример 8.4. Главные компоненты

На рис. 8.7, *слева* показаны среднецентрированные данные, впервые представленные на рис. 8.5, вместе с двумя их главными компонентами (указывающими в двух ортогональных направлениях наибольшей дисперсии в наборе данных), обозначенными красными стрелками. Справа показана *кодированная* версия данных в пространстве, где главные компоненты находятся на одной линии с координатными осями.

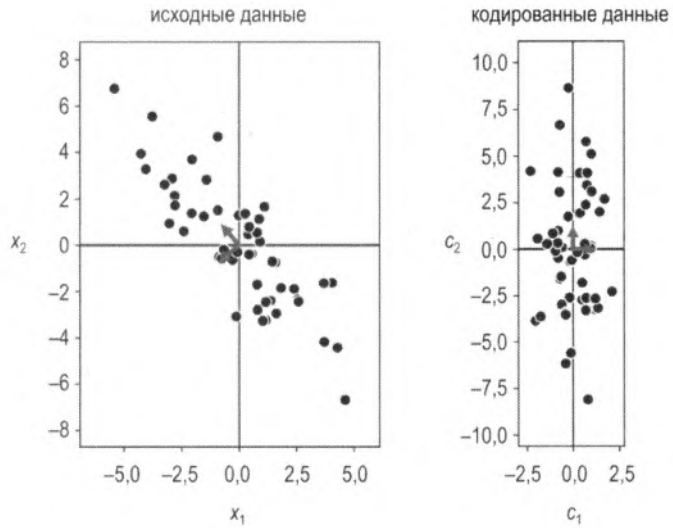


Рис. 8.7. Иллюстрация примера 8.4

Пример 8.5. Предупреждающий пример!

В то время как анализ главных компонент (PCA) технически может использоваться для уменьшения размерности данных в сценарии предсказательного моделирования (в надежде на повышение точности, времени вычислений и т. д.), этот метод может вызвать серьезные проблемы в случае классификации. На рис. 8.8 показано уменьшение размерности пространства признаков с помощью PCA на моделируемом двухклассовом наборе данных, где эти два класса линейно разделимы. Поскольку идеальное одномерное подпространство для данных в этом случае проходит (почти) параллельно идеальному линейному классификатору, проецирование полного набора данных на это подпространство полностью разрушает межклассо-

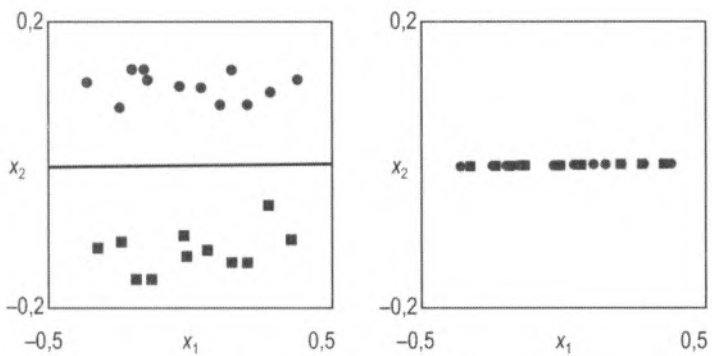


Рис. 8.8. Иллюстрация примера 8.5. Классификационный набор данных, состоящий из двух линейно разделяемых классов. Идеальное одномерное подпространство, полученное с помощью PCA, показано черным цветом (*слева*). Уменьшение размерности пространства признаков путем проецирования данных на это подпространство полностью разрушает исходную разделимость данных (*справа*)

вое разделение. Именно по этой причине, хотя и принято *сферировать* классификационные данные, используя PCA, как подробно описано в *разд. 9.5*, необходимо быть чрезвычайно осторожным, применяя PCA в качестве инструмента уменьшения размерности в случае классификации или когда данные изначально не расположены в линейном подпространстве или вблизи него.

8.3.4. Имплементация на Python

Ниже приведена имплементация на Python, участвующая в вычислении главных компонент набора данных, включая центрирование данных, вычисление главных компонент и PCA-кодирование. Эта имплементация активно задействует линейно-алгебраический подмодуль библиотеки NumPy.

Сначала мы центрируем данные, используя короткую имплементацию:

```
1| # центрировать входной набор данных X
2| def center(X):
3|     X_means = np.mean(X,axis=1)[:,np.newaxis]
4|     X_centered = X - X_means
5|     return X_centered
```

Затем вычисляем главные компоненты среднецентрированных данных:

```
1| # функция для вычисления главных компонент входного набора данных X
2| def compute_pcs(X,lam):
3|     # создать матрицу ковариаций данных
4|     P = float(X.shape[1])
5|     Cov = 1/P*np.dot(X,X.T) + lam*np.eye(X.shape[0])
6|
7|     # использовать функцию numpy для вычисления собственных векторов/значений
8|     D,V = np.linalg.eigh(Cov)
9|     return D,V
```

Обратите внимание, что на практике часто бывает полезно слегка *регуляризовать* матрицу перед вычислением ее собственных значений/векторов, чтобы избежать естественных проблем численной нестабильности, связанных с их вычислением. Здесь это означает добавление малой взвешенной матрицы тождественности $\lambda \mathbf{I}_{N\times N}$, где $\lambda \geq 0$ — это некоторое малое значение (например, 10^{-5}), в матрицу ковариаций данных перед вычислением ее собственных значений/векторов. Одним словом, во избежание вычислительных проблем мы обычно вычисляем главные компоненты регуляризованной матрицы ковариаций $\frac{1}{P}\mathbf{X}\mathbf{X}^T + \lambda \mathbf{I}_{N\times N}$ вместо сырой матрицы ковариаций как таковой. Отсюда следует добавление выражения `lam*np.eye(X.shape[0])` в строке 5 приведенной выше имплементации.

8.4. Рекомендательные системы

В этом разделе мы обсудим основополагающую линейную *рекомендательную систему* — популярный вычислительный каркас инструмент обучения без учителя, обычно задействуемый компаниями для автоматической рекомендации товаров и услуг своим клиентам. Однако с точки зрения машинного обучения описанная здесь базовая рекомендательная система — это просто незначительное видоизменение в нашей стержневой методике обучения без учителя: анализе главных компонент.

8.4.1. Мотивация

Рекомендательные системы широко используются сегодня в электронной коммерции, предоставляя клиентам персонализированные рекомендации по товарам и услугам на основе предыдущей истории покупок и рейтингов потребителей, а также истории рейтингов похожих клиентов. К примеру, провайдер фильмов, наподобие компании Netflix с миллионами пользователей и десятками тысяч фильмов, записывает отзывы и рейтинги пользователей (обычно в виде числа по шкале от 1 до 5, где 5 — это наиболее благоприятный рейтинг) в большую матрицу, такую как та, что показана на рис. 8.9. Эти матрицы очень редко заполняются, т. к. отдельный клиент, вероятнее всего, оценивает только малую часть имеющихся фильмов.

При наличии такого рода данных с рейтингами товаров онлайновые кинотеатры и коммерческие веб-сайты часто используют методику обучения без учителя, которую мы обсудим в этом разделе, в качестве главного инструмента для предоставле-

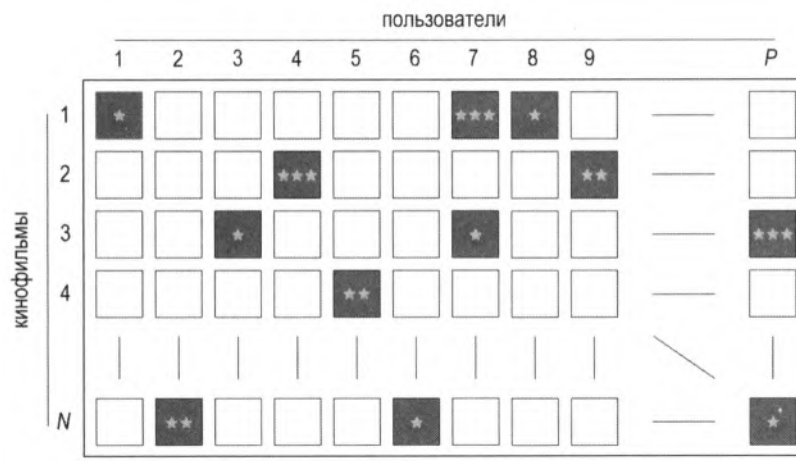


Рис. 8.9. Пример матрицы рейтингов фильмов заполнен очень скудно, и каждый пользователь оценил только очень малое число фильмов. На этой диаграмме фильмы пронумерованы вдоль строк, а пользователи — вдоль столбцов. Для того чтобы правильно рекомендовать пользователям фильмы для просмотра, мы пытаемся разумно угадать недостающие значения этой матрицы, а затем рекомендуем фильмы, которые, согласно нашему предсказанию, пользователи высоко оценят (и, следовательно, получат наибольшее удовольствие)

ния клиентам персонализированных рекомендаций относительно того, что они хотели бы приобрести дальше. С методикой порождения персонализированных рекомендаций, которую мы обсуждаем здесь, мы стремимся сначала разумно угадать значения отсутствующих записей в матрице рейтингов. Далее, для того чтобы рекомендовать новый товар конкретному пользователю, мы проверяем нашу заполненную матрицу рейтингов для товаров, которые, согласно нашему предсказанию, пользователь высоко оценит (и, следовательно, получит удовольствие), и рекомендуем их.

8.4.2. Нотация и моделирование

С рекомендательной системой мы продолжаем использовать знакомую нам нотацию $\{\mathbf{x}_1, ..., \mathbf{x}_P\}$ для обозначения входных данных, каждый элемент которых имеет размерность N. В этом приложении точка $\mathbf{x}_p$ обозначает рейтинговый вектор нашего p-го клиента из всех N возможных товаров, имеющихся для рейтинговой оценки. Число товаров N, вероятно, довольно велико, причем настолько, что каждый клиент имеет возможность приобрести и просмотреть лишь очень малую выборку из них, что делает вектор $\mathbf{x}_p$ весьма разреженным (с любыми оценками, которые пользователь p ввел в систему). Обозначим индексное множество этих непустых записей в $\mathbf{x}_p$ через

$$\Omega_p = \left\{(j, p) \mid j\text{-й элемент в } \mathbf{x}_p \text{ заполнен}\right\}. \tag{8.20}$$

Поскольку наша цель состоит в том, чтобы заполнить недостающие записи каждого входного вектора $\mathbf{x}_p$, у нас нет выбора, кроме как принимать допущения о том, каковы вкусы пользователей в целом. Самое простое допущение, которое мы можем принять, состоит в том, что вкусы каждого пользователя могут быть выражены как линейная комбинация некоторого малого множества фундаментальных профилей пользовательских вкусов. Например, в случае фильмов эти профили могут включать поклонника романтических фильмом, поклонника комедийных фильмов, поклонника боевиков и т. д. Относительно малое число таких категорий или типов пользователей по сравнению с суммарным числом пользователей обеспечивает полезную основу для разумного угадывания значений, отсутствующих в наборе пользовательских рейтингов. Формально это означает, что мы исходим из того, что существует некоторое идеальное охватывающее множество K векторов основополагающих вкусов (которые мы можем упаковать в $N \times K$-матрицу $\mathbf{C}$), в результате чего каждый вектор $\mathbf{x}_p$ может быть действительно выражен как линейная комбинация:

$$\mathbf{C}\mathbf{w}_p \approx \mathbf{x}_p, \quad p = 1, ..., P. \tag{8.21}$$

Для того чтобы затем усвоить и охватывающее множество $\mathbf{C}$, и каждый весовой вектор $\mathbf{w}_p$, мы могли бы первоначально предложить минимизировать стоимость наименьших квадратов, аналогичную той, которая показана в уравнении (8.16). Однако наши входные данные теперь неполны, т. к. у нас есть доступ только к тем

элементам, которые индексированы множеством Ω_p для $\mathbf{x}_p$. Следовательно, мы можем минимизировать стоимость наименьших квадратов только над этими элементами, т. е.

$$g(\mathbf{w}_1, ..., \mathbf{w}_P, \mathbf{C}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\left\{\mathbf{C}\mathbf{w}_p - \mathbf{x}_p\right\}\big|_{\Omega_p}\right\|_2^2. \quad (8.22)$$

Нотация $\{\mathbf{v}\}|_{\Omega_p}$ здесь обозначает взятие только этих элементов $\mathbf{v}$ в индексном множестве Ω_p. Поскольку стоимость наименьших квадратов здесь определяется только по выборочному числу индексов, мы не можем использовать какие-либо ортонормальные решения для этой стоимости или сконструировать стоимость, подобно линейному автокодировщику в уравнении (8.17). Однако мы легко можем использовать схемы градиентного спуска (*см. разд. 3.5*) и координатного спуска (*см. разд. 3.2.2*), чтобы надлежаще ее минимизировать.

8.5. Кластеризация методом *K* средних

Предмет этого раздела, *алгоритм K средних*, является элементарным примером еще одного набора методов обучения без учителя, именуемых *алгоритмами кластеризации*. В отличие от анализа главных компонент (PCA), который был разработан для уменьшения окружающей размерности (или признаковой размерности) пространства данных, алгоритмы кластеризации предназначены для (надлежащего) уменьшения числа точек (или размерности данных) в наборе и тем самым помогают нам лучше понять его структуру.

8.5.1. Представление набора данных посредством кластеров

Один из способов упростить набор данных состоит в том, чтобы сгруппировать близлежащие точки в *кластеры*. Возьмем следующий ниже набор двумерных точек данных, показанный на рис. 8.10, *слева*. Когда вы внимательно рассмотрите представленные там данные, то увидите, что они естественным образом делятся на три группы, или *кластера*, потому что у вас мозг работает подобно алгоритму кластеризации.

На рис. 8.10, *справа* показано визуальное представление каждого кластера, включая границу каждого кластера, нарисованную в виде уникально окрашенной сплошной кривой. Также отмечены центры кластеров символом звездочки, который соответствует уникальному цвету границы каждого кластера. На жаргоне машинного обучения эти центры кластеров часто именуются кластерными *центроидами*. Центроиды здесь позволяют нам рассуждать о наборе данных в широком смысле — вместо $P = 12$ точек мы можем рассуждать о нашем наборе данных грубо в терминах этих $K = 3$ кластерных центроидов, поскольку каждый представляет собой порцию данных.

Каким образом математически можно описать сценарий кластеризации, который мы беспрепятственно видим, когда рассматриваем точки/кластеры (см. рис. 8.10)?

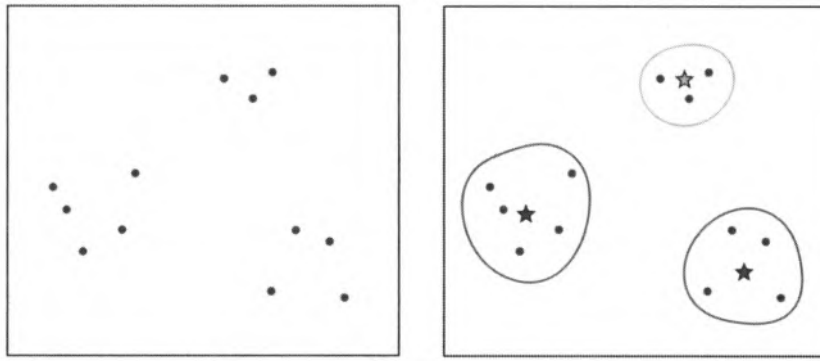


Рис. 8.10. Двумерный набор данных с $P = 12$ точками данных (*слева*). Показаны данные, которые естественным образом сгруппировались в $K = 3$ кластера (*справа*). Точки, геометрически близкие друг к другу, принадлежат одному и тому же кластеру, и каждая граница кластера грубо обозначена уникально окрашенной сплошной кривой. Центр каждого кластера (центроид) отмечен символом звездочки, окрашенным в соответствии с границей своего кластера

Начнем с некоторой нотации. Как и в предыдущих разделах, мы обозначим наше множество из P точек обобщенно через $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_P$. Для того чтобы все оставалось как можно общеприменимым, мы также обозначим через K число кластеров в наборе данных (например, в наборе данных на рис. 8.10 $K = 3$). Поскольку каждый кластер имеет центроид, для этого нам также нужна нотация, и мы будем использовать $\mathbf{c}_1$, $\mathbf{c}_2$, ..., $\mathbf{c}_K$, чтобы их обозначить, где $\mathbf{c}_k$ — это центроид k-го кластера. Наконец, нам понадобится нотация для обозначения множества точек, принадлежащих каждому кластеру. Обозначим множество индексов тех точек, которые принадлежат к k-му кластеру, как

$$S_k = \{p \,|\, \text{если } \mathbf{x}_p \text{ принадлежит к } k\text{-му кластеру}\}. \tag{8.23}$$

С учетом всей нашей нотации мы теперь можем лучше описать сценарий кластеризации, показанный на рис. 8.10. Предположим, что мы определили каждый кластер и его центроид "на глаз" (см. рис. 8.10, *справа*). Поскольку центроид обозначает центр кластера, то, предположительно, каждый из них может быть выражен как *среднее значение точек, отнесенных к своему кластеру*:

$$\mathbf{c}_k = \frac{1}{|S_k|} \sum_{p \in S_k} \mathbf{x}_p . \tag{8.24}$$

Эта формула подтверждает интуицию, что каждый центроид представляет собой часть данных — среднее значение тех точек, которые принадлежат каждому кластеру.

Далее мы можем математически констатировать очевидный и неявный факт о простом сценарии кластеризации, представленном на рис. 8.10: каждая точка принадлежит кластеру, к центроиду которого она ближе всего. Выразить это алгебраически для заданной точки $\mathbf{x}_p$ — значит просто сказать, что точка должна принадле-

жать кластеру, расстояние от которой до центроида $\left\|\mathbf{x}_p - \mathbf{c}_k\right\|_2$ является *минимальным*. Другими словами, точка $\mathbf{x}_p$ принадлежит или относится к кластеру $k^\star$, если

$$a_p = \underset{k=1,\ldots,K}{\arg\min} \left\|\mathbf{x}_p - \mathbf{c}_k\right\|_2 . \tag{8.25}$$

На жаргоне машинного обучения они называются кластерными *отнесениями* (assignments).

8.5.2. Усвоение кластеров для представления данных

Конечно, мы не хотим полагаться на наши зрительные возможности в вопросе идентификации кластеров в наборе данных. Нам нужен алгоритм, который сделает это за нас автоматически. К счастью, мы можем решить эту проблему довольно легко, используя описанный выше каркас для математического описания кластеров, и результирующий алгоритм называется *кластеризационным алгоритмом K средних*.

Для начала предположим, что мы хотим автоматически сгруппировать набор данных из P точек в K кластеров. Заметьте, что мы зафиксируем K и обратимся к тому, как надлежаще определиться с его значением, позже.

Поскольку мы не знаем, где расположены кластеры и их центроиды, мы можем начать со случайной догадки о местоположении наших K центроидов (мы должны с чего-то начать). Эта "случайная догадка" — наша инициализация — в отношении K центроидов может быть случайным подмножеством из K наших точек, случайным множеством K точек в пространстве данных или любым количеством других типов инициализаций. Имея первоначальные местоположения центроидов, мы можем определить кластерные отнесения, просто перебирая наши точки в цикле и для каждой $\mathbf{x}_p$ отыскивая ее ближайший центроид, используя формулу, которую мы видели выше:

$$a_p = \underset{k=1,\ldots,K}{\arg\min} \left\|\mathbf{x}_p - \mathbf{c}_k\right\|_2 . \tag{8.26}$$

Теперь у нас есть и первоначальная догадка о наших центроидах, и кластерные отнесения. Имея на руках наши кластерные отнесения, мы можем обновить местоположения наших центроидов — как среднее значение точек, недавно отнесенных к каждому кластеру:

$$\mathbf{c}_k = \frac{1}{|S_k|} \sum_{p \in S_k} \mathbf{x}_p . \tag{8.27}$$

Эти первые три шага — инициализация центроидов, отнесение точек к каждому кластеру и обновление расположения центроидов — показаны на рис. 8.11, *сверху*; эти шаги выполнены на наборе данных с рис. 8.10.

Для того чтобы уточнять наши центроиды/кластеры дальше, мы можем просто повторить описанный выше двухшаговый процесс: первый шаг — по-новому относя точки к кластерам на основе наших новых местоположений центроидов; второй

шаг — обновляя местоположения центроидов в качестве среднего арифметического этих точек, отнесенных к каждому кластеру. Вторая итерация алгоритма проиллюстрирована на конкретном примере на рис. 8.11, *снизу*. Мы можем прекратить наши действия, например, после передопределенного максимального числа итераций или когда центроиды кластеров не будут сильно менять свое местоположение от одной итерации к другой.

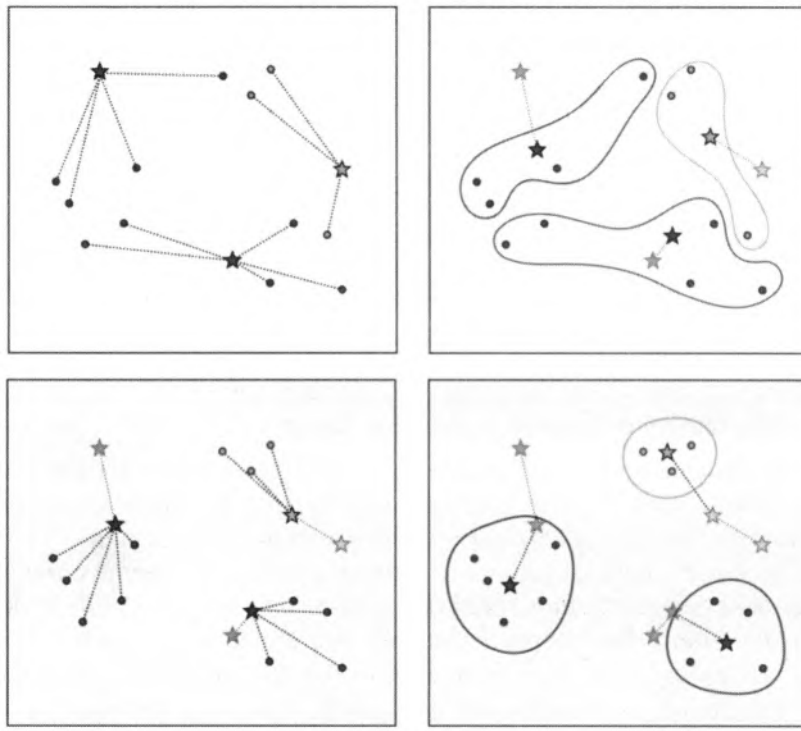


Рис. 8.11. Первые две итерации алгоритма *K* средних иллюстрируются с использованием набора данных, впервые показанного на рис. 8.10. Набор точек данных со случайными инициализациями центроидов и отнесениями (*слева сверху*). Местоположения центроидов обновляются по мере того, как среднее арифметическое точек назначается каждому кластеру (*справа сверху*). Отнесение точек к кластерам на основе обновленных местоположений центроидов (*слева снизу*). Обновленное местоположение центроидов, задаваемое средними значениями кластеров (*справа снизу*)

Пример 8.6. Влияние инициализации

Результат алгоритма, достигающий слабых минимумов, может оказать существенное влияние на качество усвоенных кластеров. Например, на рис. 8.12 мы используем двумерный набор данных с $K = 2$ отыскиваемыми кластерами. С первоначальными позициями центроидов, показанными на верхней панели, алгоритм *K* средних застревает в локальном минимуме и, следовательно, не способен надлежаще кластеризировать данные. Однако другая инициализация для одного из центроидов приводит к успешной кластеризации данных, как показано на нижней

панели рисунка. С целью преодоления этой трудности на практике мы часто выполняем алгоритм много раз с разными инициализациями, причем наилучшим решением является та, которая приводит к наименьшему значению некоторого целевого значения качества кластеров.

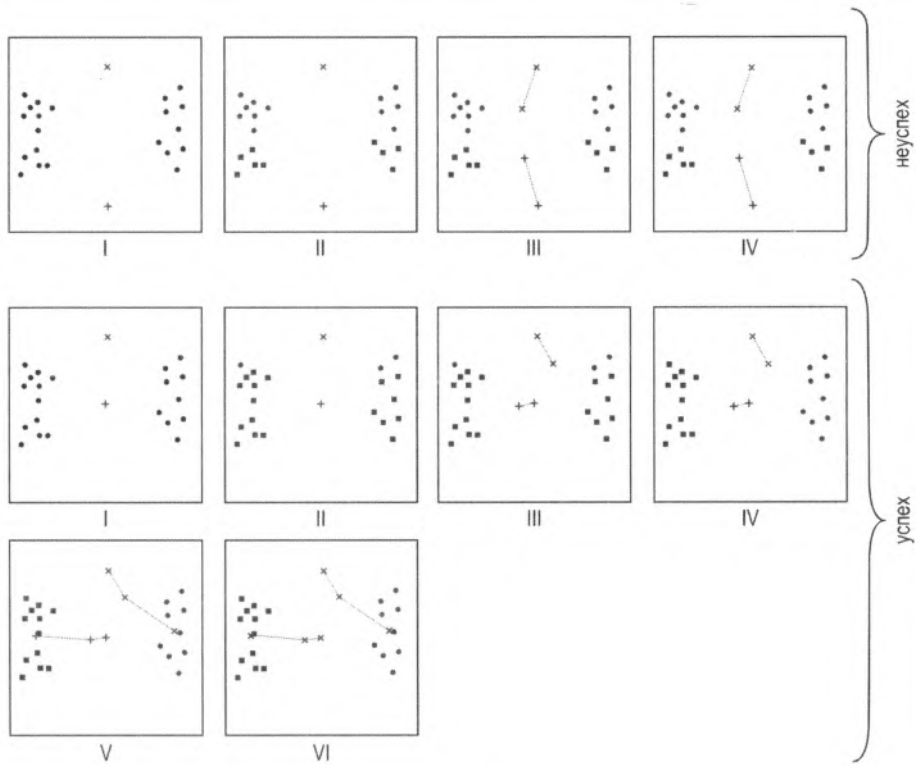


Рис. 8.12. Успех или неуспех метода K средних зависит от инициализации центроидов. *Вверху*: I — инициализируются два центроида; II — обновляется отнесение к кластерам; III — обновляются местоположения центроидов; IV — отсутствие изменений в отнесении точек данных к кластерам приводит к остановке алгоритма.
В центре и *внизу*: I — инициализируются два центроида, причем красный (в виде +) инициализируется по-другому; II — обновляется отнесение к кластерам; III — обновляются местоположения центроидов; IV — обновляется отнесение к кластерам; V — обновляются местоположения центроидов; VI — отсутствие изменений в отнесении точек данных к кластерам приводит к остановке алгоритма

Например, одной из метрик для определения наилучшей кластеризации из набора прогонов является *среднее расстояние от каждой точки до центроида ее кластера*, именуемое *средним внутрикластерным расстоянием*. Обозначив через $\mathbf{c}_{k_p}$ окончательный кластерный центроид p-й точки $\mathbf{x}_p$, среднее расстояние от каждой точки до соответствующего ей центроида можно записать как

$$\text{среднее внутрикластерное расстояние} = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{x}_p - \mathbf{c}_{k_p}\right\|_2 . \qquad (8.28)$$

Вычисляя его для каждого прогона K средних, мы выбираем окончательную кластеризацию, которая достигает *наименьшего* значения, как наилучшую кластеризующую расстановку.

Пример 8.7. Выбор идеального числа кластеров K

Для того чтобы определить оптимальную настройку параметра K, т. е. число кластеров, в которые следует группировать данные, мы, как правило, должны опробовать для K диапазон разных значений, выполнить алгоритм K средних в каждом случае и сравнить результаты, используя соответствующую метрику, такую как среднее внутрикластерное расстояние из уравнения (8.28). Конечно, если мы достигнем оптимальной кластеризации для каждого значения K (возможно, выполнив алгоритм несколько раз для каждого значения K), то внутрикластерное расстояние *всегда должно монотонно уменьшаться по мере увеличения значения* K, т. к. мы разбиваем набор данных на всё меньшие и меньшие порции.

Например, на рис. 8.13 представлены результаты выполнения десяти прогонов алгоритма K средних; при это мы варьировали значение K из $K = 1, ..., 10$ и сохраняли кластеризацию, которая обеспечивала наименьшее внутрикластерное расстояние для каждого значения K в наборе данных, показанном на рис. 8.13, *слева*. На рис. 8.13, *справа* построен график наилучшего значения расстояния, достигнутого для каждого значения K; указанный график на жаргоне машинного обучения часто упоминается как график *крутого склона* (или каменистой осыпи, scree plot).

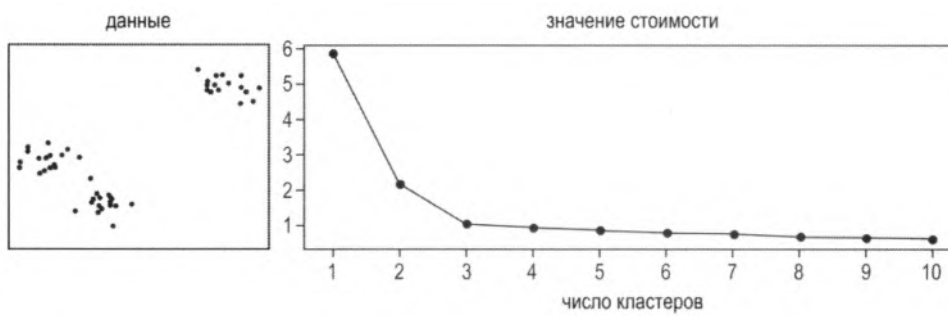


Рис. 8.13. Иллюстрация примера 8.7. Кластеризуемый набор данных (*слева*) и график крутого склона (*справа*)

Как и следовало ожидать, внутрикластерное расстояние монотонно уменьшается по мере увеличения K. Однако обратите внимание, что приведенный выше график крутого склона имеет *"локоть"* при $K = 3$, из чего следует, что увеличение числа кластеров с трех до четырех и далее уменьшает значение расстояния очень слабо. Исходя из этого, мы можем утверждать, что $K = 3$ является хорошим вариантом числа кластеров для этого конкретного набора данных (что также имеет смысл в настоящем случае, поскольку мы можем визуализировать набор данных и ясно видеть, что он имеет три кластера), поскольку любое меньшее число кластеров и внутрикластерное расстояние сравнительно велико, тогда как добавление новых кластеров не уменьшает общее внутрикластерное расстояние слишком сильно.

Этот пример иллюстрирует типичное использование графика крутого склона для принятия решения об идеальном числе кластеров K для K средних. Мы вычисляем, а затем строим график внутрикластерного расстояния над диапазоном значений для K и выбираем значение в "локте" графика. На практике это значение часто выбирается субъективно (путем визуального анализа графика крутого склона), как это было сделано здесь.

8.6. Общая методика матричной факторизации

В этом разделе мы свяжем воедино описанные в данной главе методы обучения без учителя, описывая их через своеобразную призму *матричной факторизации*.

8.6.1. Обучение без учителя и задачи матричной факторизации

Если мы компактно представим наши P точек входных данных, укомплектовав их по столбцам в матрицу данных $\mathbf{X}$, как в уравнении (8.18), то мы можем записать стоимостную функцию наименьших квадратов из уравнения (8.16), которая является базисом для анализа главных компонент (*см. разд. 8.3*), компактно:

$$g(\mathbf{W}, \mathbf{C}) = \frac{1}{P}\|\mathbf{CW} - \mathbf{X}\|_F^2 . \tag{8.29}$$

Здесь $\|\mathbf{A}\|_F^2 = \sum_{n=1}^{N}\sum_{p=1}^{P} A_{n,p}^2$ — норма Фробениуса; $A_{n,p}$ — (n, p)-й элемент матрицы $\mathbf{A}$, являющийся аналогом квадратичной ℓ_2-нормы для матриц (*см. разд. П3.5.3*).

Мы также можем выразить наше множество желаемых аппроксимаций, которые мотивируют анализ главных компонент (приведенный в уравнении (8.15)) и которые минимизация этой стоимостной функции заставляет соблюдаться как можно точнее:

$$\mathbf{CW} \approx \mathbf{X} . \tag{8.30}$$

Это множество желаемых аппроксимаций часто именуется *матричной факторизацией*, т. к. мы хотим *факторизировать* матрицу $\mathbf{X}$ в произведение двух матриц $\mathbf{C}$ и $\mathbf{W}$. Она представляет собой матричный аналог факторизации одного числа на два "более простых", например, как $5 \times 2 = 10$. Отсюда, другими словами, задача анализа главных компонент может быть интерпретирована как базовый образчик *задачи матричной факторизации*.

Анализ главных компонент (PCA) не единственный метод обучения без учителя, который мы можем переделать таким образом. Рекомендательные системы (*см. разд. 8.4*) приводят к стоимостной функции, которая близко отражает результаты PCA — и так же близко отражает его компактную форму, приведенную выше. Здесь единственная разница заключается в том, что многие элементы матрицы данных неизвестны, поэтому факторизация ограничивается только теми значениями матрицы $\mathbf{X}$, которые *известны*. Обозначив через Ω индексное множество извест-

ных значений матрицы $\mathbf{X}$, матричную факторизацию, участвующую в рекомендательных системах, можно записать в форме:

$$\{\mathbf{CW} \approx \mathbf{X}\}|_{\Omega}, \tag{8.31}$$

где запись $\{\mathbf{V}\}|_{\Omega}$ используется для обозначения того, что нас интересуют только элементы входной матрицы $\mathbf{V}$ в индексном множестве Ω, что является незначительным отклонением PCA-факторизации из уравнения (8.30). Соответствующая стоимость наименьших квадратов схожим образом является незначительным видоизменением в компактной стоимости наименьших квадратов, лежащей в основе PCA, из уравнения (8.29) и задается как

$$g(\mathbf{W}, \mathbf{C}) = \frac{1}{P}\|\{\mathbf{CW} - \mathbf{X}\}|_{\Omega}\|_{l}^{2}. \tag{8.32}$$

Обратите внимание, что это просто матричная форма стоимостной функции рекомендательных систем, приведенной ранее в уравнении (8.22).

Наконец, мы также легко видим, что метод K средних (*см. разд. 8.5*) попадает в ту же категорию и тоже может быть интерпретирован как *задача матричной факторизации*. Мы можем сделать это, сначала переинтерпретировав наше первоначальное желание с помощью кластеризации методом K средних, что точки в k-м кластере должны лежать близко к своему центроиду, что может быть записано математически как

$$\mathbf{c}_k \approx \mathbf{x}_p \text{ для всех } p \in S_k \quad k = 1, ..., K, \tag{8.33}$$

где $\mathbf{c}_k$ — центроид k-го кластера; S_k — множество индексов подмножества тех P точек данных, которые принадлежат этому кластеру. Эти желаемые отношения можно записать в более удобной в матричной нотации для центроидов, обозначая через $\mathbf{e}_k$ k-й стандартный базисный вектор (т. е. $K \times 1$ -вектор с 1 в k-м слоте и нулями в остальных), схожим образом как

$$\mathbf{Ce}_k \approx \mathbf{x}_p \text{ для всех } p \in S_k \quad k = 1, ..., K. \tag{8.34}$$

Затем, введя матричную нотацию для весов (здесь ограниченную стандартными базисными векторами) и данных, мы также можем записать вышеприведенные отношения как

$$\mathbf{CW} \approx \mathbf{X} \text{ для всех } p \in S_k \quad k = 1, ..., K, \tag{8.35}$$

где

$$\mathbf{w}_p \in \{\mathbf{e}_k\}_{k=1}^{K}, \quad p = 1, ..., P. \tag{8.36}$$

На рис. 8.14 наглядно проиллюстрировано приведенное выше компактно написанное желаемое отношение K средних для малого прототипического набора данных. Обратите внимание, что местоположение единственного ненулевого элемента в каждом столбце матрицы отнесений $\mathbf{W}$ определяет принадлежность к кластеру соответствующей точки данных в $\mathbf{X}$. Поэтому, другими словами, K средних тоже является задачей матричной факторизации (с очень конкретным набором ограничений на матрицу $\mathbf{W}$).

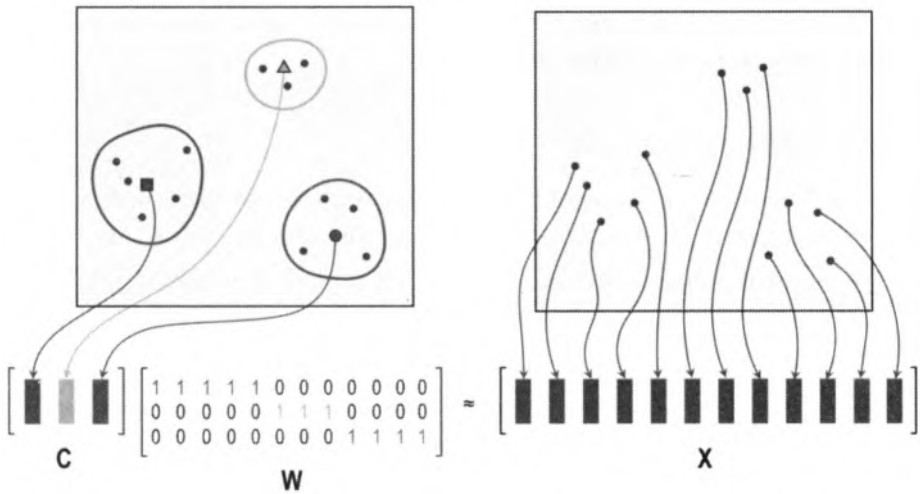


Рис. 8.14. Отношения кластеризации методом K средних, описанные в компактной матричной форме. Центроиды кластеров в $\mathbf{C}$ лежат близко к соответствующим им кластерным точкам в $\mathbf{X}$. p-й столбец матрицы отнесений $\mathbf{W}$ содержит стандартный базисный вектор, соответствующий кластерному центроиду точки данных

Сформулировав желаемый исход, когда параметры установлены оптимально, соответствующую стоимостную функцию для K средних можно также записать компактно в виде

$$g(\mathbf{W}, \mathbf{C}) = \frac{1}{P}\|\mathbf{CW} - \mathbf{X}\|_F^2 \tag{8.37}$$

с учетом ограничения, что $\mathbf{w}_p \in \{\mathbf{e}_k\}_{k=1}^K$ для $p = 1, ..., P$. Другими словами, мы можем интерпретировать алгоритм K средних, описанный в *разд. 8.5*, как способ решения задачи ограниченной оптимизации:

$$\begin{aligned} &\underset{\mathbf{C},\mathbf{W}}{\text{минимизировать}} \quad \|\mathbf{CW} - \mathbf{X}\|_F^2 \\ &\text{с учетом, что} \qquad \mathbf{w}_p \in \{\mathbf{e}_k\}_{k=1}^K, \quad p = 0, ..., P. \end{aligned} \tag{8.38}$$

Можно легко показать, что алгоритм K средних, выведенный нами в предыдущем разделе, также представляет собой множество обновлений, получаемых в результате применения метода блочно-координатного спуска для решения вышеупомянутой задачи оптимизации K средних. Этот взгляд на метод K средних особенно полезен, поскольку в естественной процедуре выведения K средних, показанной в предыдущем разделе, этот метод является несколько эвристическим алгоритмом (т. е. он не привязан к минимизации стоимостной функции, как и любой другой метод, который мы обсуждаем). Одним из практических следствий этого является то, что раньше у нас не было каркаса, в рамках которого можно было бы судить о том, как прогрессирует одиночный прогон алгоритма. Теперь он у нас есть. И мы уже знаем, что можем рассматривать алгоритм K средних точно так же, как и любой другой метод оптимизации, который мы обсуждаем, — как способ минимизирования кон-

кретной стоимостной функции, — и можем использовать стоимостную функцию, чтобы понять, как алгоритм функционирует.

8.6.2. Дальнейшие вариации

В уравнении (8.38) мы увидели, как метод *K* средних может быть переделан в задачу ограниченной матричной факторизации, где каждый столбец $\mathbf{w}_p$ матрицы отнесений $\mathbf{W}$ ограничен стандартным базисным вектором. Это делается для того, чтобы гарантировать отнесение каждой точки данных $\mathbf{x}_p$ к одному (и только одному) кластерному центроиду $\mathbf{c}_k$. Существует много других популярных задач матричной факторизации, которые — с точки зрения моделирования — просто задействуют иные ограничения, чем те, которые даны для *K* средних. Например, естественное обобщение метода *K* средних, именуемое *разреженным кодированием* [30], представляет собой кластерный алгоритм, отличающийся от *K* средних только тем, что он допускает отнесение точек данных к *многочисленным кластерам*. Разреженное кодирование представляет собой задачу неотрицательной матричной факторизации, часто записываемую как

$$\begin{aligned} &\underset{\mathbf{C},\mathbf{W}}{\text{минимизировать}} \ \|\mathbf{CW}-\mathbf{X}\|_F^2 \\ &\text{с учетом, что} \qquad \|\mathbf{w}_p\|_0 \le S, \quad p=1,\ \ldots,\ P, \end{aligned} \tag{8.39}$$

где ограничения *K* средних заменяются ограничениями формы $\|\mathbf{w}_p\|_0 \le S$, что делает возможным одновременное отнесение каждой точки $\mathbf{x}_p$ не более чем к *S* кластерам. Напомним, что $\|\mathbf{w}_p\|_0$ указывает на число ненулевых элементов в векторе $\mathbf{w}_p$ (*см. разд. П3.5.1*).

Помимо разреженности поиск неотрицательной факторизации входной матрицы $\mathbf{X}$ является еще одним ограничением, иногда накладываемым на матрицы $\mathbf{C}$ и $\mathbf{W}$, давая так называемую *задачу неотрицательной матричной факторизации*:

$$\begin{aligned} &\underset{\mathbf{C},\mathbf{W}}{\text{минимизировать}} \ \|\mathbf{CW}-\mathbf{X}\|_F^2 \\ &\text{с учетом, что} \qquad \mathbf{C},\ \mathbf{W} \ge 0. \end{aligned} \tag{8.40}$$

Неотрицательная матричная факторизация (см. [31]) используется преимущественно в ситуациях, когда данные естественным образом являются неотрицательными (например, представление текстовых данных в форме мешка слов, представление интенсивности пикселов в фотоснимковых данных и т. д.) там, где наличие отрицательных значений препятствует интерпретируемости усваиваемых решений.

В табл. 8.1 приведен список часто встречающихся задач матричной факторизации с учетом возможных ограничений на $\mathbf{C}$ и $\mathbf{W}$.

Таблица 8.1. Часто встречающиеся задачи матричной факторизации $\mathbf{CW} \approx \mathbf{X}$ с учетом возможных ограничений на $\mathbf{C}$ и $\mathbf{W}$

Задача матричной факторизации	Ограничения на C и W
Анализ главных компонент	C является ортонормальной
Рекомендательные системы	Нет ограничений на **C** или **W**; **X** известна лишь частично
Кластеризация методом *K* средних	Каждый столбец матрицы **W** является стандартным базисным вектором
Усвоение разряженного словаря	Каждый столбец матрицы **W** является разреженным
Неотрицательная матричная факторизация	Как **C**, так и **W** являются неотрицательными

8.7. Заключение

В этой главе мы познакомились с обучением без учителя через подробное описание ряда популярных концепций и моделей. Такие задачи отличаются от тех, что мы видели в предыдущих трех главах, тем, что они содержат только вход (и никакого выхода).

Мы начали в *разд. 8.2* с рассмотрения понятий охватывающих множеств, а также понятий кодировок/декодировок. Затем в *разд. 8.3* мы рассмотрели тему усвоения оптимальных линейных кодировок посредством так называемого *линейного автокодировщика*. В этом разделе мы также изучили специальное решение с использованием *анализа главных компонент* (PCA) задачи линейного автокодировщика, которое может быть удобно вычислено в терминах собственных векторов матрицы ковариаций данных. Далее в *разд. 8.4* мы рассмотрели популярное видоизменение в понятии усвоения охватывающих множеств — так называемые рекомендательные системы. Затем в *разд. 8.5* была введена кластеризация методом *K средних*. Наконец, в *разд. 8.6* мы ввели каркас *матричной факторизации*, который компактно демонстрирует сходство между анализом главных компонент, рекомендательными системами, кластеризацией методом *K* средних и более продвинутыми моделями обучения без учителя.

8.8. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:

github.com/jermwatt/machine_learning_refined.

8.1. Стандартный базис

Простым примером ортонормального охватывающего множества является множество *N стандартных базисных векторов*. *N*-й элемент стандартного базиса пред-

ставляет собой вектор, который полностью состоит из нулей, за исключением 1 в его n-м элементе:

$$n\text{-й элемент стандартного базиса } \mathbf{c}_n = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{8.41}$$

Такой вектор мы назвали *вектором в кодировке с одним активным состоянием* в *разд. 7.5*. Упростить формулу для оптимального весового вектора/кодировки из уравнения (8.11) при использовании стандартного базиса.

8.2. Кодировка данных

Повторить эксперимент, описанный в примере 8.1, воспроизводя иллюстрации, показанные на рис. 8.2.

8.3. Ортогональные матрицы и собственные значения

Показать, что $N \times K$-матрица $\mathbf{C}$ ортогональна, *если и только если* все ненулевые собственные значения $\mathbf{C}\mathbf{C}^T$ равны $+1$.

8.4. Невыпуклость линейного автокодировщика

При $K = 1$ построить контурный график на диапазоне $[-5; 5] \times [-5; 5]$ линейного автокодировщика из уравнения (8.17) над набором данных, показанным на рис. 8.5, *слева*. Сколько глобальных минимумов, судя по всему, имеет этот контурный график? С учетом концепции линейного автоавтокодировщика и результата, описанного в примере 8.3, описать оптимальный охватывающий вектор (векторы), представленный этими минимумами, и объяснить, как он (они) соотносится с тем, что показано на рис. 8.5, *слева*.

8.5. Минимизирование линейного автокодировщика над игрушечным набором данных

Повторить эксперимент, описанный в примере 8.3, воспроизводя иллюстрации на рис. 8.5. Имплементировать свой код так, чтобы можно было легко вычислить градиент линейного автокодировщика, используя библиотеку autograd.

8.6. Получение базиса PCA

Повторить эксперимент, описанный в примере 8.4, воспроизводя иллюстрации на рис. 8.7. В качестве основы для своей работы вы можете использовать имплементацию, приведенную в *разд. 8.3.4*.

8.7. Предупреждающий пример

Повторить эксперимент, описанный в примере 8.5, воспроизводя иллюстрации на рис. 8.8.

8.8. Выполнение алгоритма *K* средних

Имплементировать детализированный алгоритм K средних и применить его для надлежащего кластеризирования набора данных, показанного на рис. 8.13, *слева*, используя $K = 3$ кластерных центроида. Визуализировать свои результаты, построив график набора данных и окрасив каждый кластер уникальным цветом.

8.9. Построение графика крутого склона

Повторить эксперимент, описанный в примере 8.7, воспроизводя иллюстрацию, показанную на рис. 8.13, *справа*.

8.10. Попеременная минимизации

Хотя стоимостная функция наименьших квадратов в основе PCA из уравнения (8.29) не является выпуклой, она является *двояковыпуклой* в том смысле, что она выпукла в каждой матрице параметров $\mathbf{C}$ и $\mathbf{W}$ *независимо*. Эта реализация приводит к естественному расширению метода координатного спуска, описанного в *разд. 3.2.2*, именуемому *попеременной минимизацией*. Такие методы оптимизации широко используются для решения общих задач матричной факторизации (например, перечисленных в табл. 8.1).

В этом подходе мы минимизируем стоимость наименьших квадратов в основе PCA последовательно, минимизируя стоимость (до полного завершения) над одной матрицей за раз, т. е. $\mathbf{C}$ или $\mathbf{W}$ независимо (причем другая матрица остается фиксированной). Этот процесс — фиксации одной из двух матриц и минимизации до полного завершения над другой — повторяется до тех пор, пока состояние обеих матриц существенно не изменится или пока не будет достигнуто максимальное число итераций.

Попеременное минимизирование $\mathbf{C}$ и $\mathbf{W}$ до полного завершения сводится к последовательному решению двух систем уравнений первого порядка, генерируемых проверкой условия уравнения первого порядка (11.4) соответственно в $\mathbf{C}$ и $\mathbf{W}$. Записать эти две системы уравнений.

8.9. Обобщения

8.9.1. Все минимумы автокодировщика являются ортогональными матрицами

Для того чтобы показать, что все минимумы линейного автокодировщика в уравнении (8.17) являются ортогональными матрицами, мы сначала подставляем разложение (*см. разд. П3.4*) $\mathbf{CC}^T$ собственного значения как $\mathbf{CC}^T = \mathbf{VDV}^T$, где $\mathbf{V}$ — $N \times H$-матрица ортогональных собственных векторов; $\mathbf{D}$ — диагональная $N \times N$-матрица не более K *неотрицательных* собственных значений вдоль верхних K элемен-

тов ее диагонали (поскольку $\mathbf{CC}^T$ — *матрица внешних произведений*, см. упражнение 4.2), для матрицы $\mathbf{CC}^T$ в p-м слагаемом линейного автокодировщика (напомним, мы исходим из того, что наши данные *среднецентрированы*):

$$\left\|\mathbf{CC}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2 = \left\|\mathbf{VDV}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2 = \mathbf{x}_p^T\mathbf{VDDV}^T\mathbf{x}_p - 2\mathbf{x}_p^T\mathbf{VDV}^T\mathbf{x}_p + \mathbf{x}_p^T\mathbf{x}_p. \quad (8.42)$$

Введя $\mathbf{I}_{N\times N} = \mathbf{VV}^T$ посередине внутреннего произведения $\mathbf{x}_p^T\mathbf{x}_p = \mathbf{x}_p^T\mathbf{VV}^T\mathbf{x}_p$, обозначив $\mathbf{q}_p = \mathbf{V}^T\mathbf{x}_p$ и $\mathbf{A}^2 = \mathbf{AA}$ для любой квадратной матрицы $\mathbf{A}$, мы можем переписать приведенную выше правую часть:

$$\mathbf{q}_p^T\mathbf{DDq}_p - 2\mathbf{q}_p^T\mathbf{Dq}_p + \mathbf{q}_p^T\mathbf{q}_p = \mathbf{q}_p^T\left(\mathbf{D}^2 - 2\mathbf{D} + \mathbf{I}_{N\times N}\right)\mathbf{q}_p = \mathbf{q}_p^T\left(\mathbf{D} - \mathbf{I}_{N\times N}\right)^2\mathbf{q}_p, \quad (8.43)$$

где последнее равенство следует из полного квадрата.

Выполнив это для каждого слагаемого, линейный автокодировщик из уравнения (8.17) можно записать эквивалентным образом:

$$g(\mathbf{C}) = \sum_{p=1}^{P}\mathbf{q}_p^T\left(\mathbf{D} - \mathbf{I}_{N\times N}\right)^2\mathbf{q}_p. \quad (8.44)$$

Поскольку элементы матрицы $\mathbf{D}$ неотрицательные, легко увидеть, что эта величина минимизируется для $\mathbf{C}$ так, что $g(\mathbf{C}) = (N-K)\sum_{p=1}^{P}\mathbf{x}_p^T\mathbf{x}_p$, т. е. где $\mathbf{D}_{K\times K}$ (верхняя $K\times K$-порция матрицы $\mathbf{D}$) является именно тождеством. Другими словами, линейный автокодировщик минимизируется над матрицами $\mathbf{C}$, где все ненулевые собственные значения в $\mathbf{CC}^T$ равны $+1$, и единственные такие матрицы, которые обладают этим свойством, являются *ортогональными* (см. упражнение 8.3).

8.9.2. Формальная процедура выведения главных компонент

Для того чтобы начать процедуру выведения классического решения задачи *анализа главных компонент* для линейного автокодировщика в уравнении (8.17), от нас требуется лишь рассмотреть одно слагаемое стоимости при допущении, что $\mathbf{C}$ является ортонормальной. Расширив p-е слагаемое, мы имеем

$$\left\|\mathbf{CC}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2 = \mathbf{x}_p^T\mathbf{CC}^T\mathbf{CC}^T\mathbf{x}_p - 2\mathbf{x}_p^T\mathbf{CC}^T\mathbf{x}_p + \mathbf{x}_p^T\mathbf{x}_p, \quad (8.45)$$

а затем, используя наше допущение о том, что $\mathbf{C}^T\mathbf{C} = \mathbf{I}_{K\times K}$, мы видим, что оно может быть переписано эквивалентным образом:

$$-\mathbf{x}_p^T\mathbf{CC}^T\mathbf{x}_p + \mathbf{x}_p^T\mathbf{x}_p = -\left\|\mathbf{C}^T\mathbf{x}_p\right\|_2^2 + \left\|\mathbf{x}_p\right\|_2^2. \quad (8.46)$$

Поскольку наша цель состоит в том, чтобы минимизировать суммирование членов, принимающих вышеприведенную форму, и точка данных $\mathbf{x}_p$ фиксирована и не включает переменную $\mathbf{C}$, относительно которой мы минимизируем, минимизирование исходного слагаемого слева эквивалентно минимизированию *только первого*

члена — $\left\|\mathbf{C}^T\mathbf{x}_p\right\|_2^2$ в правой части равенства. Просуммировав эти члены, p-й из которых можно записать разложенным над каждым отдельным базисным элементом, который мы стремимся усвоить как

$$-\left\|\mathbf{C}^T\mathbf{x}_p\right\|_2^2 = -\sum_{n=1}^{K}\left(\mathbf{c}_n^T\mathbf{x}_p\right)^2, \tag{8.47}$$

мы получаем следующую подлежащую минимизированию эквивалентную стоимостную функцию для нашего идеального ортонормального базиса:

$$g(\mathbf{C}) = -\frac{1}{P}\sum_{p=1}^{P}\sum_{n=1}^{K}\left(\mathbf{c}_n^T\mathbf{x}_p\right)^2. \tag{8.48}$$

Рассматривая эту уменьшенную форму нашей линейной стоимостной функции автокодировщика, мы видим, что она *полностью раскладывается над базисными векторами* $\mathbf{c}_n$, т. е. нет таких членов, когда $\mathbf{c}_i$ и $\mathbf{c}_j$ взаимодействуют при $i \neq j$. Из этого вытекает, что мы можем оптимизировать наш ортонормальный базис *по одному элементу за раз*. Изменив порядок приведенных выше слагаемых, мы можем изолировать каждый отдельный базисный элемент над всем набором данных, записав его эквивалентным образом как

$$g(\mathbf{C}) = -\frac{1}{P}\sum_{p=1}^{P}\sum_{n=1}^{K}\left(\mathbf{c}_n^T\mathbf{x}_p\right)^2. \tag{8.49}$$

Теперь мы можем думать о минимизировании нашей стоимостной функции по одному базисному элементу за раз. Начиная с $\mathbf{c}_1$, мы сначала изолируем только те приведенные выше релевантные члены, которые состоят из $-\frac{1}{P}\sum_{p=1}^{P}\left(\mathbf{c}_1^T\mathbf{x}_p\right)^2$. Поскольку перед знаком суммы стоит знак "минус", это значит, что нужно *максимизировать* его отрицательное значение, которое мы обозначим через

$$h(\mathbf{c}_1) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathbf{c}_1^T\mathbf{x}_p\right)^2. \tag{8.50}$$

Поскольку наш базис ограничен быть ортонормальным, базисный элемент $\mathbf{c}_1$ в частности ограничен иметь единичную длину. Статистически уравнение (8.50) измеряет *дисперсию набора данных в направлении, определенном* $\mathbf{c}_1$. Заметьте: эта величина является именно дисперсией, поскольку предполагается, что наши данные были среднецентрированы. Поскольку мы стремимся максимизировать эту величину, мы также можем сформулировать нашу оптимизацию исключительно в терминах выборочных статистик: мы стремимся восстановить форму базисного вектора $\mathbf{c}_1$, который указывает на максимальное направление дисперсии в наборе данных.

Для того чтобы определить максимальное значение приведенной выше функции или направления максимальной дисперсии данных, мы можем переписать приведенную выше формулу путем комплектации (среднецентрированных) точек дан-

ных $\mathbf{x}_p$ по столбцам — сформировав $N \times P$-*матрицу данных* $\mathbf{X}$ (как показано в уравнении (8.18)), получив эквивалентную формулу:

$$h(\mathbf{c}_1) = \frac{1}{P}\mathbf{c}_1^T \mathbf{X}\mathbf{X}^P \mathbf{c}_1 = \mathbf{c}_1^T \left(\frac{1}{P}\mathbf{X}\mathbf{X}^P\right)\mathbf{c}_1. \tag{8.51}$$

Записанное в этой форме вышеизложенное принимает форму так называемого отношения Рэлея, максимум которого выражается алгебраически в замкнутой форме на основе разложения собственного значения/собственного вектора матрицы $\mathbf{X}\mathbf{X}^T$ (в среднем члене) или схожим образом матрицы $\frac{1}{P}\mathbf{X}\mathbf{X}^T$ (в члене справа). Поскольку матрица $\frac{1}{P}\mathbf{X}\mathbf{X}^T$ может статистически интерпретироваться как *матрица ковариаций* данных, чаще всего используется конкретное алгебраическое расположение справа.

Итак, обозначим через $\mathbf{v}_1$ и d_1 соответствующий собственный вектор и наибольшее собственное значение матрицы $\frac{1}{P}\mathbf{X}\mathbf{X}^T$. Тогда максимум из вышеперечисленного возникает при $\mathbf{c}_1 = \mathbf{v}_1$, где $h(\mathbf{v}_1) = d_1$ — что также является дисперсией в этом направлении. На жаргоне машинного обучения $\mathbf{v}_1$ называется *первой главной компонентой* данных.

Имея на руках наш первый базисный вектор, мы можем перейти к определению второго элемента нашего идеального ортонормального охватывающего множества. Изолировав соответствующие члены из вышеприведенного и следуя тому же мыслительному процессу, который мы прошли выше, получаем функцию, по виду уже знакомую нам:

$$h(\mathbf{c}_2) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathbf{c}_2^T \mathbf{x}_p\right)^2, \tag{8.52}$$

которую мы стремимся максимизировать, чтобы восстановить наш второй базисный вектор. Эта формула имеет ту же самую статистическую интерпретацию, что и аналогичная версия первого базисного вектора выше, — здесь она снова вычисляет дисперсию данных в направлении $\mathbf{c}_2$. Поскольку наша цель здесь состоит в максимизировании (учитывая, что $\mathbf{c}_1$ уже был решен и что $\mathbf{c}_1^T \mathbf{c}_2 = 0$ вследствие нашего ортонормального допущения), статистическая интерпретация здесь заключается в том, что мы стремимся отыскать *второе* по величине ортогональное направление дисперсии в данных.

Эта формула также может быть записана в компактной векторно-матричной форме как $h(\mathbf{c}_2) = \mathbf{c}_2^T \left(\frac{1}{P}\mathbf{X}\mathbf{X}^P\right)\mathbf{c}_2$, и ее максимум (учитывая, что из нашего ограничения на ортонормальный базис следует, что мы должны иметь $\mathbf{c}_1^T \mathbf{c}_2 = 0$) снова выражается в замкнутой форме в терминах разложения собственного значения/собственного

вектора ковариационной матрицы $\frac{1}{P}\mathbf{X}\mathbf{X}^T$. Здесь тот же анализ, ведущий к надлежащей форме $\mathbf{c}_1$, показывает, что максимум из вышеперечисленного возникает, когда $\mathbf{c}_2 = \mathbf{v}_2$ собственному вектору матрицы $\frac{1}{P}\mathbf{X}\mathbf{X}^T$, связанному с ее вторым по величине собственным значением d_2, и дисперсия в этом направлении тогда $h(\mathbf{v}_2) = d_2$. Этот идеальный базисный элемент/направление называется *второй главной компонентой данных.*

В более общем случае, следуя тому же анализу для n-го члена нашего идеального ортонормального базиса, мы стремимся максимизировать уже знакомую нам формулу:

$$h(\mathbf{c}_n) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathbf{c}_n^T\mathbf{x}_p\right)^2. \qquad (8.53)$$

Как и в первых двух случаях выше, стремление максимизировать эту величину можно интерпретировать как стремление раскрыть n-е ортонормальное направление дисперсии в данных. И следуя тем же аргументам, записав вышеизложенное компактнее, как $h(\mathbf{c}_n) = \mathbf{c}_n^T\left(\frac{1}{P}\mathbf{X}\mathbf{X}^P\right)\mathbf{c}_n$ и т. д., мы можем показать, что оно принимает вид $\mathbf{c}_n = \mathbf{v}_n$, где $\mathbf{v}_n$ — n-й собственный вектор матрицы $\frac{1}{P}\mathbf{X}\mathbf{X}^T$, связанный с ее n-м наибольшим собственным значением d_n, и здесь выборочная дисперсия выражается в терминах этого собственного значения $h(\mathbf{c}_n) = d_n$. Этот усвоенный элемент/направление называется *n-й главной компонентой данных.*

9 Инженерия и отбор признаков

9.1. Введение

В этой главе мы обсудим *принципы инженерии* и *отбора признаков*.

Методы инженерии, или конструирования, или выработки, признаков состоят из массива приемов, которые применяются к данным *до* того, как они будут использованы в моделях с учителем или без. Некоторые из этих инструментов, например *методы шкалирования признаков* (*см. разд. 9.3–9.5*), надлежаще *нормализуют* входные данные и обеспечивают единообразный конвейер предобработки для обучения, что значительно повышает эффективность многих методов локальной оптимизации.

Другая ветвь инженерии признаков фокусируется на разработке преобразований данных, которые извлекают полезную информацию из сырых входных данных. Например, в случае двухклассовой классификации эти инструменты ориентированы на извлечение критически значимых элементов набора данных, которые обеспечивают рассмотрение экземпляров в пределах одного класса как "похожих", а экземпляров из разных классов — как "непохожих". Конструирование таких инструментов часто требует глубоких знаний в предметной области и богатого опыта работы с конкретными видами данных. Однако мы увидим, что одна простая концепция — преобразование *гистограммных* признаков — является общепринятым инструментом инженерии признаков, используемым для широкого ряда типов данных, включая категориальные, текстовые, фотоснимковые и звуковые данные. Мы дадим высокоуровневый обзор этого популярного подхода к инженерии признаков в *разд. 9.2*.

Люди часто являются неотъемлемым компонентом парадигмы машинного обучения, и очень важно, чтобы человек мог интерпретировать и/или извлекать информацию из модели машинного обучения. *Результативность* модели является распространенной и относительно простой метрикой, используемой людьми для интерпретации: обеспечивает ли модель хорошие[1] предсказательные результаты? Иногда очень полезно понять, *какие* входные признаки были наиболее адекватными для достижения высокой результативности, поскольку это помогает нам уточнить наше понимание природы рассматриваемой задачи. Это делается с помощью

[1] Здесь "хорошо" может означать, например, что ученик достигает согласованного эталонного значения точности, ошибки и т. д.

так называемого *отбора признаков*. В *разд. 9.6* и *9.7* мы обсудим два популярных способа выполнения отбора признаков: *бустинг* и *регуляризацию*.

Отбор признаков можно понимать, как контролируемый прием редукции размерности, который редуцирует суммарное число признаков, участвующих в регрессии или классификации, делая результирующую модель более пригодной для интерпретации человеком. Абстрактная иллюстрация этой концепции показана на рис. 9.1, *слева* и визуально сопоставлена с результатами анализа главных компонент (PCA) и кластеризации (двух методов обучения без учителя, представленных в предыдущей главе). В отличие от отбора признаков, когда PCA (*см. разд. 8.3*) применяется для уменьшения размерности набора данных, он делает это путем усвоения нового (меньшего) множества признаков, над которым набор данных может быть справедливо представлен. Точно так же любой метод кластеризации (например, *K* средних, подробно описанный в *разд. 8.5*) усваивает новые представления, чтобы уменьшить гигантское число точек в наборе данных.

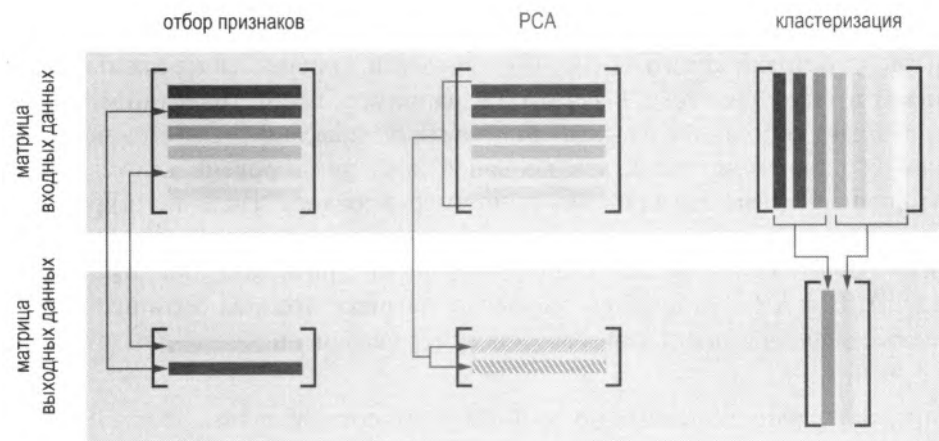


Рис. 9.1. Сравнение отбора признаков, анализа главных компонент (PCA) и кластеризации в качестве схем уменьшения размерности на произвольной матрице данных, подобной тем, которые мы обсуждали в предыдущих главах для обучения с учителем и без, строки которых содержат признаки, а столбцы — отдельные точки данных. Первые два метода уменьшают размерность пространства признаков, или, другими словами, число строк в матрице данных. Однако эти два метода работают по-разному: в то время как отбор признаков буквально отбирает строки из исходной матрицы для сохранения, PCA использует геометрию пространства признаков для создания новой матрицы данных, основанной на версии с более низкой размерностью признаков. Метод *K* средних, с другой стороны, уменьшает размерность данных/число точек данных или, что эквивалентно, число столбцов в матрице входных данных. Он делает это, отыскивая малое число новых усредненных представителей или центроидов входных данных, формируя новую матрицу данных, меньшее число столбцов которой (которых нет в исходной матрице данных) являются именно этими центроидами

9.2. Гистограммные признаки

Гистограмма — это чрезвычайно простой, но полезный способ подытоживания и представления содержимого массива. В этом разделе мы увидим, что эта довольно простая концепция лежит в основе инженерии признаков для часто встречающихся типов входных данных, включая категориальные, текстовые, фотоснимковые и звуковые типы данных. Хотя каждый из этих типов данных существенно отличается по своей природе, мы увидим, как понятие гистограммно-обусловленного признака имеет смысл в каждом контексте. Наше обсуждение ориентировано на то, чтобы дать читателю высокоуровневое, интуитивное понимание того, как работают общепринятые методы обработки признаков на основе гистограмм. Заинтересованному читателю рекомендуется ознакомиться со специализированными статьями (ссылки на которые имеются во всем этом разделе) по каждому типу данных для дальнейшего изучения.

9.2.1. Гистограммные признаки для категориальных данных

Каждая парадигма машинного обучения требует, чтобы данные, с которыми мы имеем дело, состояли строго из *числовых* значений. Однако сырые данные не всегда поступают в такой упаковке. Рассмотрим, например, гипотетический медицинский набор данных, состоящий из жизненно важных показаний о состоянии здоровья нескольких пациентов, таких как кровяное давление, уровень глюкозы крови и группа крови. Первые два признака (например, кровяное давление и уровень глюкозы крови), естественно, являются числовыми и, следовательно, готовы к использованию. Группа крови является *категориальным* признаком, принимающим значения O, A, B и AB. Такие категориальные признаки должны быть преобразованы в числовые значения перед тем, как они будут введены в любой алгоритм машинного обучения.

Первый, интуитивно понятный подход для этого состоит в том, чтобы представить каждую категорию отдельным действительным числом, например, назначив 0 группе крови O, 1 — группе A, 2 — группе B и 3 — группе AB (рис. 9.2, *слева сверху*). Здесь важно то, как мы назначаем числа каждой категории. В данном конкретном случае, назначая 1 группе A, 2 группе B и 3 группе AB, и поскольку число 3 ближе к числу 2, чем к числу 1, мы непреднамеренно впрыснули в наш набор данных допущение о том, что человек с группой крови AB более "похож" на человека с группой крови B, чем человек с группой крови A. Можно было бы возразить, что уместнее было бы переставить местами числа, используемые для представления категорий B и AB, так, чтобы AB теперь располагалась между (и, следовательно, была равноудалена от) A и B (рис. 9.2, *справа сверху*). Однако при таком переназначении группа крови O теперь интерпретируется как максимально отличающаяся от группы крови B — это своего рода допущение, которое *может* и *не быть* истинным в реальности.

Суть здесь в том, что любое множество чисел всегда имеет естественный порядок, и, используя такие значения, мы неизбежно добавляем в наши данные допущение

о *сходстве* или *несходстве* существующих категорий. В большинстве случаев мы хотим избежать такого рода допущений, которые коренным образом меняют геометрию задачи, в особенности, когда нам не хватает интуиции или знаний, необходимых для определения сходства между разными категориями.

Более подходящий способ кодирования категориальных признаков, позволяющий избежать этой проблемы, заключается в использовании *гистограммы*, корзины которой представляют все категории, присутствующие в интересующем нас категориальном признаке. При этом в примере с группой крови индивидуум с группой крови О больше не представлен одним числом, а представлен гистограммой с четырьмя корзинами, содержащими все нулевые значения, кроме корзины, ассоциированной с группой крови О, которая имеет значение 1. Лица с другими группами крови представлены схожим образом, как показано визуально на рис. 9.2, *снизу*. Таким образом, все представления группы крови (каждый четырехмерный вектор с одной 1 и тремя 0) геометрически равноудалены друг от друга. Этот метод кодирования категориальных признаков иногда называется кодированием с *одним активным состоянием* (one-hot encoding, *см. разд. 6.7.1*)[2].

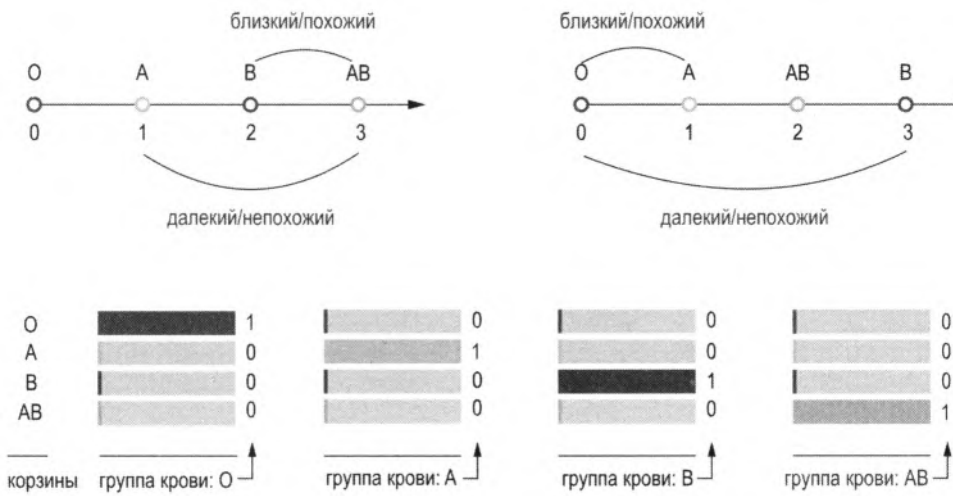


Рис. 9.2. Группа крови, конвертированная в числовые признаки (*сверху*) и гистограммные признаки (*снизу*)

9.2.2. Гистограммные признаки для текстовых данных

Многие популярные методы машинного обучения, включая сентиментный анализ, обнаружение спама, классификацию или кластеризацию документов, основаны на текстовых данных, например новостных статьях в Интернете, электронных пись-

[2] Словосочетание "кодирование с одним активным состоянием" (one-hot encoding) пришло из терминологии цифровых интегральных микросхем, где оно описывает конфигурацию микросхемы, в которой допускается, чтобы только один бит был положительным (активным). — *Прим. пер.*

мах, сообщениях в социальных медиа и т. д. Однако в случае текстовых данных прежде всего входное значение (т. е. сам документ) требует значительного объема предобработки и преобразования до своего введения в любой алгоритм машинного обучения. Очень простое, но широко используемое признаковое преобразование документа для задач машинного обучения называется гистограммой *мешка слов* (Bag of Words, BoW), или признаковым вектором. Здесь мы познакомимся с гистограммой мешка слов и обсудим ее сильные и слабые стороны, а также часто встречающиеся расширения.

Признаковый вектор документа с использованием мешка слов (BoW) — это простое гистограммное количество разных слов, содержащихся в нем по отношению к единому корпусу или коллекции документов, за вычетом тех неразличимых слов, которые не характеризуют документ (в контексте приложения).

В качестве иллюстрации этой идеи давайте построим представление в виде мешка слов для следующего ниже корпуса из двух простых текстовых документов, каждый из которых состоит из одного предложения[3].

```
1. dogs are the best
2. cats are the worst
```
(9.1)

Для того чтобы составить представление этих документов в виде мешка слов, мы начинаем с их разбора и создания репрезентативных векторов (гистограмм) $\mathbf{x}_1$ и $\mathbf{x}_2$, которые содержат число раз, когда каждое слово появляется в каждом документе. Векторы мешков слов (или BoW-векторы) для обоих этих документов изображены на рис. 9.3.

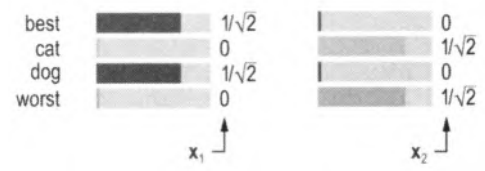


Рис. 9.3. Гистограммные признаки мешка слов для двух демонстрационных документов, показанных в уравнении (9.1)

Обратите внимание, что при создании гистограмм мешка слов неинформативные слова, такие как `are` и `the`, обычно именуемые *стоп-словами*, в окончательное представление не включаются. Далее заметьте, что мы учитываем слова `dog` и `cat` в их единственном числе, а не множественном, которое фигурировало в первоначальных документах. Этот шаг предобработки обычно называется *выделением основ слов*, или *стеммингом*, когда родственные слова с общей основой или корнем сводятся к их общему лингвистическому корню (а затем им представляются). Например, слова learn, learning, learned и learner в окончательном признаковом векторе мешка слов представляются и учитываются как learn. Кроме того, каждый BoW-вектор *нормализуется*, имея в итоге единичную длину.

[3] Собаки лучше; коты хуже. — *Прим. пер.*

Учитывая, что BoW-векторы содержат только неотрицательные записи и все имеют единичную длину, внутреннее произведение между двумя BoW-векторами $\mathbf{x}_1$ и $\mathbf{x}_2$ всегда варьируется между 0 и 1, т. е. $0 \leq \mathbf{x}_1^{\prime}\mathbf{x}_2 \leq 1$. Это внутреннее произведение или значение *корреляции* может быть использовано в качестве грубой геометрической меры сходства между двумя BoW-векторами. Например, когда два документа состоят из совершенно разных слов, корреляция равна строго нулю, их BoW-векторы перпендикулярны друг другу и два документа можно считать максимально разными. Это относится и к BoW-векторам, показанным на рис. 9.3. С другой стороны, чем выше корреляция между двумя BoW-векторами, тем более похожими считаются соответствующие им документы. Например, BoW-вектор документа `I love dogs` будет иметь довольно большую положительную корреляцию с вектором `I love cats`.

Поскольку BoW-вектор является таким вот простым представлением документа, которое полностью игнорирует порядок слов, пунктуацию и т. д., оно может обеспечить только *грубую сводку* содержимого документа. Например, два документа `dogs are better than cats` и `cats are better than dogs` будут считаться *фактически одинаковым документом* при использовании представления в виде мешка слов, даже если они имеют полностью противоположный смысл. Тем не менее грубая сводка, обеспечиваемая мешком слов, может быть достаточно разной для многих приложений. Кроме того, в то время как могут быть задействованы более сложные представления документов (вычленение порядка слов, частей речи и т. д.), они часто бывают громоздкими (см., например, [32]).

Пример 9.1. Сентиментный анализ

Определение агрегированных чувств крупной клиентской базы с использованием текстового содержимого, такого как продуктовые обзоры, твиты и комментарии в социальных медиа, обычно называют *сентиментным анализом*, или *анализом настроений*. Классификационные модели часто используются для проведения сентиментного анализа для идентификации потребительских данных с положительными либо отрицательными чувствами.

Например, на рис. 9.4, *сверху* показаны представления в виде BoW-векторов двух кратких обзоров спорного комедийного фильма: один с положительным мнением, а другой с отрицательным. BoW-векторы на этом рисунке повернуты в сторону так, чтобы горизонтальная ось содержала общие слова между двумя предложениями (после удаления стоп-слов и выделения основ). Полярно противоположное восприятие этих двух отзывов прекрасно отображено в их представлениях в виде мешка слов, которые, как хорошо видно, действительно перпендикулярны (т. е. имеют нулевую корреляцию). В общем случае, два документа с противоположными настроениями не всегда и не обязательно должны быть перпендикулярными для эффективной работы сентиментного анализа, даже если в идеале мы ожидаем, что они будут иметь малые корреляции, как показано на рис. 9.4, *снизу*.

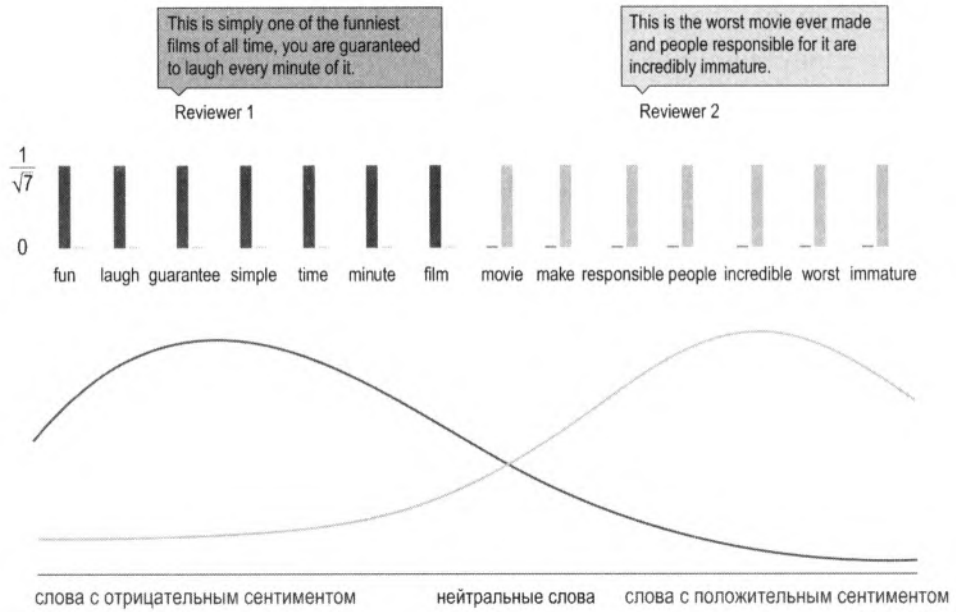


Рис. 9.4. Иллюстрация примера 9.1. Представление двух отрывков из отзывов о фильме в виде мешка слов, причем слова (после удаления стоп-слов и выделения основ) распределены между двумя отзывами и перечислены вдоль горизонтальной оси (*сверху*). Совершенно разное мнение каждого рецензента очень хорошо отражается на гистограммах мешка слов, которые имеют нулевую корреляцию. В общем случае ожидается, что кривая гистограммы типичного документа с положительным настроением будет иметь малую корреляцию с гистограммой типичного документа с отрицательным настроением[4] (*снизу*)

Пример 9.2. Обнаружение спама

Во многих детекторах спама (см. пример 1.8) признаковые BoW-векторы формируются по отношению к определенному списку спамных слов (или фраз), таких как `free`, `guarantee`, `bargain`, `act now`, `all natural` и т. д., которые часто встречаются в спамных письмах. Кроме того, такие признаки, как частота некоторых символов (например, `!` и `*`), добавляются в признак мешка слов, как и другие спам-ориентированные признаки, включая общее число заглавных букв в электронном письме и длину самой длинной сплошной последовательности заглавных букв, поскольку эти признаки могут дальше различать эти два класса.

На рис. 9.5 показаны результаты классифицирования на наборе спамных писем (впервые представленного в примере 6.10), состоящего из мешка слов (BoW), частоты символов и других спамно-концентрированных признаков. Задействуя стоимость двухклассового мягкого максимума (*см. разд. 7.3*) для усвоения разделителя, рисунок демонстрирует число неправильных классифицирований для каждого шага

[4] Перевод отзывов. Рецензент 1: это один из самых смешных фильмов всех времен и народов, вы гарантированно будет смеяться на протяжении всего фильма. Рецензент 2: это худший фильм из когда-либо снятых, и люди, ответственные за его создание, — невероятно незрелые. — *Прим. пер.*

выполнения метода Ньютона (*см. разд. 4.3*). Выражаясь конкретнее, эти результаты классифицирования показаны для того же самого набора данных с использованием лишь признаков мешка слов (синим цветом, график 1), признаков мешка слов и частоты символов (оранжевым цветом, график 2) и признаков мешка слов, частоты символов и спамно-ориентированных признаков (зеленым цветом, график 3). Неудивительно, что добавление частот символов улучшает классификацию, причем наилучшая результативность возникает, когда также используются спамно-концентрированные признаки.

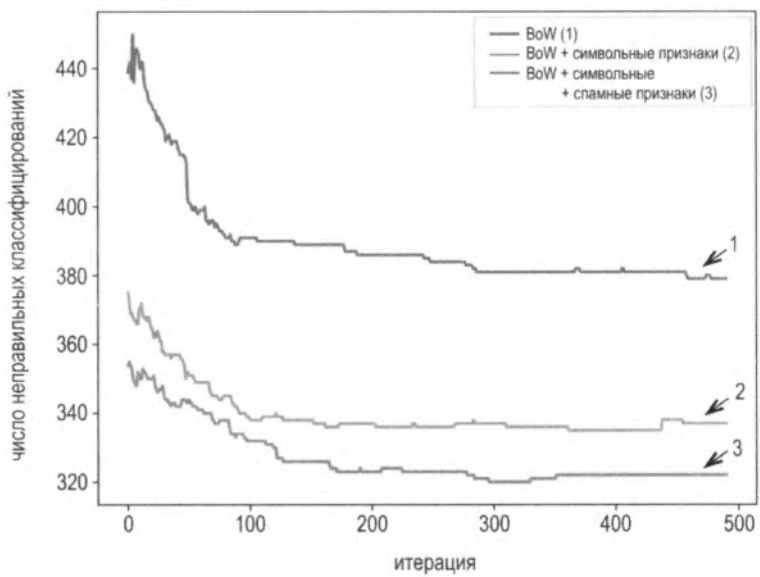


Рис. 9.5. Иллюстрация примера 9.2

9.2.3. Гистограммные признаки для фотоснимковых данных

Для выполнения задач обучения с учителем или без него, таких как распознавание объектов или сжатие изображений, сырые входные данные представляют собой пиксельные значения самого фотоснимка. Пиксельные значения 8-битного полутонового фотоснимка, т. е. снимка в оттенках серого, представляют собой всего одноединственное целое число в интервале от 0 (черный) до 255 (белый), как показано на рис. 9.6. Другими словами, полутоновое изображение — это просто *матрица* целых чисел в диапазоне от 0 до 255. Тогда цветное изображение — это всего лишь множество из трех таких полутоновых матриц, по одной для каждого из трех каналов: красного, синего и зеленого.

Хотя существует возможность использовать в качестве признаков сырые пиксельные значения непосредственно, сами пиксельные значения обычно недостаточно различимы, чтобы быть полезными для задач машинного обучения. Мы иллюстрируем причину, почему это так происходит, используя простой пример на рис. 9.7. Возьмем три изображения фигур, показанных в левом столбце этого рисунка.

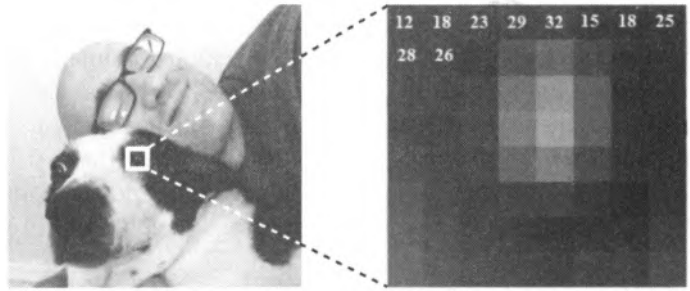


Рис. 9.6. 8-битный полутоновый фотоснимок состоит из пикселов, каждый из которых принимает значение от 0 (черный) до 255 (белый). Для того чтобы визуализировать отдельные пикселы, справа развернут небольшой блок размером 8×8 исходного фотоснимка

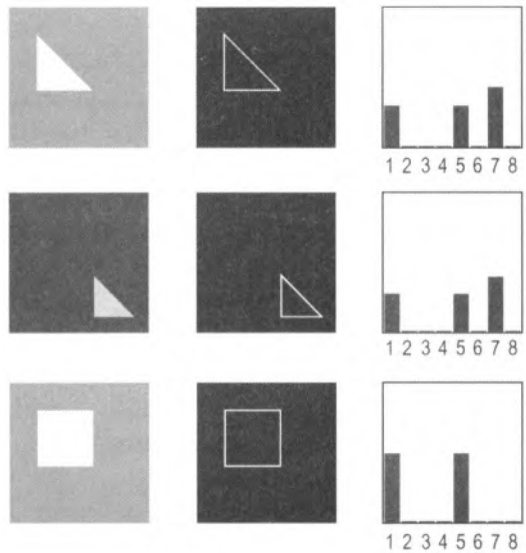


Рис. 9.7. Три изображения простых фигур. Хотя треугольники на двух верхних слева изображениях визуально похожи, это сходство не отражается при сравнении их сырых пиксельных значений (*левый столбец*). Версии исходных изображений после обнаружения ребер, здесь использующие восемь реберных ориентаций, сохраняют различительное структурное содержимое, значительно редуцируя объем информации в каждом изображении (*в центре*). Взяв нормализованные гистограммы реберного содержимого, мы получаем признаковое представление, которое довольно хорошо отражает сходство двух треугольников, отличая их от квадрата (*правый столбец*)

Первые два — это подобные треугольники, и третья фигура — квадрат. Мы хотели бы, чтобы идеальный набор признаков отражал сходство первых двух изображений, а также их отличие от последнего изображения. Однако из-за разницы в их относительном размере, положении на изображении и контрасте самого изображения (изображение с меньшим треугольником в целом темнее), если бы мы использовали сырые пиксельные значения для сравнения изображений (беря разность

между каждой парой изображений), то обнаружили бы, что квадрат и больший треугольник более похожи, чем сами два треугольника[5]. Это обусловлено тем, что пиксельные значения первого и третьего изображений благодаря их одинаковому контрасту и расположению треугольника/квадрата действительно более похожи, чем значения изображений двух треугольников.

В *центре* рисунка и *справа* проиллюстрирован двухшаговый процесс, который генерирует своего рода преобразование различающего признака, которое мы ищем. Сначала мы смещаем перспективу от самих пикселов в сторону к реберному содержимому в каждом пикселе. Беря ребра вместо пиксельных значений, мы значительно уменьшаем объем информации, с которой должны иметь дело в изображении, не разрушая идентифицирующие его структуры. В среднем столбце показаны соответствующие изображения после обнаружения ребер, в частности после выделения восьми одинаково отстоящих (под углом) реберных ориентаций, начиная с 0° (горизонтальные ребра) с семью дополнительными ориентациями с шагом 22,5°, включая 45° (захватывая диагональные ребра треугольников) и 90° (вертикальные ребра). Ясно, что ребра сохраняют характеристики, отличительные для каждого исходного изображения, в то же время значительно редуцируя объем общей информации в каждом случае.

Затем мы строим нормализованную гистограмму реберного содержимого каждого изображения (как показано для примеров на рис. 9.7, *справа*). То есть мы составляем вектор, содержащий общие суммы каждой найденной на изображении реберной ориентации (вертикальная ось гистограммы), и нормализуем результирующий вектор до единичной длины. Это полностью аналогично признаковому представлению в виде мешка слов, описанному ранее для текстовых данных, причем подсчет числа реберных ориентаций аналогичен подсчету числа "слов" в случае текстовых данных. Здесь мы также имеем нормализованную гистограмму, которая грубо представляет изображение, игнорируя расположение и упорядочение его информации. Однако, как показано на рис. 9.7, *справа* (в отличие от сырых пиксельных значений), эти гистограммные признаковые векторы захватывают характерную информацию о каждом изображении, причем два верхних изображения треугольника имеют очень похожие гистограммы и значительно отличаются от третьего изображения квадрата.

Обобщения этой простой концепции реберной гистограммы широко используются в качестве признакового преобразования для распознавания визуальных объектов, когда цель состоит в том, чтобы выделить интересующие объекты (например, лица в приложении по распознаванию лиц или пешеходов в самоуправляемом автомобиле) на примере изображения, или когда необходимо отличить друг от друга разные объекты на многочисленных изображениях (скажем, рукописные цифры, как в приведенном далее примере, или даже отличить кошек от собак, как описано в *разд. 1.2*). Это обусловлено тем фактом, что реберное содержимое тяготеет к тому, чтобы со-

---

[5] Это означает, что если мы обозначим через $\mathbf{X}_i$ i-е изображение, то обнаружим, что $\|\mathbf{X}_1 - \mathbf{X}_3\|_F < \|\mathbf{X}_1 - \mathbf{X}_2\|_F$.

хранять структуру более сложных изображений — как показано на рис. 9.8 — при резком редуцировании объема информации в изображении [33, 34]. Большинство пикселов в этом изображении не принадлежат никаким ребрам, но только с помощью ребер мы по-прежнему можем сказать, что именно содержит изображение.

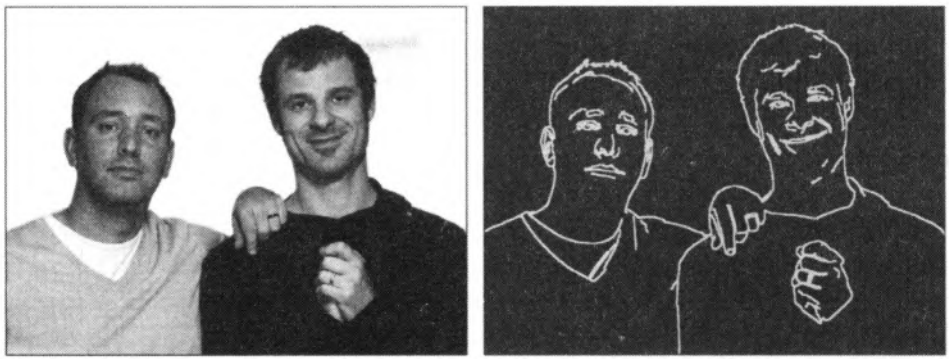


Рис. 9.8. Естественный фотоснимок в данном случае двух создателей/сценаристов телешоу "Южный парк" (воспроизводится с разрешения Джейсона Марка) (*слева*). Версия этого фотоснимка после обнаружения ребер, где ярко-желтые пикселы указывают на большое реберное содержимое, по-прежнему очень хорошо описывает сцену (в том смысле, что мы по-прежнему можем сказать, что на фотоснимке есть два человека), используя только часть информации, содержащейся в исходном фотоснимке (*справа*). Обратите внимание, что ребра были окрашены в желтый цвет только для целей визуализации

Однако для таких сложных изображений важным становится сохранение локальной информации (признаков изображения на меньших участках). Отсюда естественным способом расширения реберного гистограммного признака является его вычисление не по всему изображению, а путем разбиения изображения на относительно малые участки и вычисления реберной гистограммы каждого участка, а затем объединения результатов. На рис. 9.9 показана диаграмма общей вариации этого метода, часто используемой на практике, когда соседние гистограммы нормализуются совместно в более крупных блоках. (Дополнительные подробности см., например, в [5, 35–38].)

Интересно, что эта конструкция реберно-обусловленного гистограммного признака имитирует то, как многие животные, судя по всему, обрабатывают визуальную информацию. Из визуальных исследований, выполненных в основном на лягушках, кошках и приматах, где испытуемому животному показывают визуальные стимулы, в то время как электрические импульсы регистрируются в небольшой области мозга испытуемого животного, где обрабатывается визуальная информация, нейробиологи определили, что участвующие в процессе индивидуальные нейроны работают в общих чертах, идентифицируя ребра [39, 40]. Каждый нейрон, следовательно, действует как малый "реберный детектор", локализуя в изображении местоположение ребер конкретной ориентации и толщины, как показано на рис. 9.10. Считается, что за счет совмещения и обработки этих ребер именно эти изображения, получающиеся после обнаружения ребер, "видят" люди и другие млекопитающие.

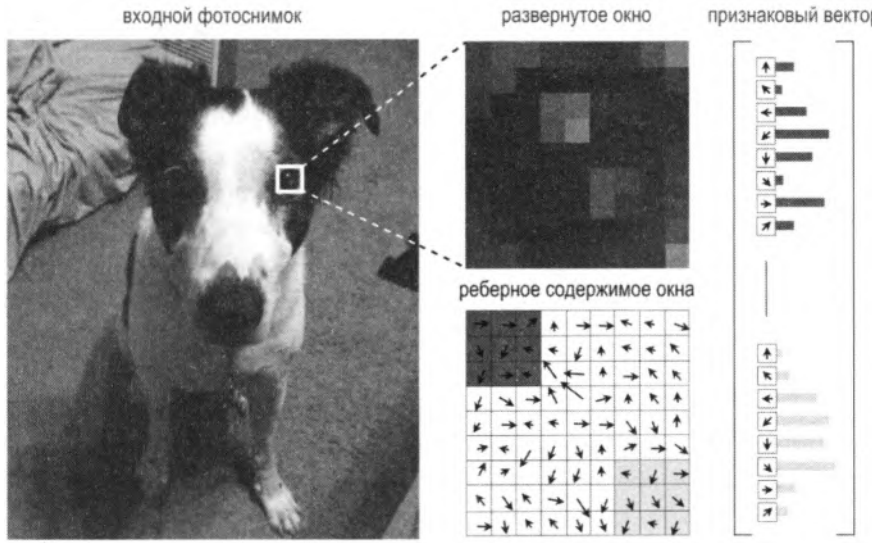


Рис. 9.9. Наглядное представление своего рода обобщенного преобразования реберных гистограммных признаков, широко используемого для обнаружения объектов. Входное изображение разбивается на малые (здесь 9×9) блоки, и реберная гистограмма вычисляется на каждом из меньших непересекающихся (здесь 3×3) участков, составляющих блок. Результирующие гистограммы затем конкатенируются и нормализуются совместно, продуцируя признаковый вектор для всего блока. Конкатенирование таких блочных признаков путем сканирования блочного окна над всем изображением дает окончательный признаковый вектор

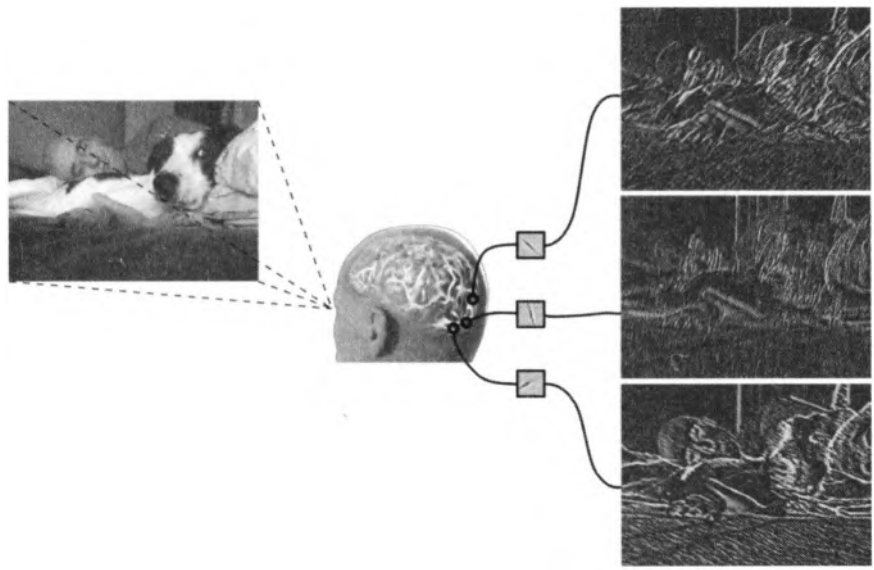


Рис. 9.10. Визуальная информация обрабатывается в той области мозга, где каждый нейрон обнаруживает в наблюдаемой сцене ребра конкретной ориентации и ширины. Считается, что то, что мы (и другие млекопитающие) "видим", является обработанной интерполяцией этих изображений с обнаруженными ребрами

Пример 9.3. Распознавание рукописных цифр

В этом примере мы рассмотрим задачу распознавания рукописных цифр (введенную в примере 1.10) и сравним тренировочную эффективность мини-пакетного градиентного спуска (20 шагов/эпох с интенсивностью усвоения $\alpha = 10^{-2}$ и размером пакета 200 применительно к стоимости многоклассового мягкого максимума) с использованием $P = 50\,000$ сырых (пиксельных) точек данных из набора данных MNIST, т. е. набора данных для распознавания рукописных цифр (введенного в примере 7.10), с эффективностью точно такой же установочной комбинации, примененной к реберным гистограммным признакам, извлеченным из этих же самых точек данных.

Эффективность этой установочной комбинации по сравнению с исходными данными как с точки зрения стоимостной функции, так и с точки зрения истории неправильного классифицирования, получающейся в результате оптимизационного прогона, показана в виде черных кривых соответственно на рис. 9.11, *слева* и *справа*. Результаты того же самого прогона над версией набора данных после излечения реберных признаков показаны сиреневыми кривыми на том же рисунке. Здесь мы видим огромный разрыв в результативности, в особенности в графике истории неправильного классифицирования на рисунке справа, где разница в результативности составляет около 4000 неправильных классифицирований (в пользу прогона над реберно-обусловленными признаками).

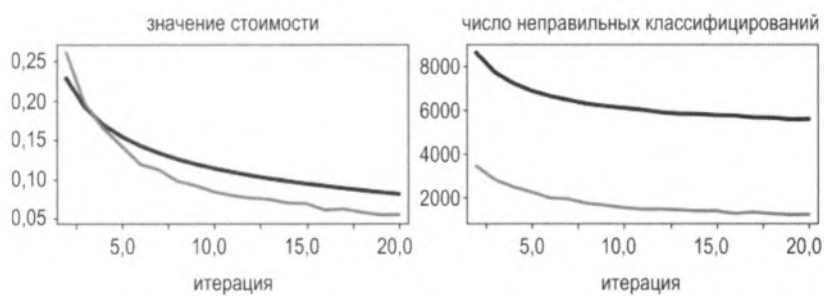


Рис. 9.11. Иллюстрация примера 9.3

9.2.4. Гистограммные признаки для аудиоданных

Как и фотоснимки, аудиосигналы в сырой форме недостаточно различимы, чтобы их можно было использовать для задач классификации на основе звука (например, распознавания речи), и снова мы можем применить надлежаще сконструированные гистограммные признаки. В случае аудиосигнала именно гистограмма его частот, иначе именуемая *спектром*, обеспечивает робастную сводку его содержимого. Как показано на рис. 9.12, спектр звукового сигнала подсчитывает (в виде гистограммы) силу каждого уровня его частоты или осцилляции. Это делается путем декомпозиции речевого сигнала на основе синусоидальных волн постоянно увеличивающейся частоты, причем веса на каждой синусоиде представляют силу этой частоты

в исходном сигнале. Каждый уровень осцилляции аналогичен направлению ребра в случае фотоснимка или отдельному слову в случае текстового признака мешка слов.

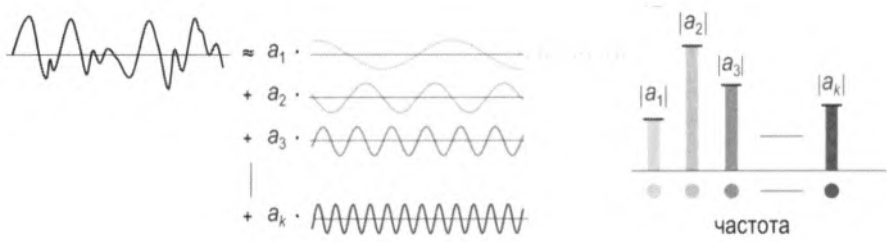


Рис. 9.12. Наглядное представление звукового сигнала и его представление в виде частотной гистограммы или спектра (левая панель). Фигуральный звуковой сигнал можно разложить на линейную комбинацию простых синусоид с варьирующимися частотами (или осцилляциями) (*слева*). Гистограмма частоты будет содержать силу каждой синусоиды в представлении звукового сигнала (*справа*)

Как и в случае со фотоснимковыми данными на рис. 9.9, вычисление частотных гистограмм над перекрывающимися окнами звукового сигнала, как показано на рис. 9.13, создает признаковый вектор, который также сохраняет важную локальную информацию и является общепринятым признаковым преобразованием, используемым для распознавания речи, именуемым *спектрограммой* [41, 42]. Дальнейшая обработка оконных гистограмм (например, чтобы подчеркнуть частоты звука, лучше всего распознаваемые человеческим ухом) также широко выполняется в практических имплементациях такого рода признакового преобразования.

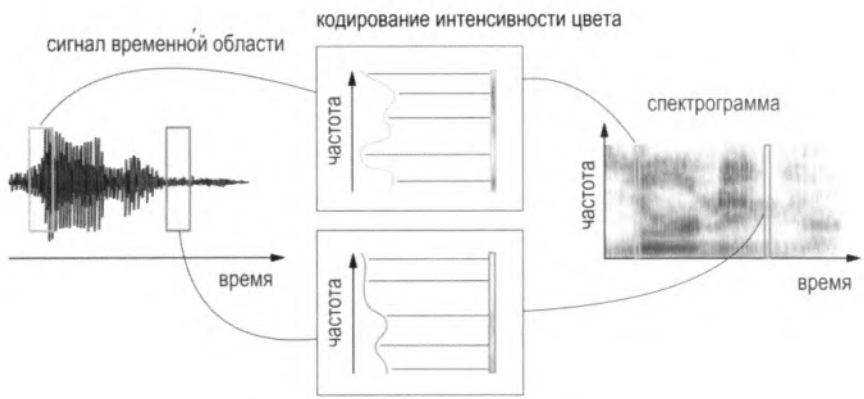


Рис. 9.13. Наглядное представление гистограммных признаков для аудиоданных. Исходный речевой сигнал (*слева*) разбивается на малые (перекрывающиеся) окна, частотные гистограммы которых вычисляются и укладываются вертикально для получения так называемой *спектрограммы* (*справа*)

9.3. Шкалирование признаков посредством стандартной нормализации

В этом разделе мы опишем популярный метод нормализации входных данных в машинном обучении, именуемый *шкалированием признаков* посредством *стандартной нормализации*. Такого рода схема инженерии признаков обеспечивает для процесса автоматического усвоения ряд выгод, включая существенное повышение скорости процесса усвоения при использовании с алгоритмами локальной оптимизации и, в частности, с методами первого порядка. Этот метод инженерии признаков также можно рассматривать как *оптимизационный трюк*, который существенно улучшает нашу способность минимизировать практически каждую модель машинного обучения.

9.3.1. Стандартная нормализация

Стандартная нормализация входных признаков набора данных представляет собой очень простую двухшаговую процедуру, сперва состоящую из *среднецентрирования* и последующего *шкалирования* каждого входного признака на величину, обратную его стандартному отклонению. С точки зрения алгебры, мы нормализуем вдоль n-го входного признака нашего набора данных, заменяя $x_{p,n}$ (n-ю координату входной точки $\mathbf{x}_p$) на

$$\frac{x_{p,n} - \mu_n}{\sigma_n}, \tag{9.2}$$

где μ_n и σ_n — среднее значение и стандартное отклонение, вычисленные вдоль n-й размерности данных, определяемые соответственно как

$$\begin{aligned} \mu_n &= \frac{1}{P}\sum_{p=1}^{P} x_{p,n}; \\ \sigma_n &= \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(x_{p,n} - \mu_n\right)^2}. \end{aligned} \tag{9.3}$$

Эта (полностью обратимая) процедура выполняется для каждой входной размерности $n = 1, ..., N$. Заметьте, что если $\sigma_n = 0$ для некоторого n, то стандартная нормализация в уравнении (9.2) не определена, т. к. она включает деление на ноль. Однако в этом случае соответствующий входной признак является *избыточным*, поскольку из этого вытекает, что n-й признак является одинаковым постоянным значением во всей совокупности данных. Как мы обсудим далее в следующем разделе, такой признак должен быть удален из набора данных в самом начале, поскольку в любой модели машинного обучения ничего нельзя усвоить из его присутствия.

Вообще говоря, стандартная нормализация изменяет контур стоимостных функций машинного обучения, делая их контуры более "округлыми". Эта идея проиллюстрирована на рис. 9.14, где в верхнем ряду показан двумерный ($N = 2$) набор данных

(*слева сверху*), а также его стандартно нормализованная версия (*справа сверху*). В пространстве входных данных стандартная нормализация создает центрированную и более компактно ограниченную версию исходных данных. В то же время, как показано на рис. 9.14, *снизу*, обобщенная стоимостная функция, связанная со стандартно нормализованной версией данных, имеет контуры, которые являются гораздо более округлыми, чем контуры, связанные с исходными ненормализованными данными.

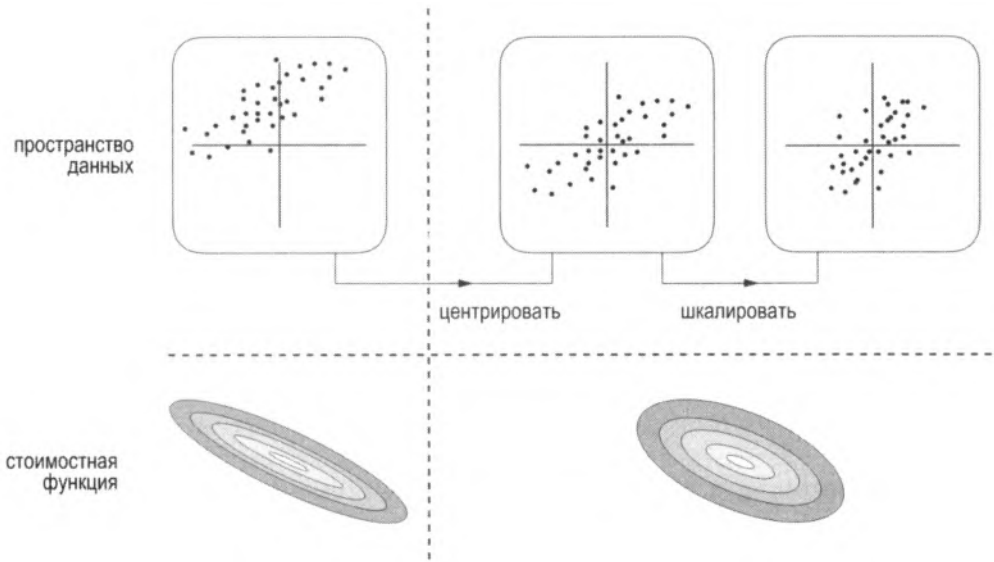


Рис. 9.14. Иллюстрация стандартной нормализации. Входное пространство обобщенного набора данных (*слева*), а также его среднецентрированная (*в центре*) и шкалированная версия (*справа*). Как проиллюстрировано в нижнем ряду, где показана стоимостная функция, соответствующая этим данным, стандартная нормализация приводит к стоимостной функции с менее эллиптическими и более округлыми контурами по сравнению с исходной стоимостной функцией

Придание контурам стоимостной функции машинного обучения более округлой формы помогает ускорить схождение схем локальной оптимизации, в особенности методов первого порядка, таких как градиентный спуск (однако приемы шкалирования признаков также помогают лучше приспособить набор данных для использования с методами второго порядка, помогая потенциально избежать проблем с числовой нестабильностью, как кратко описано в *разд. 4.3.3*). Это обусловлено тем, что, как подробно описано в *разд. 3.6.2*, направление градиентного спуска всегда указывает перпендикулярно контурам стоимостной функции. Из этого следует, что применительно к минимизированию стоимостной функции с эллиптическими контурами, как в примере на рис. 9.15, *сверху*, направление градиентного спуска будет *всегда* указывать в сторону, противоположную от минимизатора стоимостной функции. Эта характеристика естественным образом приводит к тому, что алгоритм градиентного спуска делает *зигзагообразные* шаги вперед и назад.

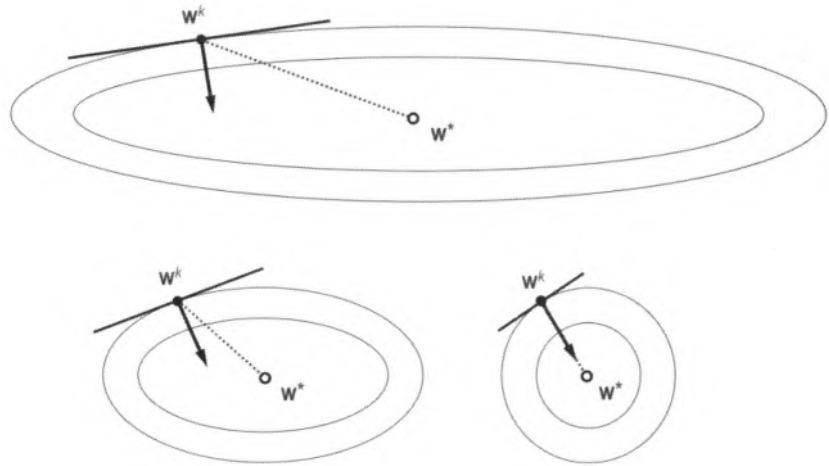


Рис. 9.15. В стандартном нормализировании входных данных мы нередко сглаживаем эллиптические контуры ассоциированной стоимостной функции (*сверху*) в более округлые (*снизу*). Из этого следует, что направление градиентного спуска, которое указывает в сторону, противоположную от минимизатора стоимостной функции, когда ее контуры являются эллиптическими (что приводит к часто встречающейся проблеме зигзагообразного поведения градиентного спуска), указывает больше в сторону минимизатора функции, т. к. ее контуры становятся более округлыми. Это делает каждый шаг градиентного спуска намного эффективнее, обычно позволяя использовать гораздо более крупные значения параметра длины шага α, и значит, для адекватного минимизирования стоимостной функции требуется значительно меньше шагов

При стандартном нормализировании входных данных мы сглаживаем такие эллиптические контуры, преобразуя их в более округлые контуры, как показано на рис. 9.15, *слева снизу* и (в идеале) *справа снизу*. С увеличением числа округлых контуров направление градиентного спуска начинает все больше и больше указывать в направлении минимизатора стоимостной функции, что делает каждый шаг градиентного спуска гораздо эффективнее. Это часто означает, что во время минимизирования стоимостной функции над стандартно нормализованными данными мы можем использовать гораздо более крупный параметр длины шага α.

Пример 9.4. Линейная регрессия со стандартно нормализованными данными

На рис. 9.16, *слева сверху* построен график простого регрессионного набора данных. Окинув взглядом данные, мы увидим, что при надлежащей настройке линейный регрессор будет очень хорошо вписываться в этот набор данных. Поскольку этот пример является низкоразмерным с двумя настраиваемыми параметрами (т. е. смещением и наклоном оптимально подогнанной прямой), мы можем визуализировать связанную с ним стоимостную функцию наименьших квадратов, как показано на рис. 9.16, *в центре сверху*. Обратите внимание, насколько эллиптичны контуры этой стоимостной функции, создавая длинную узкую долину вдоль длинной оси эллипсов. В средней верхней части рисунка также показаны 100 шагов градиентного спуска, инициализированного в точке $\mathbf{w}^0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$, с фиксированным парамет-

ром длины шага $\alpha = 10^{-1}$. Здесь шаги на контурном графике окрашиваются из зеленого в красный цвет, по мере того, как градиентный спуск начинается (зеленый) и заканчивается (красный). Рассматривая эту часть рисунка, мы видим, что даже в конце прогона у нас по-прежнему остается довольно большой путь, чтобы добраться до минимума стоимостной функции. График прямой (синим цветом), связанной с окончательным набором весов, полученным в результате этого прогона, приведен на рис. 9.16, *слева сверху*. Поскольку эти веса лежат довольно далеко от истинного минимизатора стоимостной функции, они провоцируют (относительно) слабую подгонку к данным.

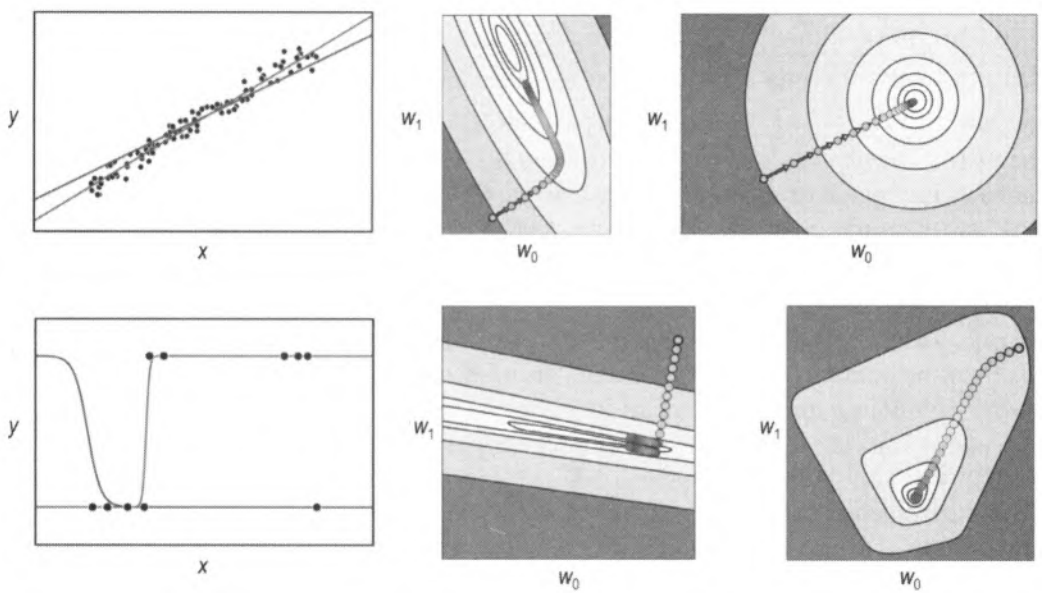


Рис. 9.16. Иллюстрация примеров 9.4 и 9.5, показывающая результат стандартной нормализации после применения к регрессионному (*сверху*) и двухклассовому классификационному (*снизу*) наборам данных

Затем мы стандартно нормализуем данные и визуализируем контурный график ассоциированной стоимости наименьших квадратов на рис. 9.16, *справа сверху*. Как мы видим, контуры этой стоимости наименьших квадратов являются совершенно округлыми, и поэтому градиентный спуск может гораздо быстрее минимизировать эту стоимость. Здесь показан прогон из 20 (вместо 100) шагов градиентного спуска с использованием той же начальной точки и параметра длины шага, что и раньше с ненормализованными данными. Обратите внимание, что поскольку стоимостная функция, связанная со стандартно нормализованной версией данных, намного легче поддается оптимизации, мы достигаем точки, гораздо более близкой к минимимуму, всего за несколько шагов, в результате чего получается линейная модель (показанная красным цветом), которая вписывается на рис. 9.16, *слева сверху* гораздо лучше, чем регрессор, предоставленный первым прогоном (показанный синим цветом).

Пример 9.5. Линейная двухклассовая классификация со стандартно нормализованными данными

На рис. 9.16, *слева снизу* показан простой двухклассовый классификационный набор данных. Как и в предыдущем примере, поскольку для этого набора данных у нас есть только два параметра, настраиваемых при усвоении линейного классификатора, мы можем визуализировать контурный график соответствующей стоимости двухклассового мягкого максимума. График контуров этой стоимостной функции построен на рис. 9.16, *в центре снизу*. Опять-таки, их чрезвычайно длинный и узкий контур говорит о том, что градиентному спуску будет безмерно трудно определить глобальный минимум (расположенный внутри самого малого показанного контура).

Наши догадки подтверждает прогон из 100 шагов градиентного спуска, начинающегося в точке $\mathbf{w}^0 = [20 \quad 30]^T$, с параметром длины шага $\alpha = 1$. Как показано нарис. 9.16, *в центре снизу*, эти шаги (окрашиваемые от зеленого до красного по мере прогрессии градиентного спуска) значительно зигзагообразны. Более того, мы видим, что в конце прогона мы все еще далеки от минимума стоимостной функции, что приводит к очень плохой подгонке к набору данных (показанной синим цветом на рис. 9.16, *снизу слева*).

На рис. 9.16, *справа снизу* представлен результат повторения этого эксперимента с использованием стандартно нормализованного входа. Здесь мы используем ту же самую инициализацию, но только 25 шагов, и (поскольку контуры связанной с ней стоимости гораздо более округлые) большее значение длины шага $\alpha = 10$. Это довольно крупное значение привело бы к тому, что первый прогон градиентного спуска разошелся бы. Тем не менее даже с таким малым числом шагов мы можем найти хорошую аппроксимацию глобального минимума. На рис. 9.16, *слева снизу* построен график соответствующей гиперболической тангенсной модели (красным цветом), которая вписывается в данные гораздо лучше, чем результат первого прогона (показанного синим цветом).

9.3.2. Стандартно нормализованная модель

После того как общая модель, принимающая N-мерный стандартно нормализованный вход, была надлежаще настроена и оптимальные параметры $w_0^\star$, $w_1^\star$, ..., $w_N^\star$ были определены для того, чтобы оценить любую новую точку, мы должны стандартно нормализовать каждый из ее входных признаков, используя ту же самую статистику, которую мы вычисляли на тренировочных данных.

9.4. Вменение значений, отсутствующих в наборе данных

В реальной ситуации по различным причинам в данных могут *отсутствовать значения*, в том числе по причине человеческого фактора при их сборе, трудностей

с хранением, неисправных датчиков и т. д. Вообще говоря, если точка данных при обучении с учителем теряет свое *выходное* значение, мы мало что можем сделать, чтобы ее спасти, и обычно такая поврежденная точка данных на практике отбрасывается. Аналогично, если большое число входных значений точки данных отсутствует, то ее лучше отбросить. Однако точка данных, в которой отсутствует всего несколько входных признаков, может быть восстановлена путем заполнения недостающих входных признаков надлежащими значениями. Этот процесс, часто именуемый *вменением*, или импутированием, является особенно полезным, когда данных не хватает.

9.4.1. Вменение среднего значения

Как обычно, предположим, что у нас есть множество из P входов, каждый из которых является N-мерным, и что множество Ω_n содержит индексы всех точек данных, у которых отсутствует значение n-го входного признака. Другими словами, для всех $j \in \Omega_n$ значение $x_{j,n}$ отсутствует в наших входных данных. Первое, что интуитивно напрашивается в качестве заполняемого значения для всех отсутствующих записей вдоль n-го входного признака, — это простое среднее арифметическое (или ожидаемое значение) набора данных вдоль этой размерности. То есть для всех $j \in \Omega_n$ мы задаем $x_{j,n} = \mu_n$, где

$$\mu_n = \frac{1}{P - |\Omega_n|} \sum_{j \notin \Omega_n} x_{j,n} \tag{9.4}$$

и где $|\Omega_n|$ обозначает число элементов в Ω_n. Это вменение среднего значения часто именуется *среднезначным вменением*. Обратите внимание, что после среднезначного вменения среднее значение всего n-го признака во входе остается неизменным, т. к.

$$\frac{1}{P}\sum_{p=1}^{P} x_{p,n} = \frac{1}{P}\sum_{j \notin \Omega_n} x_{j,n} + \frac{1}{P}\sum_{j \in \Omega_n} x_{j,n} = \frac{1}{P}(P - |\Omega_n|)\mu_n + \frac{1}{P}\sum_{j \in \Omega_n} \mu_n = \mu_n. \tag{9.5}$$

Одно из следствий вменения пропущенных в наборе данных значений с использованием среднего вдоль каждой входной размерности состоит в том, что, когда мы стандартно нормализуем этот набор данных (как подробно описано в предыдущем разделе), все значения, вменяемые с помощью среднего значения, становятся равными строго нулю. Это иллюстрируется простым примером на рис. 9.17. Отсюда следует, что любой параметр или вес в модели, который касается такой записи со вмененным средним, полностью обнуляется численно. Это поведение является желательным, учитывая, что такие значения с самого начала отсутствовали.

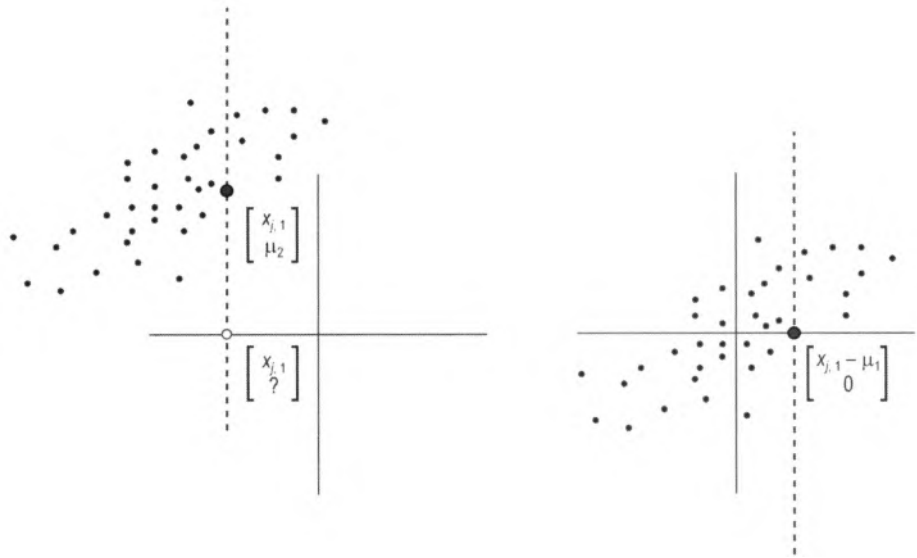


Рис. 9.17. Вход прототипического двумерного набора данных, где у одной точки данных $\mathbf{x}_j$, нарисованной в виде *полой* красной графической точки, отсутствует ее вторая запись. Средневмененная версия этой точки данных затем показывается как *заполненная* красная графическая точка (*слева*). Путем среднецентрирования такого набора данных (которое является первым шагом стандартной нормализации) средневмененный признак точки $\mathbf{x}_j$ становится равным строго нулю (*справа*)

9.5. Шкалирование признаков посредством декоррелирующего преобразования

В *разд. 9.3* мы увидели, как *шкалирование признаков* с помощью *стандартной нормализации* значительно улучшает топологию стоимостных функций машинного обучения, обеспечивая возможность гораздо быстрее минимизировать их с помощью методов первого порядка, таких как градиентный спуск (*см. разд. 3.5*). В этом разделе мы опишем, как можно использовать метод главных компонент (PCA) (*см. разд. 8.3*) для реализации более продвинутой формы нормализации входных данных, общепринято именуемый *декоррелирующим преобразованием (PCA-sphering)* или иногда *отбеливанием (whitening)*.

9.5.1. Декоррелирующее преобразование: картина в целом

Декоррелирующее преобразование развивает идею стандартной нормализации, описанную в *разд. 9.3*, еще на один шаг, используя метод главных компонент (*см. разд. 8.3*) для поворота среднецентрированного набора данных так, чтобы его наибольшие ортогональные направления дисперсии выравнивались в линию с координатными осями, перед тем как шкалировать каждый вход его стандартным отклонением. Эта простая корректировка в типичной ситуации позволяет нам лучше

уплотнить данные и, что более важно, приводит к стоимостной функции, контуры которой являются еще более "округлыми", чем контуры, предоставляемые стандартной нормализацией (на самом деле декоррелирующее преобразование регрессионных данных делает контуры стоимости наименьших квадратов для линейной регрессии *совершенно округлыми* — более подробные сведения см. в упражнении 9.6). Это проиллюстрировано на рис. 9.18, где мы наглядно сравниваем влияние стандартной нормализации и PCA-сферирования на прототипический двумерный входной набор данных, а также то, как каждая схема изменяет топологию ассоциированной стоимостной функции. Как изложено в *разд. 9.3.1*, схемы градиентного спуска работают намного лучше на столько, на сколько более округлыми мы делаем контуры стоимостной функции.

Компромисс, разумеется, заключается в том, что, хотя декоррелирующее преобразование значительно упрощает первопорядковую оптимизацию, как только она

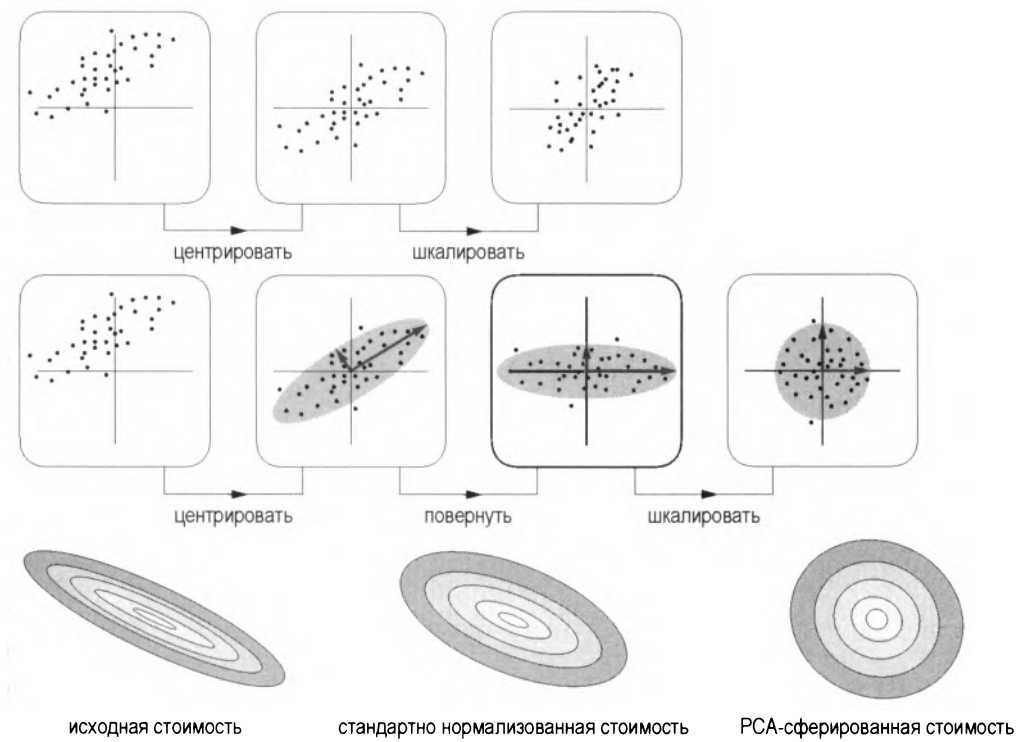


Рис. 9.18. Процедура стандартной нормализации (*сверху*) по сравнению с декоррелирующим преобразованием (*в центре*) на обобщенном наборе входных данных. При декоррелирующем преобразовании мы вставляем один дополнительный шаг в стандартный конвейер нормализации *между* среднецентрированием и шкалированием стандартным отклонением, где поворачиваем данные с помощью PCA. Это не только сжимает пространство, потребляемое данными, больше, чем стандартная нормализация (сравните части рисунка справа сверху и справа в центре), но и значительно облегчает минимизирование любой связанной с ней стоимостной функции за счет лучшего сглаживания ее контуров, делая их более округлыми (*снизу*)

вводится в действие, мы должны платить дополнительную авансовую стоимость за выполнение анализа главных компонент на данных, что делает декоррелирующее преобразование более дорогостоящей вычислительной процедурой, чем процедура стандартной нормализации. Независимо от того, оправданы ли эти дополнительные авансовые издержки, на практике они могут варьироваться от задачи к задаче, как бы часто это ни было. Это особенно верно при задействовании оптимизации первого порядка (см., например, упражнение 9.7), поскольку — как подытожено на рис. 9.15 — чем более округлыми мы делаем контуры стоимостной функции, тем легче методы градиентно-обусловленной оптимизации могут их надлежаще минимизировать.

9.5.2. Декоррелирующее преобразование: технические детали

В более формальном плане мы можем выразить схему стандартной нормализации, применяемую к одной-единственной точке данных $\mathbf{x}_p$, в два шага:

1) центрировать средним значением: для каждого n заменить $x_{p,n} \longleftarrow \left(x_{p,n} - \mu_n\right)$;

2) шкалировать страндартным отклонением: для каждого n заменить $x_{p,n} \longleftarrow \frac{x_{p,n}}{\sigma_n}$,

где μ_n — среднее значение; σ_n — стандартное отклонение:

$$\mu_n = \frac{1}{P}\sum_{n=1}^{P} x_{p,n};$$

$$\sigma_n = \sqrt{\frac{1}{P}\sum\nolimits_{p=1}^{P} x_{p,n}^2}$$

для каждого n. Обратите также внимание на то, что здесь нотация $a \longleftarrow b$ обозначает замену величины a на величину b.

Обозначив через $\mathbf{X}$ $N \times P$-матрицу входных данных, p-й столбец которой содержит входную точку данных $\mathbf{x}_p$, и через $\mathbf{V}$ множество собственных векторов матрицы ковариаций данных $\frac{1}{P}\mathbf{X}\mathbf{X}^T = \mathbf{V}\mathbf{D}\mathbf{V}^T$ (как подробно описано в *разд. 8.3.3*), мы можем записать схему PCA-сферирования, применяемую к той же самой точке данных $\mathbf{x}_p$ в трех сильно связанных шагах, особо отметив, что n-е собственное значение d_n (n-я диагональная запись матрицы $\mathbf{D}$) в точности равно[6] дисперсии σ_n^2 и эквивалентным образом — $\sqrt{d_n} = \sigma_n$:

---

[6] Определение фактора Рэлея (Rayleigh quotient, см., например, упражнение 3.3) n-го собственного значения d_n матрицы ковариаций данных утверждает, что, численно говоря, $d_n = \frac{1}{P}\mathbf{v}_n\mathbf{X}\mathbf{X}^T\mathbf{v}_n$, где $\mathbf{v}_n$ — это n-й соответствующий собственный вектор. Теперь с точки зрения наших PCA-преобра-

1) центрировать средним значением: для каждого n заменить $x_{p,n} \longleftarrow \left(x_{p,n} - \mu_n\right)$;

2) повернуть за счет PCA-сферирования: преобразовать $\mathbf{x}_p \longleftarrow \mathbf{V}^T\mathbf{x}_p$;

3) шкалировать страндартным отклонением: для каждого n заменить $x_{p,n} \longleftarrow \frac{x_{p,n}}{\sigma_n}$.

Обозначив через $\mathbf{D}^{-1/2}$ диагональную матрицу, n-й диагональный элемент которой равен $1/\sqrt{d_n}$, мы можем (после среднецентрирования данных) довольно компактно выразить шаги 2 и 3 описанного выше алгоритма PCA-сферирования как

$$\mathbf{X} \longleftarrow \mathbf{D}^{-1/2}\mathbf{V}^T\mathbf{X}. \tag{9.6}$$

9.5.3. "Отбеленная" модель

После того как общая модель, принимающая N-мерные входные данные после их *"отбеливания"* (whitening), была надлежаще настроена и были определены оптимальные параметры для того, чтобы оценить любую новую точку, мы должны "отбелить" новый входной признак, используя ту же статистику, которую вычисляли на тренировочных данных.

9.6. Отбор признаков посредством бустинга

В *главах 5–7* мы видели, как полностью настроенная линейная модель *обучения с учителем* обычно принимает форму

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = \dot{\mathbf{x}}^T\mathbf{w}^\star = w_0^\star + x_1 w_1^\star + x_2 w_2^\star + \ldots + x_N w_N^\star, \tag{9.7}$$

где веса $w_0^\star, w_1^\star, \ldots, w_N^\star$ оптимально настраиваются посредством минимизирования соответствующей стоимостной функции. Понимание замысловатых соединений входных признаков набора данных с соответствующими выходами естественным образом сводится к человеческому анализу этих $N+1$ настроенных весов. Однако не всегда бывает просто вывести смысл из такой последовательности $N+1$ чисел, что усугубляется тем фактом, что идея *интерпретируемости человеком* быстро становится несостоятельной по мере роста входной размерности N. Для того чтобы облегчить эту трудность, мы можем использовать так называемую *методику отбора* признаков.

В этом разделе мы обсудим один популярный способ выполнения отбора признаков, именуемый *бустингом* или *прямым поэтапным отбором*. Бустинг — это подход к отбору признаков, направленный снизу вверх, при котором мы постепенно надстраиваем нашу модель по одному признаку за раз, тренируя *последовательно*,

---

зованных данных это эквивалентным образом записывается как $d_n = \frac{1}{P}\left\|\mathbf{v}_n^T\mathbf{X}\right\|_2^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_{p,n}\right)^2 = \sigma_n^2$ или, другими словами, является *дисперсией* вдоль n-й оси PCA-преобразованных данных.

по одному весу за раз. Это дает интерпретаторам-людям более простой способ оценивать важность отдельного признака, а также позволяет им легче проникать вглубь конкретного явления.

9.6.1. Отбор признаков на основе бустинга

В настройке модельных весов по одному за раз мы не хотим настраивать их в любом порядке (например, случайным образом), поскольку это не будет способствовать их интерпретации человеком. Вместо этого мы хотим настраивать их, начиная с самого важного (признакового) веса, затем настроить второй самый важный (признаковый) вес, затем третий и т. д. Здесь под "важностью" мы подразумеваем то, как каждый входной признак вносит свой вклад в окончательную модель обучения с учителем, определяемую связанным с ним весом, или, другими словами, каким образом каждый вносит свой вклад в минимизирование соответствующей стоимости (или ассоциированной метрики) настолько, насколько это возможно.

Процесс бустинга начинается с модели, которую мы будем обозначать model_0 и которая состоит только из смещения w_0:

$$\text{model}_0(\mathbf{x}, \mathbf{w}) = w_0 . \tag{9.8}$$

Затем мы настраиваем параметр смещения w_0, минимизируя соответствующую стоимость (в зависимости от того, какую именно задачу мы решаем: регрессионную или классификационную) только *над* этой переменной. Например, если мы выполняем регрессию, задействуя стоимость наименьших квадратов, то минимизируем

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0(\mathbf{x}, \mathbf{w}) - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0 - y_p\right)^2 , \tag{9.9}$$

что дает оптимальное значение для нашего смещения $w_0 \leftarrow w_0^\star$. Вставка этого усвоенного веса в нашу исходную модель в уравнении (9.8) дает

$$\text{model}_0(\mathbf{x}, \mathbf{w}) = w_0^\star . \tag{9.10}$$

Далее, на *первом раунде* бустинга для того, чтобы определить наиболее важный признаковый вес (среди $w_1, w_2, ..., w_N$), мы *апробируем каждый из них*, минимизируя соответствующую стоимость над каждым в отдельности, уже установив смещение оптимальным образом. Например, в случае регрессии методом наименьших квадратов n-я подзадача из этих N подзадач принимает форму

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0(\mathbf{x}, \mathbf{w}) + w_n x_{n,p} - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0^\star + x_{n,p} w_n - y_p\right)^2 . \tag{9.11}$$

Обратите внимание, что поскольку вес смещения уже установлен, мы настраиваем только вес w_n в n-й подзадаче.

Признаковый вес, который производит *наименьшую* стоимость (или *наилучшее* значение метрики в целом) из этих N подзадач, соответствует индивидуальному при-

знаку, который помогает налучшим образом объяснить взаимосвязь между входом и выходом нашего набора данных. И, следовательно, его можно интерпретировать как самый важный признаковый вес, который мы усваиваем. Обозначив этот вес как w_{s_1}, мы затем фиксируем его на оптимально определенном значении $w_{s_1}^{\star}$ (отбрасывая все другие веса, настроенные в каждой из этих подзадач) и соответствующим образом обновляем нашу модель. Наша обновленная модель в конце первого раунда бустинга, которую мы называем model_1, представляет собой сумму нашего оптимального смещения и этого только что определенного оптимального признакового веса:

$$\text{model}_1(\mathbf{x}, \mathbf{w}) = \text{model}_0(\mathbf{x}, \mathbf{w}) + x_{s_1} w_{s_1}^{\star} = w_0^{\star} + x_{s_1} w_{s_1}^{\star}. \tag{9.12}$$

Затем этот процесс бустинга повторяется последовательно. В целом на m-м раунде бустинга m-й наиболее важный признаковый вес определяется по тому же шаблону. В начале m-го раунда (где $m > 1$) мы уже определили оптимальную настройку нашего смещения, а также верхние $m-1$ наиболее важных признаковых весов, и наша модель принимает форму

$$\text{model}_{m-1}(\mathbf{x}, \mathbf{w}) = w_1^{\star} + x_{s_1} w_{s_1}^{\star} + \ldots + x_{s_{m-1}} w_{s_{m-1}}^{\star}. \tag{9.13}$$

Затем мы выдвигаем и решаем $N - m + 1$ подзадач, по одной для каждого признакового веса, который мы еще не выбрали. Например, в случае регрессии методом наименьших квадратов n-я из них принимает форму

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_{m-1}(\mathbf{x}_p, \mathbf{w}) + w_n x_{n,p} - y_p\right)^2, \tag{9.14}$$

где опять же в каждом случае мы настраиваем только индивидуальный вес w_n. Признаковый вес, который производит наименьшую стоимость, соответствует m-му наиболее важному признаку. Обозначив этот вес через w_{s_m}, мы фиксируем его на оптимальном значении $w_{s_m}^{\star}$ и добавляем его вклад в скользящую модель как

$$\text{model}_m(\mathbf{x}, \mathbf{w}) = \text{model}_{m-1}(\mathbf{x}, \mathbf{w}) + x_{s_m} w_{s_m}^{\star}. \tag{9.15}$$

Учитывая, что у нас есть N входных признаков, мы можем продолжать до тех пор, пока $m \le M$ или пока не будет достигнуто некоторое максимальное число итераций. Обратите также внимание на то, что после M раундов бустинга мы сконструировали последовательность моделей $\{\text{model}_m(\mathbf{x}, \mathbf{w})\}_{m=0}^{M}$. Этот метод рекурсивного построения модели путем добавления признаков по одному за раз и настройки веса только добавленного признака (оставляющий все остальные фиксированными на их ранее настроенных значениях) называется *бустингом*.

9.6.2. Отбор правильного числа признаков посредством бустинга

Вспомните, что признаки отбираются в первую очередь для целей *интерпретации человеком*. Следовательно, эталонное значение числа M отбираемых признаков можно выбрать, основываясь на желании разведать набор данных, и процедура

остановится, как только это число раундов будет завершено. Можно также остановить разведывание, когда добавление новых признаков в модель приводит к очень незначительному снижению стоимости, поскольку бóльшая часть корреляции между входами и выходами уже была объяснена. Наконец, M может быть выбрано целиком на основе выборочной статистики набора данных посредством процедуры, именуемой *перекрестной валидацией*, которую мы обсудим в *главе 11*.

Независимо от того, как мы отбираем значение M, важно стандартно нормализовать входные данные (как подробно описано в *разд. 9.3*) перед тем, как мы начнем процедуру бустинга для отбора признаков. В дополнение к преимуществу ускорения оптимизации стандартная нормализация также позволяет нам справедливо сравнивать вклад каждого входного признака, рассматривая значения их настроенных весов.

Пример 9.6. Разведывание предсказателей цен на жилье посредством бустинга

Результат выполнения $M = 5$ раундов бустинга на наборе данных Boston Housing с данными о ценах на жилье в Бостоне (впервые представленном в примере 5.5) с использованием стоимости наименьших квадратов и оптимизатора методом Ньютона визуализируется на рис. 9.19, *сверху*. Этот особый вид истории стоимостной функции показывает каждый весовой/признаковый индекс, добавляемый в модель на каждом раунде бустинга (начиная со смещения, имеющего индекс 0). Как видно на горизонтальной оси, первые два признака с наибольшим вкладом, найденные с помощью бустинга, — это процент местного населения, считающегося "низшим классом", LSTAT (признак 13) и среднее число комнат в жилище (признак 6). Рассматривая гистограмму модельных весов на рис. 9.19, *снизу*, мы видим (и это неудивительно), что вес LSTAT, имеющий отрицательное значение, коррелирует с выходом (т. е. ценой дома) отрицательно, тогда как вес, связанный со средним числом комнат, коррелирует с выходом положительно.

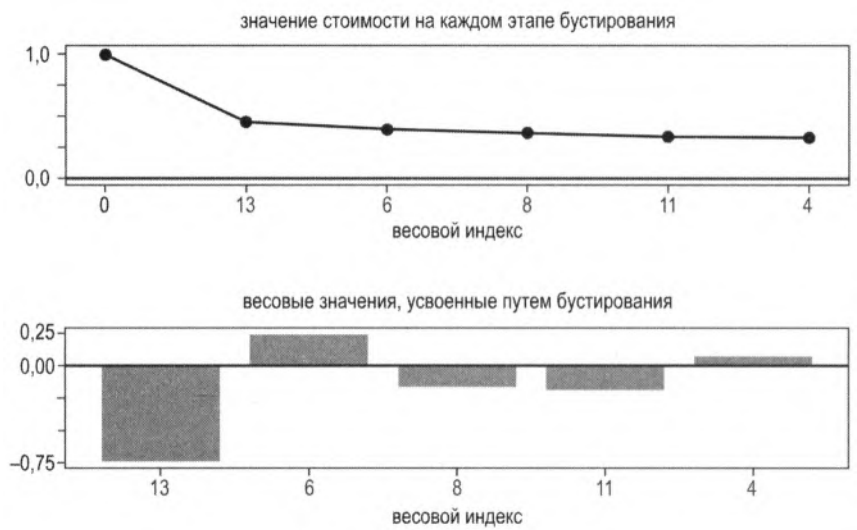


Рис. 9.19. Иллюстрация примера 9.6

Пример 9.7. Разведывание предсказателей кредитного риска посредством бустинга

На рис. 9.20 представлены результаты выполнения процедуры бустинга (с использованием стоимости мягкого максимума и оптимизатора методом Ньютона) на наборе данных кредитных баллов немецкого банка, впервые представленном в примере 6.11 и состоящем из $P = 1000$ образцов, каждый из которых представляет собой набор статистик, извлеченных из кредитных заявок в немецкий банк. Двадцатиразмерные ($N = 20$) входные признаки в этом наборе данных включают баланс текущего счета физического лица в банке (признак 1), продолжительность (в месяцах) предыдущего кредита в банке (признак 2), платежный статус любого предыдущего кредита, полученного в банке (признак 3), и текущую стоимость его сбережений/акций (признак 6). Это именно те четыре главных признака, которые обнаруживаются с помощью бустинга, большинство которых положительно коррелируют с тем, что человек представляет хороший кредитный риск (рис. 9.20, *снизу*).

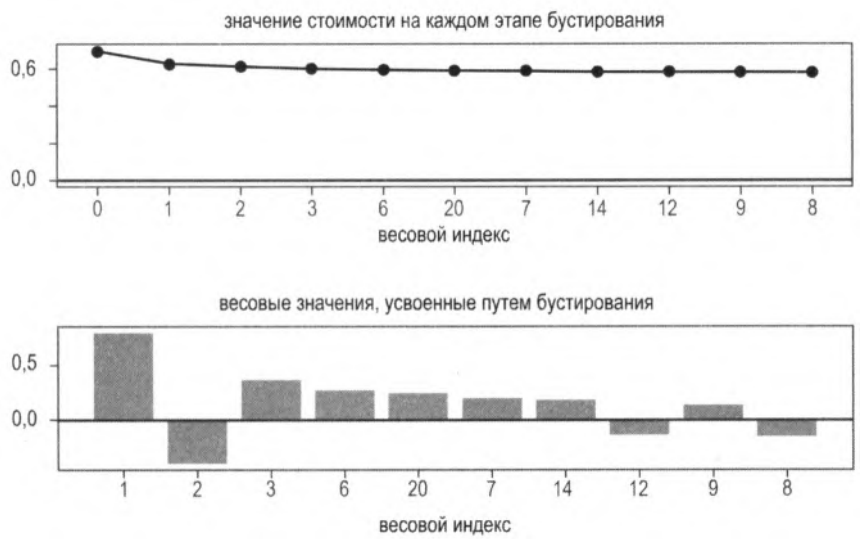


Рис. 9.20. Иллюстрация примера 9.7

9.6.3. Об эффективности бустинга как жадного алгоритма

Бустинг, по сути, является *жадным* алгоритмом, где на каждом этапе мы выбираем следующий наиболее важный (признаковый) вес и надлежаще его настраиваем, решая набор соответствующих подзадач. В то время как любой раунд бустинга требует решения ряда подзадач, каждая из них является минимизацией по отношению только к *одному весу*, и, следовательно, его решение обходится дешево независимо от используемой схемы локальной оптимизации. Это делает бустинг вычислительно эффективным подходом к отбору признаков и позволяет масштабировать его до наборов данных с крупным числом N входных размерностей. Однако слабость, присущая этому процессу, заключается в том, что при определении призна-

ковых весов по одному за раз взаимодействия между признаками/весами могут быть потенциально упущены.

Для того чтобы уловить эти потенциально пропущенные взаимодействия, можно было бы естественным образом расширить идею бустинга и попытаться добавлять в каждом раунде группу из R признаковых весов вместо только одного. Однако быстрый расчет показывает, что эта идея вскоре откажет при масштабировании. Для того чтобы определить первую наилучшую группу из R признаковых весов, на первом этапе этого подхода нам нужно было бы испытать каждую комбинацию из R весов путем решения подзадачи для каждой из них. Трудность здесь заключается в том, что существует комбинаторно много подгрупп размера R, точнее, $\binom{N}{R} = \frac{N!}{R!(N-R)!}$ таких подгрупп. А это слишком много задач, чтобы их можно было решить на практике, даже для малых и умеренных значений N и R (например, $\binom{100}{5} = 75\,287\,520$).

9.6.4. Взгляд на бустинг регрессии с точки зрения остатков

Напомним, что n-я подзадача на m-м этапе бустинга в случае регрессии методом наименьших квадратов призвана минимизировать

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_{m-1}\left(\mathbf{x}_p, \mathbf{w}\right) + w_n x_{n,p} - y_p\right)^2. \tag{9.16}$$

Если мы переставим члены в каждом слагаемом и обозначим

$$r_p^m = \left(y_p - \text{model}_{m-1}\left(\mathbf{x}_p, \mathbf{w}\right)\right), \tag{9.17}$$

тогда сможем записать стоимостную функцию наименьших квадратов в уравнении (9.16) эквивалентным образом как

$$\frac{1}{P}\sum_{p=1}^{P}\left(w_n x_{n,p} - r_p^m\right)^2, \tag{9.18}$$

где член r_p^m в правой части каждого слагаемого *фиксирован*, поскольку w_n — это единственный вес, настраиваемый тут. Указанный член r_p^m является *остатком* исходного выхода y_p после вычитания вклада модели model_m. Тогда мы можем рассуждать о каждом раунде бустинга как об определении следующего признака, который лучше всего коррелирует с остатком из предыдущего раунда.

9.7. Отбор признаков посредством регуляризации

В рассмотренной в предыдущем разделе технике отбора признаков на основе бустинга мы использовали жадный подход к отбору признаков, направленный снизу вверх: мы начинали с настройки смещения, а затем добавляли новые признаки

в нашу модель по одному за раз. В этом разделе мы вводим дополнительный подход к отбору признаков, именуемый *регуляризацией*. Вместо надстраивания модели, начиная снизу, в регуляризации мы принимаем взгляд, направленный сверху вниз, и начинаем с законченной модели, включающей все наши входные признаки, а затем постепенно удаляем входные признаки меньшей важности. Мы делаем это путем добавления в нашу стоимость второй функции (именуемой *регуляризатором*), которая штрафует все веса, заставляя нашу модель сжимать весовые значения, связанные с менее важными входными признаками.

9.7.1. Регуляризация с использованием норм весовых векторов

Простая линейная комбинация стоимостной функции g и вспомогательной функции h:

$$f(\mathbf{w}) = g(\mathbf{w}) + \lambda h(\mathbf{w}) \tag{9.19}$$

на языке машинного обучения часто упоминается как *регуляризация*, причем функция h называется регуляризатором, а параметр $\lambda \geq 0$ — *регуляризацией* или *штрафным параметром*.

При $\lambda = 0$ линейная комбинация в уравнении (9.19) сводится к исходной стоимостной функции g. По мере увеличения λ функции g и h начинают *конкурировать за доминирование*, причем линейная комбинация приобретает свойства обеих функций. Когда мы устанавливаем λ равным все более и более крупному значению, функция h начинает доминировать над комбинацией, в конечном счете полностью заглушая g, и мы в итоге получаем по существу шкалированную версию регуляризатора h.

В приложениях машинного обучения в качестве регуляризатора h очень широко применяется векторная или матричная норма модельных параметров (*см. разд. П3.5*), причем разные нормы используются для продуцирования различных эффектов в усвоении моделей машинного обучения. Далее мы покажем, как несколько общепринятых векторных норм влияют на минимизацию обобщенной стоимостной функции g.

Пример 9.8. Регуляризация с использованием ℓ_0-нормы

Векторная ℓ_0-норма вектора $\mathbf{w}$, записываемая как $\|\mathbf{w}\|_0$, измеряет его длину или магнитуду как

$$\|\mathbf{w}\|_0 = \text{число ненулевых элементов в } \mathbf{w}. \tag{9.20}$$

Регуляризируя стоимость g с использованием этого регуляризатора (т. е. $f(\mathbf{w}) = g(\mathbf{w}) + \lambda\|\mathbf{w}\|_0$), мы штрафуем регуляризованную стоимость f для каждого ненулевого элемента $\mathbf{w}$, т. к. каждый такой элемент добавляет одну единицу в $\|\mathbf{w}\|_0$. И наоборот, при минимизировании f две функции g и $\|\mathbf{w}\|_0$ конкурируют за доминирование, причем g желает, чтобы $\mathbf{w}$ разрешилась как точка вблизи ее минимизатора, тогда как регуляризатор $\|\mathbf{w}\|_0$ стремится определить $\mathbf{w}$, который имеет как

можно меньше ненулевых элементов, или, другими словами, весовой вектор $\mathbf{w}$, который очень *разрежен*.

Пример 9.9. Регуляризация с использованием ℓ_1-нормы

Векторная ℓ_1-норма, записываемая как $\|\mathbf{w}\|_1$, измеряет магнитуду вектора $\mathbf{w}$ как

$$\|\mathbf{w}\|_1 = \sum_{n=0}^{N} |w_n| . \tag{9.21}$$

Регуляризируя стоимость g с использованием этого регуляризатора (т. е. $f(\mathbf{w}) = g(\mathbf{w}) + \lambda\|\mathbf{w}\|_1$), мы штрафуем регуляризованную стоимость, основываясь на сумме абсолютных значений элементов вектора $\mathbf{w}$.

И наоборот, при минимизировании этой суммы две функции g и $\|\mathbf{w}\|_1$ конкурируют за доминирование, причем g желает, чтобы $\mathbf{w}$ разрешился как точка вблизи его минимизатора, тогда как регуляризатор $\|\mathbf{w}\|_1$ стремится определить $\mathbf{w}$, который мал с точки зрения абсолютного значения каждой его компоненты, но также поскольку ℓ_1-норма тесно связана с ℓ_0-нормой (*см. разд. П3.5*), такой, который имеет мало ненулевых элементов и, следовательно, является *разреженным*.

Пример 9.10. Регуляризация с использованием ℓ_2-нормы

Векторная ℓ_2-норма вектора $\mathbf{w}$, записываемая как $\|\mathbf{w}\|_2$, измеряет его магнитуду как

$$\|\mathbf{w}\|_2 = \sqrt{\sum_{n=0}^{N} w_n^2} . \tag{9.22}$$

Регуляризируя стоимость g с использованием этого регуляризатора (т. е. $f(\mathbf{w}) = g(\mathbf{w}) + \lambda\|\mathbf{w}\|_2$), мы штрафуем регуляризованную стоимость, основываясь на сумме *квадратов* элементов в $\mathbf{w}$.

И наоборот, при минимизировании этой суммы две функции g и $\|\mathbf{w}\|_2$ конкурируют за доминирование, причем g желает, чтобы $\mathbf{w}$ разрешилась как точка вблизи его минимизатора, тогда как регуляризатор $\|\mathbf{w}\|_2$ стремится определить $\mathbf{w}$, который мал в том смысле, что все его записи имеют малое квадратичное значение.

До настоящего момента мы уже видели ряд примеров регуляризации ℓ_2, например в контексте многомерной логистической (softmax) классификации в *разд. 6.4.6* и метода опорных векторов в *разд. 6.5.4*.

9.7.2. Отбор признаков посредством ℓ_1-регуляризации

В контексте машинного обучения, индуцируя обнаружение разреженных весовых векторов, регуляризаторы ℓ_0 и ℓ_1 помогают раскрывать идентичность *наиболее важных признаков* набора данных. Это обусловлено тем, что когда мы задействуем

такие нормы, как регуляризатор, мы принуждаем восстановленные модельные веса быть довольно *разреженными* (при условии, что мы установили λ соответствующим образом), при этом остаются только те веса, которые связаны с наиболее важными модельными признаками. Это делает любую норму (по крайней мере в теории) вполне подходящей для задачи отбора признаков.

Из двух норм, индуцирующих разреженность, описанных в примерах 9.8 и 9.9, ℓ_0-норма (хотя и способствует разреженности непосредственно и в наибольшей степени) является наиболее сложной для применения из-за ее *прерывистой* природы, что довольно затрудняет минимизацию ℓ_0-регуляризованной стоимостной функции. Хотя ℓ_1-норма индуцирует разреженность в меньшей степени, она вместе с тем является *выпуклой* и *дифференцируемой* (почти везде), что делает возможным использование методов первого порядка для ее минимизации. Ввиду такого практического преимущества ℓ_1-норма на сегодняшний день является наиболее часто используемым регуляризатором для отбора признаков на практике.

Наконец, вспомните, как подробно описано в предыдущем разделе, что при выполнении отбора признаков нас интересует только определение важности *признаковых* весов $w_1, w_2, ..., , w_N$, отсюда нам нужно регуляризировать только их (а не вес смещения w_0). Из этого вытекает, что наша регуляризация примет форму

$$f(\mathbf{w}) = g(\mathbf{w}) + \lambda \sum_{n=1}^{N} |w_n| . \tag{9.23}$$

Задействуя нашу индивидуальную нотацию для смещения и признаковых весов (использованную, например, в *разд. 6.4.6*)

$$\text{(смещение)}: \; b = w_0 \qquad \text{(признаковые веса)}: \; \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}, \tag{9.24}$$

мы можем записать эту общую ℓ_1-регуляризованную стоимостную функцию эквивалентным образом как

$$f(b, \boldsymbol{\omega}) = g(b, \boldsymbol{\omega}) + \lambda \|\boldsymbol{\omega}\|_1 . \tag{9.25}$$

9.7.3. Отбор верного регуляризационного параметра

Поскольку признаки отбираются в целях *интерпретации человеком*, значение λ может быть установлено на основе нескольких факторов. Эталонное значение для λ можно выбрать исходя из желания разведать набор данных, отыскивая значение, которое обеспечивает достаточную разреженность при сохранении низкой стоимости. Значение λ также может быть подобрано целиком на основе выборочной статистики набора данных посредством процедуры, именуемой *перекрестной валидацией*, которую мы обсудим в *главе 11*.

Точно так же, как для бустинга, и независимо от того, как мы отбираем λ, важно стандартно нормализовать входные данные, чтобы мы могли справедливо сравни-

вать влияние каждого входного признака, рассматривая их настроенные весовые значения, восстановленные в результате минимизации регуляризованной стоимостной функции.

Пример 9.11. Разведывание предсказателей цен на жилье посредством регуляризации

В этом примере мы сформируем ℓ_1-регуляризованную стоимость наименьших квадратов с использованием набора данных о ценах на жилье в Бостоне (впервые введенного в примере 5.5 и используемого в контексте отбора признаков на основе бустинга в примере 9.6) и рассмотрим 50 равноотстоящих значений для λ в интервале [0; 130]. Для каждого значения λ в этом интервале, начиная с 0 (как показано на рис. 9.21, *сверху*), мы выполняем градиентный спуск с фиксированным числом шагов и параметром длины шага, чтобы минимизировать регуляризованную стоимость. К тому времени, когда λ устанавливается равным наибольшему значению в указанном интервале, три главнейших веса восстанавливаются за счет настройки параметров (как показано на рис. 9.21, *снизу*), что соответствует признакам 6, 13 и 11. Первые два признака (например, признаки 6 и 13) также были определены как важные посредством бустинга в примере 9.6.

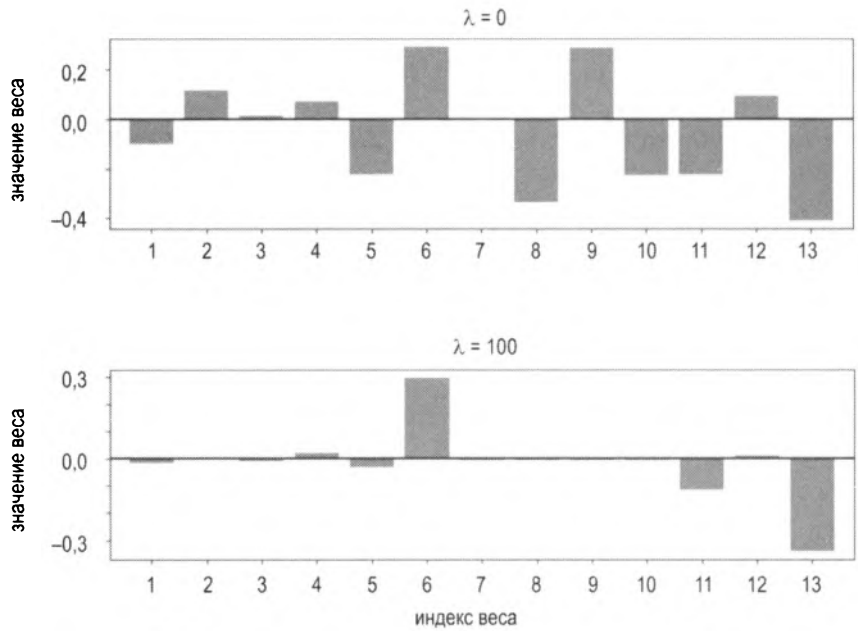


Рис. 9.21. Иллюстрация примера 9.11

Пример 9.12. Разведывание предсказателей кредитного риска посредством регуляризации

В этом примере мы минимизируем ℓ_1-регуляризованную стоимость двухклассового softmax над набором данных кредитных баллов немецкого банка (впервые введенном в примере 9.20), используя 50 равноотстоящих значений для λ в интервале

[0; 130]. Для каждого значения λ в этом интервале, начиная с 0 (как показано на рис. 9.22, *сверху*), мы выполняем градиентный спуск с фиксированным числом шагов и параметром длины шага, чтобы минимизировать регуляризованную стоимость.

К моменту, когда $\lambda \approx 40$, остаются пять главнейших весов, соответствующих признакам 1, 2, 3, 6 и 7 (как показано на рис. 9.22, *снизу*, где $\lambda = 130$). Первые четыре из этих признаков также были определены как важные с помощью бустинга в примере 9.7.

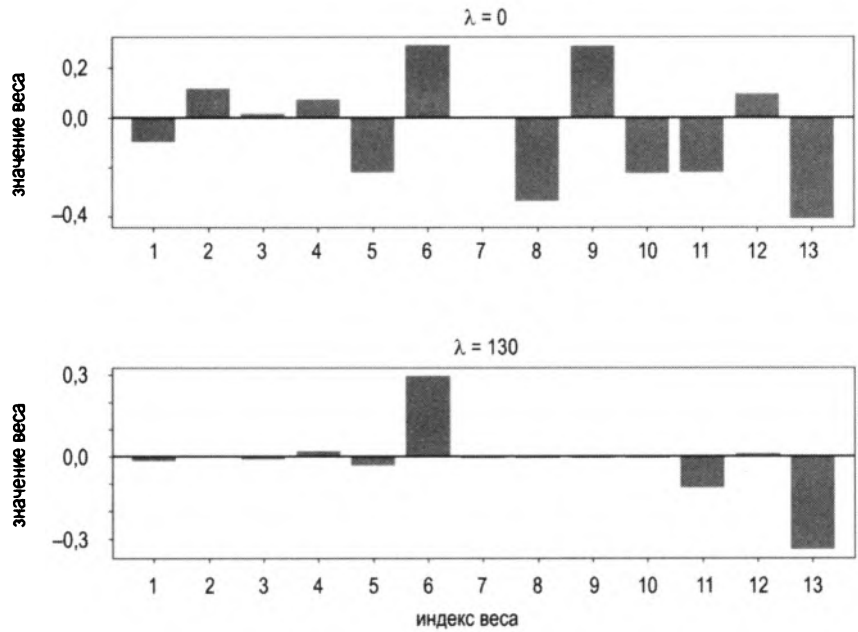


Рис. 9.22. Иллюстрация примера 9.12

9.7.4. Сравнение регуляризации и бустинга

В то время как бустинг является эффективной жадной схемой, идея регуляризации, подробно описанная в этом разделе, характеризуется вычислительной интенсивностью при ее выполнении, т. к. для каждого апробируемого значения λ должен быть завершен полный прогон локальной оптимизации. С другой стороны, в то время как бустинг является подходом, направленным снизу вверх, который идентифицирует отдельные признаки по одному, регуляризация, наоборот, использует подход, направленный сверху вниз, и идентифицирует сразу все важные признаки. В принципе, это позволяет регуляризации выявлять группы важных признаков, коррелирующих с выходом таким взаимосвязанным образом, который может быть упущен при бустинге.

9.8. Заключение

В этой главе мы рассмотрели фундаментальные методы *инженерии признаков* и *отбора признаков*.

Мы начали с обсуждения методологи инженерии признаков, которые используются в качестве шагов предобработки данных практически для всех задач машинного обучения и которые мы будем широко использовать в остальной части книги. В *разд. 9.2* мы подробно описали гистограммные признаки, которые аккуратно подытоживают содержимое в данных и могут быть сконструированы практически для любой модальности данных. В *разд. 9.3–9.5* были описаны различные методы шкалирования входных данных, включая *стандартную нормализацию* и *декоррелирующее преобразование*. Эти методы стандартизируют входные данные и улучшают топологию стоимостных функций машинного обучения, значительно облегчая их минимизирование (в особенности при использовании первопорядковых методов, описанных в *главе 3*). Наконец, в *разд. 9.6* и *9.7* мы описали два взаимодополняющих подхода к отбору признаков — *бустинг* и *регуляризацию*, которые обеспечивают возможность прямого анализа человеком массы индивидуальных признаков, включенных в натренированную модель машинного обучения.

9.9. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:
github.com/jermwatt/machine_learning_refined.

9.1. Спамная электронная почта

Повторить эксперимент, описанный в примере 9.2, используя любую желаемую схему локальной оптимизации и стоимость двухклассового мягкого максимума. Позаботиться о создании графика, подобного показанному на рис. 9.5, чтобы сравнить результаты использования каждой комбинации признаков.

9.2. Классификация набора данных MNIST: пикселы в сравнении с реберными признаками

Повторить эксперимент, описанный в примере 9.3, и создать пару графиков истории стоимостной функции/неправильного классифицирования, как показано на рис. 9.11. В зависимости от деталей вашей имплементации ваши результаты могут незначительно отличаться от приведенных в примере.

9.3. Студенческая кредитная задолженность

Сгенерировать два контурных графика стоимостной функции наименьших квадратов над набором данных студенческой кредитной задолженности [2], показанным на рис. 1.8, а также его стандартно нормализованную версию. Сравнить совокупную топологию каждого контурного графика и описать, почему график, связанный со стандартно нормализованными данными, будет гораздо легче оптимизироваться

посредством градиентного спуска. Действительно, подгонка к исходному набору данных посредством градиентного спуска здесь почти невозможна. Минимизировать стоимость наименьших квадратов над стандартно нормализованной версией данных посредством градиентного спуска и воспроизвести график, показанный на рис. 1.8.

9.4. Наименьшие квадраты и совершенно округлые контуры: часть 1

В примере 9.4 мы видели, как контурный график стоимости наименьших квадратов над однорегрессионным ($N = 1$) набором данных изменялся с *высокоэллиптического* контура до совершенно округлого после того, как данные были стандартно нормализованы (см. рис. 9.16). Показать, что контурный график стоимости наименьших квадратов по сравнению со стандартно нормализованными данными всегда будет совершенно округлым при $N = 1$. Затем описать, почему это *не* обязательно происходит при $N > 1$.

9.5. Набор данных по раку молочной железы

Выполнить линейную двухклассовую классификацию на наборе данных по раку молочной железы [43], состоящем из $P = 569$ точек данных, используя надлежащую стоимостную функцию и локальный оптимизатор. Заполнить все недостающие значения в данных с помощью вменения среднего значения и сообщить о наилучшей частоте неправильного классифицирования, которую вы смогли достичь. Выяснить, как быстро вы можете минимизировать вашу стоимостную функцию по сравнению с исходными и стандартно нормализованными данными, используя одинаковые параметры (например, одинаковое число шагов, параметр длины шага и т. д.).

9.6. Декоррелирующее преобразование и стоимость наименьших квадратов для линейной регрессии

Стоимость наименьших квадратов для линейной регрессии представляет собой идеальный пример того, как декоррелирующее преобразование набора данных положительно влияет на топологию стоимостной функции машинного обучения (что значительно облегчает выполнение надлежащего минимизирования). Это обусловлено тем, что — как мы покажем формально ниже — декоррелирующее преобразование входов в регрессионном наборе данных приводит к стоимости наименьших квадратов для линейной регрессии, которая имеет *полностью округлые контуры* и поэтому очень легко надлежаще минимизируется. Хотя декоррелирующее преобразование не улучшает все стоимостные функции в такой положительной степени, этот пример все же показывает влияние PCA-сферирования на улучшение топологии стоимостных функций машинного обучения в целом.

Для того чтобы увидеть, как декоррелирующее преобразование совершенно сглаживает контуры стоимости наименьших квадратов для линейной регрессии, сначала обратите внимание (как подробно описано в *разд. 5.9.1*), что стоимость наименьших квадратов всегда (независимо от используемого набора данных) является выпуклой квадратичной функцией общей формы $g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}$, где, в част-

ности, $\mathbf{C} = \frac{1}{P}\sum_{p=1}^{P} \dot{\mathbf{x}}_p \dot{\mathbf{x}}_p^T$. Если вход регрессионного набора данных PCA-сферирован, тогда нижняя $N \times N$-подматрица $\mathbf{C}$ задается как $\frac{1}{P}\mathbf{S}\mathbf{S}^T$, где $\mathbf{S}$ определяется в уравнении (9.6). Однако из-за самого характера определения $\mathbf{S}$ мы имеем, что $\frac{1}{P}\mathbf{S}\mathbf{S}^T = \frac{1}{P}\mathbf{I}_{N\times N}$, где $\mathbf{I}_{N\times N}$ — это $N \times N$-матрица тождественности, и отсюда в общем случае $\mathbf{C} = \frac{1}{P}\mathbf{I}_{(N+1)\times(N+1)}$. Другими словами, стоимость наименьших квадратов над PCA-сферированными входными данными является выпуклой квадратичной функцией, в которой все собственные значения равны 1, а это подразумевает, что данная квадратичная функция имеет совершенно округлые контуры (*см. разд. 4.2.2*).

9.7. Сравнение стандартной нормализации с декоррелирующим преобразованием на наборе данных MNIST

Сравнить выполнение десяти шагов градиентного спуска с использованием стоимости многоклассового мягкого максимума над более 50 000 случайными цифрами из набора данных MNIST (введенного в примере 7.10), стандартно нормализованной версии и PCA-сферированной версии данных. Для каждого прогона использовать наибольшую фиксированную длину шага формы $\alpha = 10^{\gamma}$ для целого числа γ, которое вы найдете, чтобы произвести спуск. Построить график, сравнивающий ход выполнения каждого прогона с точки зрения стоимостной функции и числа неправильных классифицированй. Вдобавок обеспечить, чтобы инициализация каждого прогона была довольно малой, в особенности первого прогона, где вы вообще не применяете нормализацию к входным данным, т. к. каждая сырая входная точка этого набора данных имеет большую магнитуду. В случае инициализации сырых данных в точке, слишком удаленной от начала координат, может легко возникнуть числовое переполнение, приводящее к получению значений `nan` или `inf`, разрушая остальную часть соответствующего прогона локальной оптимизации.

9.8. Наименьшие квадраты и совершенно округлые контуры: часть 2

В упражнении 9.4 мы видели, как декоррелирующее преобразование входных данных изменяет топологию стоимости наименьших квадратов, в результате чего ее контуры становятся совершенно округлыми. Объяснить, каким образом это чрезвычайно облегчает минимизацию такой PCA-сферированной стоимости наименьших квадратов. В частности, объяснить, как — независимо от набора данных — такая стоимость может быть минимизирована лишь с помощью одного "упрощенного" ньютонова шага, как описано в *разд. П1.8.1*, где мы игнорируем внедиагональные элементы матрицы Гессе при выполнении ньютонова шага.

9.9. Разведывание предсказателей цен на жилье

Имплементировать процедуру бустинга, описанную в *разд. 9.6.1* для линейной регрессии, задействовав стоимость наименьших квадратов, и повторить эксперимент, описанный в примере 9.6. От вас не требуется воспроизводить визуализации, пред-

ставленные на рис. 9.19, но убедитесь, что вы можете прийти к аналогичным выводам, описанным в указанном примере.

9.10. Предсказание автомобильного пробега в милях на галлон израсходованного топлива

Выполнить $M = 6$ раундов бустинга на автомобильном наборе данных Auto-MPG, представленном в примере 5.6, задействовав стоимостную функцию наименьших квадратов, чтобы осуществить отбор признаков. Предоставить интерпретацию трех наиболее важных признаков, которые вы найдете, и их соотношения с результатами.

9.11. Изучение важных предсказателей кредитного риска

Имплементировать процедуру бустинга, описанную в *разд. 9.6.1* для линейной классификации, задействовав стоимость двухклассового мягкого максимума, и повторить эксперимент, описанный в примере 9.7. От вас не требуется воспроизводить визуализации, показанные на рис. 9.20, но убедитесь, что вы можете прийти к тем же выводам, что и в приведенном примере.

9.12. Разведывание предсказателей цен на жилье

Имплементировать процедуру регуляризации, описанную в *разд. 9.7*, для линейной регрессии, задействовав стоимость наименьших квадратов, и повторить эксперимент, описанный в примере 9.11. От вас не требуется воспроизводить визуализации, показанные на рис. 9.21, но убедитесь, что вы можете прийти к тем же выводам, что и в приведенном примере.

9.13. Изучение важных предсказателей кредитного риска

Имплементировать процедуру регуляризации, описанную в *разд. 9.7* для линейной классификации, задействовав стоимость двухклассового мягкого максимума, и повторить эксперимент, описанный в примере 9.12. От вас не требуется воспроизводить визуализации, показанные на рис. 9.22, но убедитесь, что вы можете прийти к тем же выводам, что и в приведенном примере.

Часть III

Нелинейное обучение

10 Принципы инженерии нелинейных признаков

10.1. Введение

До сих пор мы имели дело с главенствующими парадигмами обучения с учителем и без учителя, подразумевая — для простоты — допущение о *линейных* моделях. В этой главе мы отбросим это упрощающее допущение и начнем наш набег на нелинейное машинное обучение, изучив *инженерию* нелинейных *признаков* в контексте обучения с учителем и без учителя. В то время как инженерия нелинейных признаков действительно возможна только тогда, когда набор данных (или явление, его генерирующее) хорошо понятен, понимание базиса инженерии нелинейных признаков чрезвычайно ценно, поскольку оно позволяет нам ввести ряд ключевых понятий в относительно простую среду, которая будет присутствовать в нашем обсуждении нелинейного обучения в будущем. Как мы увидим, эти важные понятия включают в себя разнообразие формальных математических и программных принципов, инструменты моделирования и жаргонные термины.

10.2. Нелинейная регрессия

В этом разделе мы введем общий каркас нелинейной регрессии посредством *инженерии* нелинейных преобразований признаков, основываясь на визуальной интуиции, вместе с рядом примеров. Форма многих классических законов из естественных наук — как впервые обсуждалось в примере 1.2 — может быть выведена посредством описанных здесь общих методов (как мы разведаем в примере 10.2 и упражнениях 10.2 и 10.4).

10.2.1. Принципы моделирования

В *главе 5* мы подробно описали базовую линейную модель для регрессии

$$\text{model}(\mathbf{x}, \mathbf{w}) = w_0 + x_1 w_1 + x_2 w_2 + ... + x_N w_N \tag{10.1}$$

или, компактнее:

$$\text{model}(\mathbf{x}, \mathbf{w}) = \dot{\mathbf{x}}^T \mathbf{w}, \tag{10.2}$$

где

$$\dot{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \text{ и } \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{10.3}$$

Для того чтобы настроить параметры нашей линейной модели над обобщенным набором данных из P точек $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ так, чтобы он представлял данные (пример которых показан на рис. 10.1, *слева*) как можно лучше, или, если формулировать алгебраически, так, чтобы мы имели[1]:

$$\dot{\mathbf{x}}_p^T \mathbf{w} \approx y_p, \qquad p = 1, 2, ..., P, \tag{10.4}$$

мы минимизируем надлежащую регрессионную стоимостную функцию, например стоимость наименьших квадратов

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\dot{\mathbf{x}}_p^T \mathbf{w} - y_p\right)^2. \tag{10.5}$$

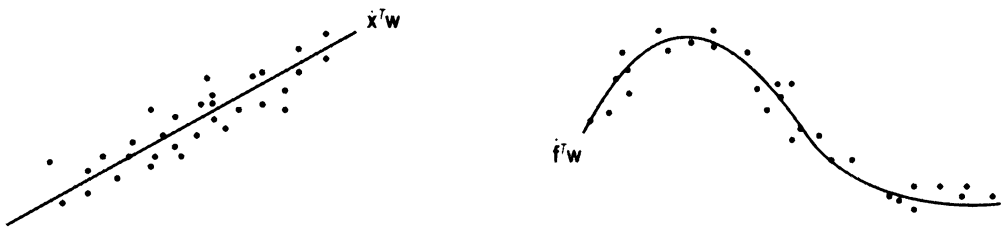


Рис. 10.1. Иллюстрация линейной регрессии. Здесь подгонка к данным определяется линейной моделью $\dot{\mathbf{x}}^T\mathbf{w}$ (*слева*). Нелинейная регрессия достигается путем впрыскивания в нашу модель нелинейных признаковых преобразований (*справа*). Здесь подгонка к данным является нелинейной кривой, определяемой выражением $\dot{\mathbf{f}}^T\mathbf{w}$

Мы можем перейти от *линейной* регрессии к общей *нелинейной*, как в ее принципах, так и в имплементации, просто заменив линейную модель, используемую в построении нашей линейной регрессии, нелинейной. Например, вместо линейной модели мы можем использовать нелинейную, предусматривающую одну-единственную нелинейную функцию f (например, квадратичную, синусно-волновую, логистическую функцию и т. д., которые могут быть параметризованы или непараметризованы). На жаргоне машинного обучения такая нелинейная функция f часто называется нелинейным *признаковым преобразованием* (или просто *признаком*), поскольку она преобразует наши исходные входные признаки $\mathbf{x}$. Тогда наша соответствующая нелинейная модель примет форму

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f(\mathbf{x})w_1, \tag{10.6}$$

где множество Θ содержит все параметры модели, включая веса линейной комбинации (здесь w_0 и w_1), а также потенциальные внутренние параметры самой функции f.

---

[1] Следуя компактной нотации из уравнения (10.3), $\dot{\mathbf{x}}_p^T = \begin{bmatrix} 1 & x_{1,p} & x_{2,p} & \dots & x_{N,p} \end{bmatrix}$, где $x_{n,p}$ — это n-й элемент в $\mathbf{x}_p$.

Мы можем расширить эту идею, чтобы создавать нелинейные модели, которые используют больше чем просто одно-единственное нелинейное признаковое преобразование. В общем случае мы можем сформировать нелинейную модель как взвешенную сумму B нелинейных функций нашего входа:

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1(x)w_1 + f_2(x)w_2 + \ldots + f_B(x)w_B, \tag{10.7}$$

где $f_1, f_2, \ldots, f_B$ — нелинейные (параметризованные или непараметризованные) функции, а все w с w_0 по w_B вместе с любыми дополнительными весами, внутренними для нелинейных функций, представлены в весовом множестве Θ.

Независимо от того, какие нелинейные функции мы выбираем, шаги, которые мы предпринимаем для окончательного разрешения такой модели для целей регрессии, полностью аналогичны тому, что мы видели для простого случая линейной регрессии. По аналогии с линейным случаем здесь также полезно записать обобщенную нелинейную модель из уравнения (10.7) компактнее, как

$$\text{model}(\mathbf{x}, \Theta) = \dot{\mathbf{f}}^T \mathbf{w}, \tag{10.8}$$

обозначив

$$\dot{\mathbf{f}} = \begin{bmatrix} 1 \\ f_1(x) \\ f_2(x) \\ \vdots \\ f_B(x) \end{bmatrix} \text{ и } \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_B \end{bmatrix}. \tag{10.9}$$

Опять-таки, для того чтобы настроить параметры нашей обобщенной нелинейной модели над набором данных из P точек так, чтобы она представляла данные (пример которых показан на рис. 10.1, *справа*) как можно лучше, или, если сформулировать алгебраически, так, чтобы мы имели[2]

$$\dot{\mathbf{f}}_p^T \mathbf{w} \approx y_p, \qquad p = 1, 2, \ldots, P, \tag{10.10}$$

мы минимизируем надлежащую регрессионную стоимостную функцию над Θ, например стоимость наименьших квадратов:

$$g(\Theta) = \frac{1}{P} \sum_{p=1}^{P} \left(\dot{\mathbf{f}}_p^T \mathbf{w} - y_p \right)^2. \tag{10.11}$$

Несмотря на все эти структурные сходства между линейным и нелинейным каркасами, все еще остается один вопрос: как определить соответствующие нелинейные признаковые преобразования для нашей модели и их число B для обобщенного набора данных? Это действительно одна из самых важных проблем, с которыми мы сталкиваемся в машинном обучении, и как раз ее мы и обсудим подробно в настоящей главе, а также в нескольких последующих.

---

[2] Здесь $\dot{\mathbf{f}}_p^T = \left[1 \; f_1(\mathbf{x}_p) \; f_2(\mathbf{x}_p) \; \ldots \; f_B(\mathbf{x}_p) \right]$, следуя компактной нотации в уравнении (10.9).

10.2.2. Инженерия признаков

Начнем наше расследование нелинейной регрессии с обсуждения нескольких простых примеров, где мы сможем определить вид и число нелинейных признаков, которые нам нужны, *визуализируя* данные и полагаясь на наши собственные способности распознавания шаблонов с целью определения соответствующих нелинейностей. Это экземпляр того, что в более широком смысле называется *инженерией признаков*, в которой функциональная форма нелинейностей для использования в моделях машинного обучения определяется (или конструируется) людьми с помощью их опыта, знаний предметной области, интуиции в отношении решаемой задачи и т. д.

Пример 10.1. Моделирование волны

На рис. 10.2, слева показан нелинейно регрессионный набор данных. Из-за волнистого внешнего вида этих данных мы можем обоснованно предложить нелинейную модель, состоящую из синусной функции

$$f(x) = \sin\left(v_0 + xv_1\right), \tag{10.12}$$

параметризованной настраиваемыми весами v_0 и v_1, причем наша регрессионная модель задется как

$$\text{model}\left(x, \Theta\right) = w_0 + f(x)w_1, \tag{10.13}$$

где $\Theta = \{w_0, w_1, v_0, v_1\}$. Интуитивно представляется, что эта модель могла бы хорошо вписаться в данные, если бы все ее параметры были надлежаще настроены. На левой панели рис. 10.2 мы показываем результирующую модель, подогнанную к данным (зеленым цветом) путем минимизирования стоимости наименьших квадратов посредством градиентного спуска.

После того как наши веса полностью настроены, обратите внимание на то, что наша модель задается *линейно* в терминах ее признакового преобразования. Это озна-

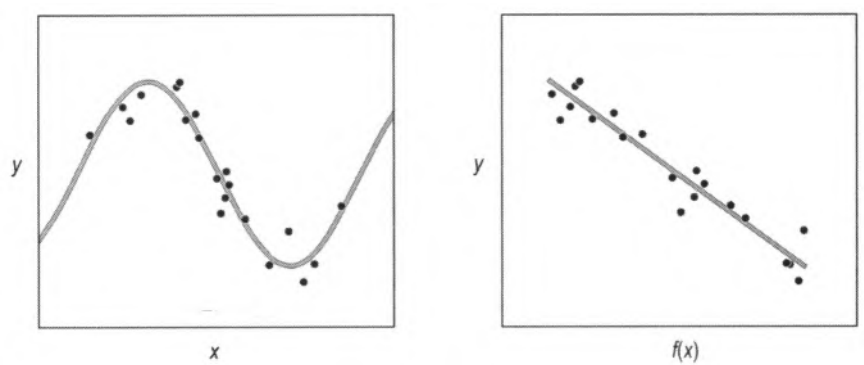


Рис. 10.2. Иллюстрация примера 10.1. Нелинейный регрессионный набор данных и соответствующе настроенная модель, определенная в уравнениях (10.12) и (10.13) (*слева*). Взгляд на те же самые данные и настроенную модель в *пространстве преобразованных признаков* (*справа*)

чает, что если мы построим график преобразованной версии нашего набора данных, т. е. $\{(f(x_p), y_p)\}_{p=1}^{P}$, где внутренние признаковые веса v_0 и v_1 были оптимально настроены, наша модель вписывается в эти преобразованные данные *линейно*, как показано на рис. 10.2, *справа*. Другими словами, в *пространстве этих преобразованных признаков*, входная ось которого задается функцией $f(x)$, а выходная — y, наша настроенная нелинейная модель становится линейной.

Наконец, обратите внимание, что, как подробно описано в *разд. 9.3* для случая линейной регрессии, с нелинейной регрессией по-прежнему очень выгодно задействовать стандартную нормализацию для шкалирования наших входных данных при использовании схем градиентного спуска для минимизирования соответствующей стоимостной функции наименьших квадратов g. На рис. 10.3 показаны графики истории стоимостной функции, полученные в результате прогона градиентного спуска после задействования исходной ненормализованной (в черном цвете) и стандартно нормализованной (в сиреневом цвете) версий входных данных. Сравнивая эти две истории, мы видим, что при использовании стандартно нормализованного входа значение стоимостной функции существенно более низкое.

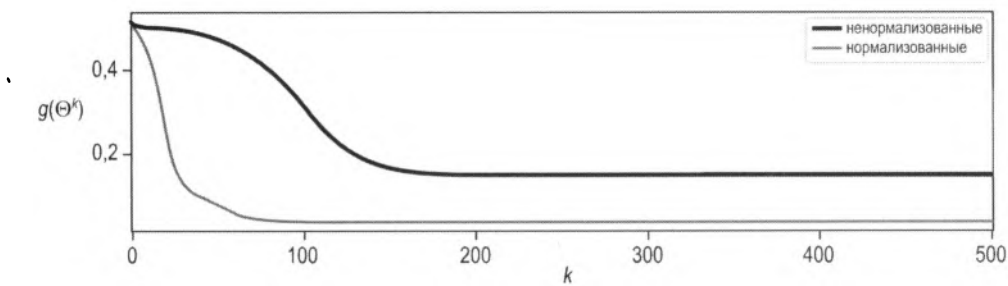


Рис. 10.3. Иллюстрация примера 10.1

Пример 10.2. Галилей и гравитация

В 1638 году Галилео Галилей, печально известный своим изгнанием из католической церкви за то, что осмелился утверждать, что Земля вращается вокруг Солнца, а не наоборот (как считалось в то время), опубликовал свою последнюю книгу — "Рассуждения и математические демонстрации, относящиеся к двум новым наукам" [44]. В этой книге, написанной в стиле беседы между тремя людьми в традиции Аристотеля, он обосновал свои экспериментальные и философские доказательства понятия равномерно ускоренного физического движения. В частности, у Галилея (и других) имелось интуитивное представление о том, что ускорение объекта, вызванное (силой, которую мы теперь называем) гравитацией, является равномерным во времени или, другими словами, что расстояние, на которое объект смещается, прямо пропорционально времени, которое было затрачено на это перемещение, в квадрате. Эта связь была эмпирически подтверждена с помощью следующего гениально простого эксперимента, проведенного Галилеем.

Многократно пуская металлический шар вниз по рифленому 5,5-метровому куску дерева, установленному под наклоном, как показано на рис. 10.4, Галилей засекал время, которое шару требовалось, чтобы пройти 1/4, 1/2, 2/3, 3/4 и весь путь вниз по деревянной рампе[3].

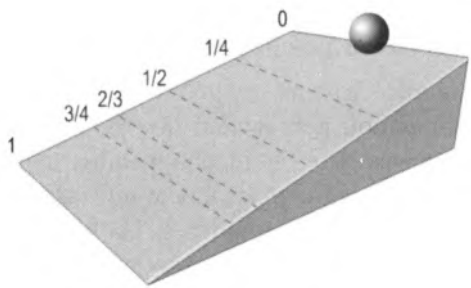


Рис. 10.4. Иллюстрация примера 10.2. Фигуральная иллюстрация экспериментальной рамповой установки Галилея, используемой для исследования взаимосвязи между временем и расстоянием падения объекта из-за силы тяжести. При выполнении этого эксперимента, он многократно пускал шар вниз по рампе и подсчитывал, сколько времени требовалось на то, чтобы пройти 1/4, 1/2, 2/3, 3/4 и весь путь вниз по рампе

Данные из современной реконструкции [45] этого эксперимента (усредненные по 30 испытаниям) показаны на рис. 10.5, *слева*, где входной осью является время (в секундах), а выходной — часть пандуса, пройденная шаром во время эксперимента. Данные здесь показывают нелинейную *квадратичную зависимость* между входом и выходом. Это транслируется в использование квадратичной модели

$$\text{model}(x, \Theta) = w_0 + f_1(x)w_1 + f_2(x)w_2 \qquad (10.14)$$

с двумя *непараметризованными* признаковыми преобразованиями: тождественным преобразованием $f_1(x) = x$ и квадратичным преобразованием $f_2(x) = x^2$. Замена $f_1(x)$ и $f_2(x)$ в уравнении (10.14) на x и x^2 дает уже знакомую квадратичную форму $w_0 + xw_1 + x^2w_2$.

После стандартного нормализирования входов этого набора данных (*см. разд. 9.3*) мы минимизируем соответствующую стоимость наименьших квадратов посредством градиентного спуска и строим график соответствующей наилучшей нелинейной подгонки на исходных данных на рис. 10.5, *слева*. Поскольку эта модель представляет собой линейную комбинацию двух признаковых преобразований (плюс вес смещения), мы также можем визуализировать соответствующую ей *линейную подгонку* в пространстве преобразованных признаков, как показано на рис. 10.5,

---

[3] Почему Галилей просто не сбрасывал шар с некоторой высоты и не определял количество времени, требуемого для того, чтобы достичь определенных расстояний до земли? Да потому, что до того не существовало надежного способа измерения времени. В результате для своих экспериментов с рампой ему пришлось использовать *водяные часы*! Интересно, что именно Галилей в своих исследованиях маятника направил человечество по пути к своему первому надежному хронометру.

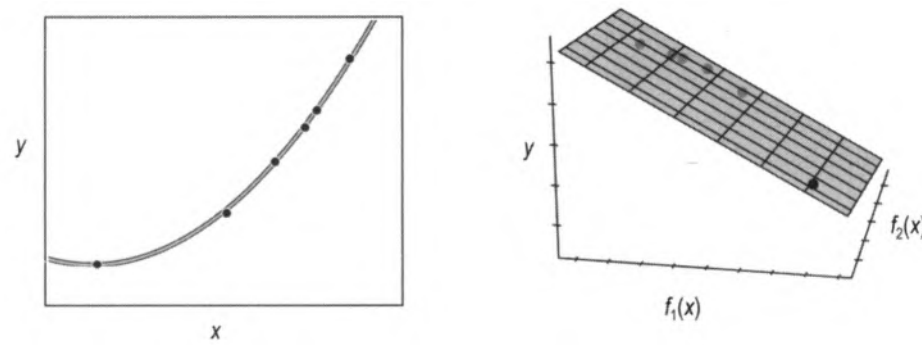


Рис. 10.5. Иллюстрация примера 10.2. Данные из современной реконструкции знаменитого эксперимента, выполненного Галилеем, вместе с хорошо настроенной квадратичной моделью (*слева*). Взгляд на тот же самый набор данных и модель в пространстве преобразованных признаков (*справа*)

справа. В этом пространстве входные оси задаются соответственно $f_1(x)$ и $f_2(x)$, а наши преобразованные точки — тройкой $\left(f_1(x_p), f_2(x_p), y_p\right)$.

10.2.3. Имплементация на Python

Далее мы продемонстрируем универсальный способ имплементации обобщенной нелинейной модели, показанной в уравнении (10.8), обобщив нашу первоначальную линейную имплементацию из *разд. 5.2.4*.

```
# имплементация нашей модели при задействовании
# общего нелинейного признакового преобразования
def model(x, theta):

    # признаковое преобразование
    f = feature_transforms(x, theta[0])

    # вычислить линейную комбинацию и вернуться
    a = theta[1][0] + np.dot(f.T, theta[1][1:])

    return a.T
```

Здесь наше обобщенное множество преобразований сконструированных признаков имплементировано в Python-функции `feature_transformations` и отличается в зависимости от того, как определены сами признаки. Мы имплементировали эту функцию как можно более обобщенно, чтобы охватить случай, когда наши желаемые признаковые преобразования имеют внутренние параметры. То есть мы упаковываем модельные веса в множество Θ как список `theta`, содержащий *внутренние веса* функции `feature_transforms` в его первой записи `theta[0]`, и веса в финальной линейной комбинации модели, хранящейся во второй записи `theta[1]`.

Например, функция `feature_transforms`, задействуемая в примере 10.1, может быть имплементироавана следующим образом:

```
def feature_transforms(x, w):

    # вычислить признаковое преобразование
    f = np.sin(w[0] + np.dot(x.T, w[1:])).T

    return f
```

Если наши желаемые признаковые преобразования не имеют внутренних параметров, то мы можем оставить пустым параметрический вход в эту функцию либо имплементировать приведенную выше модель немного по-другому, вычислив наше множество признаковых преобразований как

```
f = feature_transforms(x)
```

и вычислив линейную комбинацию этих преобразованных признаков как

```
a = theta[0] + np.dot(f.T, theta[1:])
```

В любом случае для того чтобы успешно выполнить нелинейную регрессию, мы можем сосредоточить наше внимание исключительно на имплементировании функции `feature_transforms`, задействовав обернутую в autograd библиотеку NumPy, если мы хотим применить автоматическое дифференцирование (*см. разд. П2.10*). Ничто в том, как мы имплементируем наши *регрессионные стоимостные функции*, не отходит от первоначального контекста линейной регрессии, подробно описанного в *главе 5*. Другими словами, после того как мы правильно имплементировали данное множество признаковых преобразований, задействовав приведенную выше модель model, мы можем настроить параметры нашей нелинейной регрессии точно так же, как мы сделали это в *главе 5*, используя любую регрессионную стоимостную функцию и схему локальной оптимизации. Единственное, что следует иметь в виду, что при задействовании *параметризованных* моделей (например, в примере 10.1) соответствующие стоимостные функции обычно не являются *выпуклыми*. Отсюда необходимо применять методы оптимизации либо нулевого, либо первого порядка, либо методы оптимизации второго порядка, скорректированные в манере, подробно описанной в *разд. П1.7*.

10.3. Нелинейная многовыходная регрессия

В этом разделе мы приведем описание инженерии нелинейных признаков для многовыходной регрессии, впервые представленной в разд. 5.6. Оно полностью отражает рассуждения из предыдущего раздела с одним небольшим, но важным отличием: в многовыходном случае мы можем моделировать каждую регрессию *отдельно*, задействуя (потенциально) разную нелинейную модель для каждого выхода, или *совместно*, создавая единую нелинейную модель для всех выходов одновременно.

10.3.1. Принципы моделирования

В случае линейной многовыходной регрессии мы строим C линейных моделей формы $\dot{\mathbf{x}}^T\mathbf{w}_c$, или, что эквивалентно, одну совместную линейную модель, вклю-

чающую все C регрессий, укладывая весовые векторы $\mathbf{w}_c$ по столбцам в $(N+1)\times C$-матрицу $\mathbf{W}$ (*см. разд. 5.6.1*) и формируя многовыходную линейную модель

$$\text{model}(\mathbf{x}, \mathbf{W}) = \dot{\mathbf{x}}^T\mathbf{W}. \tag{10.15}$$

Имея набор данных из P точек $\left\{\left(\mathbf{x}_p, \mathbf{y}_p\right)\right\}_{p=1}^{P}$, где каждый парный вход $\mathbf{x}_p$ и выход $\mathbf{y}_p$ имеют размеры соответственно $N\times 1$ и $1\times C$, мы стремимся настроить параметры в $\mathbf{W}$, чтобы усвоить линейную зависимость между входом и выходом как

$$\dot{\mathbf{x}}_p^T\mathbf{W} \approx \mathbf{y}_p, \qquad p = 1, 2, ..., P \tag{10.16}$$

путем минимизирования соответствующей стоимостной функции, например стоимости наименьших квадратов

$$g(\mathbf{W}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\dot{\mathbf{x}}_p^T\mathbf{W} - \mathbf{y}_p\right\|_2^2. \tag{10.17}$$

С многовыходной регрессией переход от линейного к нелинейному моделированию близко отражает то, что мы видели в предыдущем разделе. То есть для c-й регрессионной задачи мы строим модель, используя (в общем случае) B_c нелинейных признаковых преобразований как

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = w_{c,0} + f_{c,1}(\mathbf{x})w_{c,1} + f_{c,2}(\mathbf{x})w_{c,2} + ... + f_{c,B_c}(\mathbf{x})w_{c,B_c}, \tag{10.18}$$

где $f_{c,1}$, $f_{c,2}$, ..., f_{c,B_c} — нелинейные (потенциально параметризованные) функции, а все w с $w_{c,0}$ по w_{c,B_c} (наряду с любыми дополнительными весами, внутренними для нелинейных функций) представлены в весовом множестве Θ_c.

Для того чтобы упростить изнурительную работу по выбору нелинейных признаков для каждого регрессора, мы можем выбрать одно множество нелинейных признаковых преобразований и использовать их *совместно* между всеми C регрессионными моделями. Если мы выберем одинаковое множество из B нелинейных признаков для всех C моделей, то c-я модель принимает форму

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = w_{c,0} + f_1(\mathbf{x})w_{c,1} + f_2(\mathbf{x})w_{c,2} + ... + f_B(\mathbf{x})w_{c,B}, \tag{10.19}$$

где Θ_c теперь содержит линейно-комбинационные веса $w_{c,0}$, $w_{c,1}$, ..., $w_{c,B}$ и любые веса, внутренние для совместных признаковых преобразований. Обратите внимание, что в этом случае единственными параметрами, уникальными для c-й модели, являются линейно-комбинационные веса, поскольку каждая модель совместно использует любые веса, внутренние для признаковых преобразований. Задействовав ту же самую компактную нотацию для наших признаковых преобразований, что и в уравнении (10.9), мы можем выразить каждую из этих моделей компактнее:

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = \dot{\mathbf{f}}^T\mathbf{w}_c. \tag{10.20}$$

На рис. 10.6 показана прототипическая многовыходная регрессия с использованием этой нотации. Мы можем выразить все C моделей вместе, уложив все C весовых векторов $\mathbf{w}_c$ по столбцам в весовую $(B+1)\times C$-матрицу $\mathbf{W}$, получив совместную модель в виде

$$\text{model}(\mathbf{x}, \Theta) = \dot{\mathbf{f}}^T \mathbf{W}. \tag{10.21}$$

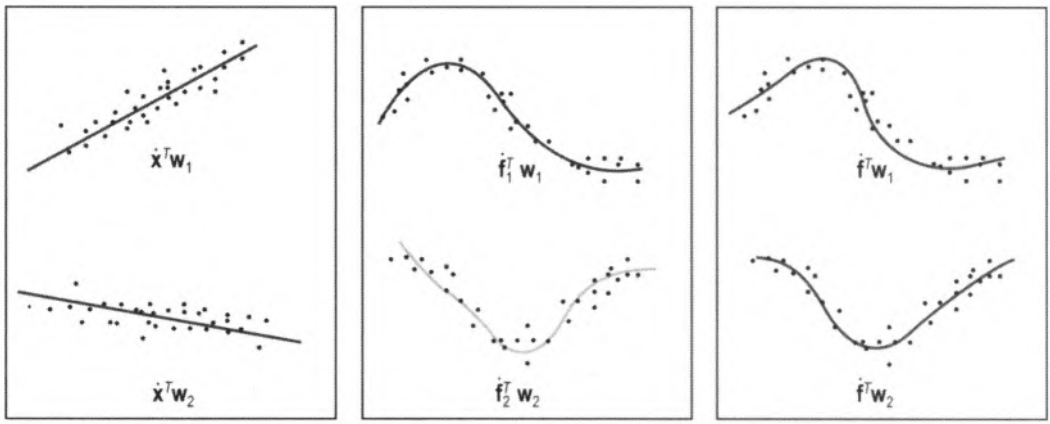


Рис. 10.6. Фигуральные иллюстрации многовыходной регрессии с $C = 2$ выходами. Линейная многовыходная регрессия (*слева*). Нелинейная многовыходная регрессия, где каждый регрессор использует собственные особые нелинейные признаковые преобразования (*в центре*). Нелинейная многовыходная регрессия, в которой оба регрессора совместно используют одинаковые нелинейные признаковые преобразования (*справа*)

Это является прямым обобщением исходной линейной модели, показанной в уравнении (10.15), и множество Θ содержит линейно-комбинационные веса в $\mathbf{W}$, а также любые параметры, внутренние для самих наших признаковых преобразований. Для того чтобы настроить веса нашей совместной модели так, чтобы она представляла наш набор данных как можно лучше, или, если сформулировать алгебраически, так, чтобы мы имели

$$\dot{\mathbf{f}}_p^T \mathbf{W} \approx \mathbf{y}_p, \qquad p = 1, 2, ..., P, \tag{10.22}$$

мы минимизируем надлежащую регрессионную стоимость этой модели над параметрами в Θ, например стоимостью наименьших квадратов[4].

$$g(\Theta) = \frac{1}{P}\sum_{p=1}^{P} \left\| \dot{\mathbf{f}}_p^T \mathbf{W} - \mathbf{y}_p \right\|_2^2. \tag{10.23}$$

[4] Обратите внимание, что если эти признаковые преобразования не содержат никаких внутренних параметров (например, полиномиальных функций), то каждую отдельную регрессионную модель можно настроить отдельно. Однако при задействовании *параметризованных* признаков (например, нейронных сетей) стоимостная функция не разлагается по каждому регрессору, и мы должны настраивать все параметры нашей модели совместно, т. е. мы должны обучать все C регрессий *одновременно*. Это отличается от линейного случая, где настраивание параметров линейной модели (одного регрессора за раз либо одновременно всех) возвращает одинаковый результат (*см. разд. 5.6.2*).

10.3.2. Инженерия признаков

В случае многовыходной регрессии задача определения надлежащих признаков с помощью визуального анализа является более сложной, чем базовый экземпляр регрессии, подробно описанный в предыдущем разделе. Приведем один относительно простой пример такого рода инженерии признаков, чтобы дать поверхностное представление об этой трудности и о связанном с ней нелинейном моделировании.

Пример 10.3. Моделирование волн

На рис. 10.7 показан пример нелинейной многовыходной регрессии с использованием игрушечного набора данных с входной размерностью $N = 2$ и выходной размерностью $C = 2$, где вход в паре с первым и вторым выходами показаны соответственно слева и справа. Оба экземпляра, по всей видимости, имеют *синусоидальную* природу, причем каждый из них имеет собственный уникальный контур.



Рис. 10.7. Иллюстрация примера 10.3. Сравните с линейным случаем на рис. 5.11

Исходя из визуального анализа данных, мы можем обоснованно решить моделировать обе регрессии одновременно, используя $B = 2$ параметризованных синусоидальных признаковых преобразований:

$$\begin{aligned} f_1(x) &= \sin\left(v_{1,0} + v_{1,1}x_1 + v_{1,2}x_2\right); \\ f_2(x) &= \sin\left(v_{2,0} + v_{2,1}x_1 + v_{2,2}x_2\right). \end{aligned} \tag{10.24}$$

Совместная подгонка этого множества нелинейных признаков путем минимизирования стоимости наименьших квадратов из уравнения (10.23) с использованием градиентного спуска приводит к нелинейным поверхностям, графики которых построены на рис. 10.7 зеленым цветом.

10.3.3. Имплементация на Python

Как и в линейном случае, подробно описанном в *разд. 5.6.3*, здесь мы также можем вернуться к нашей общей Python-имплементации нелинейной регрессии, введенной в *разд. 10.2.3*, и задействовать точно такую же модель и имплементацию стоимостной функции, как и в одновыходном варианте. Единственное различие заключается в том, как мы определяем наши признаковые преобразования и размеры нашей матрицы линейно-комбинационных весов.

10.4. Нелинейная двухклассовая классификация

В этом разделе мы введем общий каркас нелинейной классификации, а также ряд элементарных примеров. Как и в предыдущих разделах, все эти примеры являются низкоразмерными, что позволяет нам визуально выискивать шаблоны в данных и предлагать соответствующие нелинейности, которые мы можем внести в нашу линейную контролируемую парадигму для порождения нелинейных классификаций. При этом мы, по существу, выполняем инженерию нелинейных признаков для задачи двухклассовой классификации.

10.4.1. Принципы моделирования

Когда мы задействовали линейную модель в выведении линейной двухклассовой классификации в *главе 6*, эта линейность была просто *допущением* о своего рода границе, которая (в основном) разделяет два класса данных. После того как мы задействовали по умолчанию значения меток $y_p \in \{-1, +1\}$ и выразили нашу линейную модель алгебраически как

$$\text{model}(\mathbf{x}, \mathbf{w}) = \dot{\mathbf{x}}^T\mathbf{w}, \tag{10.25}$$

наша линейная граница решения теперь состоит из всех входных точек $\mathbf{x}$, где $\dot{\mathbf{x}}^T\mathbf{w} = 0$. Схожим образом предсказания меток выглядят (*см. разд. 7.6*) как

$$y = \text{sign}\left(\dot{\mathbf{x}}^T\mathbf{W}\right). \tag{10.26}$$

Для того чтобы настроить $\mathbf{w}$, мы минимизируем надлежащую стоимостную функцию двухклассовой классификации, например стоимость двухклассового мягкого максимума (или перекрестной энтропии):

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}\right). \tag{10.27}$$

Мы можем настроить этот каркас для прыжка от линейной классификации к нелинейной совершенно аналогично тому, как мы сделали это с регрессией в *разд. 10.2*. То есть вместо нашей линейной модели мы можем подставить нелинейную обобщенную форму

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + \ldots + f_B(\mathbf{x})w_B, \tag{10.28}$$

где f_1, f_2, ..., f_B — нелинейные параметризованные или непараметризованные функции, и все w с w_0 по w_B (наряду с любыми дополнительными весами, внутренними для нелинейных функций) представлены в весовом множестве Θ. Как и в случае с регрессией, здесь мы также можем выразить это компактнее (*см. разд. 10.2.1*):

$$\text{model}(\mathbf{x}, \Theta) = \dot{\mathbf{f}}^T\mathbf{w}. \tag{10.29}$$

В полной аналогии с линейным случаем наша граница решения здесь состоит из всех входов $\mathbf{x}$, где $\dot{\mathbf{f}}^T\mathbf{w} = 0$, и схожим образом предсказания делаются как

$$y = \text{sign}\left(\dot{\mathbf{f}}^T\mathbf{w}\right). \tag{10.30}$$

На рис. 10.8 показана прототипическая двухклассовая классификация с использованием этой нотации.

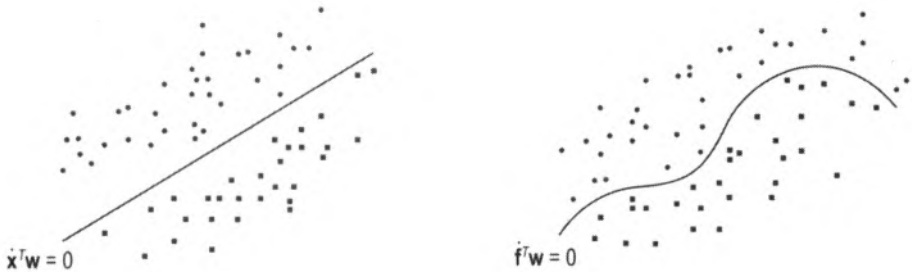


Рис. 10.8. Фигуральные иллюстрации линейной и нелинейной двухклассовой классификации. *В* линейном случае разделяющая граница определяется как $\dot{\mathbf{x}}^T\mathbf{w} = 0$ (*слева*). В нелинейном случае разделяющая граница определяется как $\dot{\mathbf{f}}^T\mathbf{w} = 0$ (*справа*)

Наконец, для того, чтобы настроить параметры в Θ, мы должны минимизировать надлежащую стоимостную функцию по отношению к ней, например стоимость двухклассового softmax (опять же в полной аналогии с уравнением (10.27)):

$$g(\Theta) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\dot{\mathbf{f}}_p^T\mathbf{w}}\right). \tag{10.31}$$

10.4.2. Инженерия признаков

С низкоразмерными наборами данных мы можем в некоторых случаях довольно легко вырабатывать надлежащее множество нелинейных признаков для двухклассовой классификации путем рассмотрения данных визуально. Далее мы изучим два таких примера.

Пример 10.4. Когда граница решения состоит всего из двух отдельных точек

Обсуждая классификацию через призму логистической регрессии в *разд. 6.3*, мы увидели, что линейную классификацию можно понимать как специфический пример нелинейной регрессии. В частности, мы видели, как с этой точки зрения мы стремимся подогнать кривую (или поверхность в более высоких размерностях), которая состоит из линейной комбинации нашего входа, пропущенного через гиперболический тангенс tanh. Для наборов данных с входной размерностью $N = 1$, как показано на рис. 10.9, *слева*, это приводит к получению границы решения, которая определяется одной-*единственной* точкой.

Однако линейная граница решения в общем случае является достаточно негибкой и не обеспечивает хорошего разделения даже в случае простого примера, показанно-

го на рис. 10.9, *справа*. Для такого набора данных нам явно нужна модель, способная пересекать входное пространство (ось x) *дважды* в точках, разделенных некоторым расстоянием, чего линейная модель никогда не сможет сделать.

Какого рода простая функция пересекает горизонтальную ось дважды? Квадратичная функция может. Если приспособить квадратичную функцию к нужной высоте, то она, конечно же, сможет пересечь горизонтальную ось дважды и, будучи пропущенной через функцию tanh, действительно даст нам желаемые предсказания (как показано на рис. 10.9, *справа*).

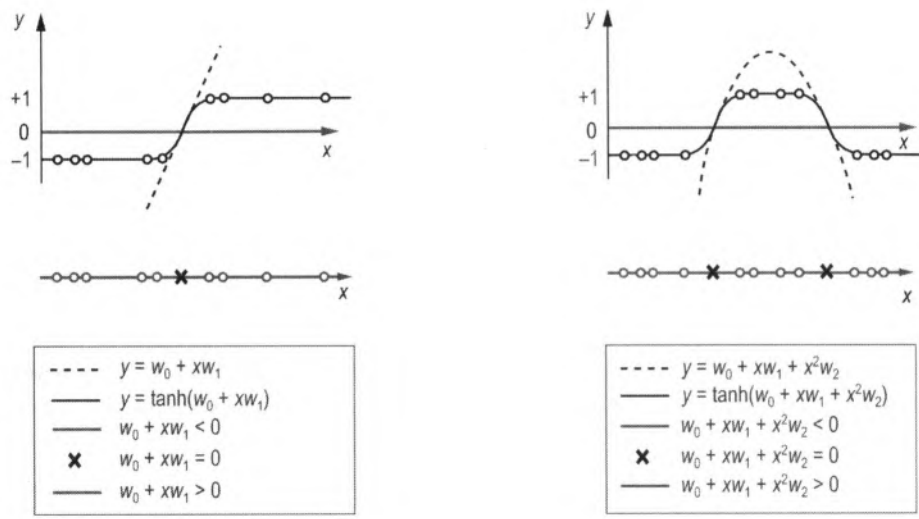


Рис. 10.9. Иллюстрация примера 10.4. Прототипический линейный двухклассовый классификационный набор данных с полностью настроенной линейной моделью, показанной с точки зрения регрессии (*слева сверху*) и с точки зрения перцептрона (*слева снизу*), где значения меток кодируются в виде цветов (красный для $+1$ и синий для -1). Простой нелинейный двухклассовый классификационный набор данных, который требует границы решения, состоящей из двух точек, чего линейная модель не может обеспечить (*справа*). Как видно здесь, квадратичная модель способна достичь этой цели (при условии, что ее параметры настроены надлежаще)

Квадратичная модель принимает форму

$$\text{model}(\mathbf{x}, \Theta) = w_0 + xw_1 + x^2 w_2, \tag{10.32}$$

в которой используются два признаковых преобразования: тождественное преобразование $f_1(x) = x$ и квадратичное $f_2(x) = x^2$, причем весовое множество Θ содержит только веса w_0, w_1 и w_2.

На рис. 10.10, *слева* представлен синтетический набор данных, подобный тому, что показан на рис. 10.9, *справа*. Зеленым цветом изображен результат полной настройки квадратичной модели из уравнения (10.32) путем минимизирования (посредством градиентного спуска) соответствующего двухклассового мягкого максимума из

уравнения (10.31). В правой части рисунка показан тот же самый набор данных, только в *пространстве преобразованных признаков*, определяемом нашими двумя признаками (как впервые подробно описано в примерах 10.1 и 10.2), где нелинейная граница решения становится линейной. Эта находка верна в общем случае: хорошо разделяющая *нелинейная* граница решения в исходном пространстве набора данных преобразуется в хорошо разделяющую *линейную* границу решения в пространстве преобразованных признаков. Это аналогично случаю регрессии, описанному в *разд. 10.2.2*, где хорошая нелинейная подгонка в исходном пространстве соответствует хорошей линейной подгонке в пространстве преобразованных признаков.

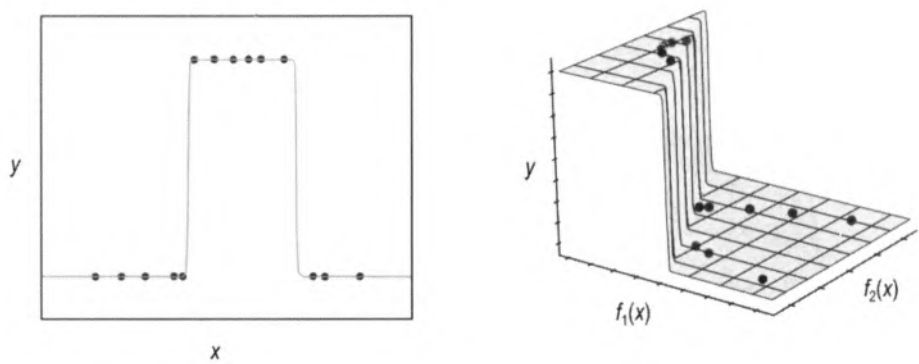


Рис. 10.10. Иллюстрация примера 10.4

Пример 10.5. Эллиптическая граница решения

На рис. 10.11, *слева* представлен моделированный двухклассовый классификационный набор данных с входной размерностью $N = 2$, показанный с точки зрения перцептрона в верхней части рисунка и с точки зрения регрессии в нижней части.

Визуально исследовав набор данных, делаем вывод, что своего рода эллиптическая граница решения, центрированная в начале координат, определяемая формулой

$$\text{model}(\mathbf{x}, \Theta) = w_0 + x_1^2 w_1 + x_2^2 w_2, \tag{10.33}$$

возможно, отлично справится с классификацией. Разобрав эту формулу, мы видим, что мы использовали два признаковых преобразования, т. е. $f_1(x) = x_1^2$ и $f_2(x) = x_2^2$ с параметрическим множеством $\Theta = \{w_0, w_1, w_2\}$.

Минимизировав стоимость мягкого максимума в уравнении (10.31) с помощью этой модели (посредством градиентного спуска), мы показываем результирующую нелинейную границу решения черным цветом с точки зрения перцептрона в верхней части рисунка и с точки зрения регрессии в нижней части. Наконец, на рисунках справа представлены данные в *пространстве преобразованных признаков* с соответствующей линейной границей решения.

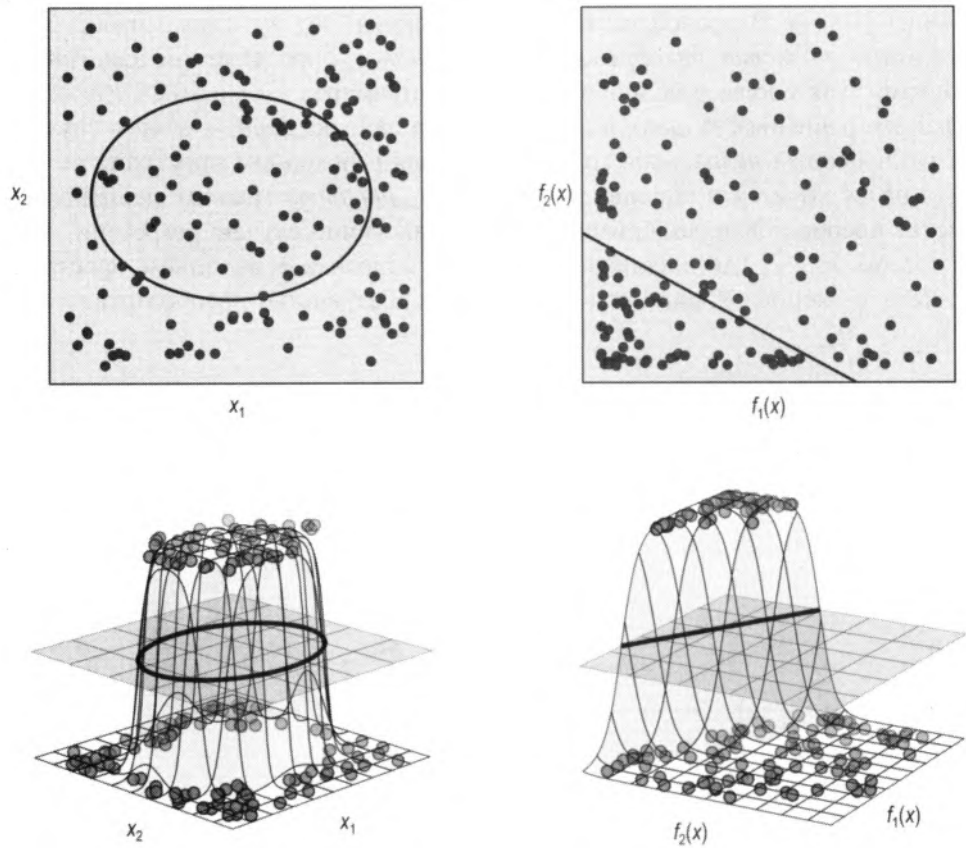


Рис. 10.11. Иллюстрация примера 10.5

10.4.3. Имплементация на Python

Общая нелинейная модель в уравнении (10.29) может быть имплементирована точно так же, как описано в *разд. 10.2.3*, поскольку это та же самая общая нелинейная модель, которую мы используем с нелинейной регрессией. Следовательно, как и в случае с регрессией, нам не нужно изменять имплементацию любой двухклассовой классификационной стоимостной функции, представленной в *главе 6*, чтобы выполнить нелинейную классификацию: от нас требуется лишь надлежаще определить наши нелинейные преобразования на Python.

10.5. Нелинейная многоклассовая классификация

В этом разделе мы представим общее нелинейное расширение линейной многоклассовой классификации, впервые введенной в *главе 7*. Оно очень точно отражает то, что мы видели в предыдущих разделах, и, в частности, почти полностью похоже на обсуждение нелинейной многовыходной регрессии в *разд. 10.3*.

10.5.1. Принципы моделирования

Как мы видели в *главе 7* в случае линейной многоклассовой классификации, мы строим C линейных моделей формы $\dot{\mathbf{x}}^T\mathbf{w}_c$, которые можно представить совместно, уложив весовые векторы $\mathbf{w}_c$ по столбцам в $(N+1)\times C$-матрицу $\mathbf{W}$ (*см. разд. 7.3.9*) и сформировав единую многовыходную линейную модель формы

$$\text{model}(\mathbf{x}, \mathbf{W}) = \dot{\mathbf{x}}^T\mathbf{W}. \tag{10.34}$$

Имея набор данных из P точек $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$, где каждый вход $\mathbf{x}_p$ является N-мерным и каждый y_p является значением метки в множестве $y_p \in \{0, 1, ..., C-1\}$, мы стремимся настроить параметры в $\mathbf{W}$, чтобы удовлетворить правило слияния

$$y_p = \underset{c=0,1,...,C-1}{\arg\max}\left[\text{model}\left(\mathbf{x}_p, \mathbf{W}\right)\right], \qquad p = 1, 2, ..., P \tag{10.35}$$

путем настраивания каждого столбца $\mathbf{W}$ по одному за раз по принципу "один против всех" (*см. разд. 7.2*) либо минимизирования соответствующей стоимости, например стоимости многоклассового softmax

$$g(\mathbf{W}) = \frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{c=0}^{C-1} e^{\dot{\mathbf{x}}_p^T\mathbf{w}_c}\right) - \dot{\mathbf{x}}_p^T\mathbf{w}_{y_p}\right] \tag{10.36}$$

над всей матрицей $\mathbf{W}$ одновременно (*см. разд. 7.3*).

В случае многоклассовой классификации переход от линейного моделирования к нелинейному очень точно отражает то, что мы видели в случае многовыходной регрессии в *разд. 10.3*. То есть для c-го классификатора мы можем построить модель, используя (в общем случае) B_c нелинейных признаковых преобразований:

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = w_{c,0} + f_{c,1}(\mathbf{x})w_{c,1} + f_{c,2}(\mathbf{x})w_{c,2} + ... + f_{c,B_c}(\mathbf{x})w_{c,B_c}, \tag{10.37}$$

где $f_{c,1}$, $f_{c,2}$, ..., f_{c,B_c} — нелинейные параметризованные или непараметризованные функции, и все w с $w_{c,0}$ по w_{c,B_c} (наряду с любыми дополнительными весами, внутренними для нелинейных функций) представлены в весовом множестве Θ_c.

Для того чтобы упростить изнурительную работу по выбору нелинейных признаков для каждого классификатора, мы можем вместо этого выбрать *одно* множество нелинейных признаковых преобразований и использовать их *совместно* между всеми двухклассовыми моделями. Если мы выберем одинаковое множество B нелинейных признаков для всех C моделей, то c-я модель примет форму

$$\text{model}_c\left(\mathbf{x}, \Theta_c\right) = w_{c,0} + f_1(\mathbf{x})w_{c,1} + f_2(\mathbf{x})w_{c,2} + ... + f_B(\mathbf{x})w_{c,B}, \tag{10.38}$$

где Θ_c теперь содержит линейно-комбинационные веса $w_{c,0}$, $w_{c,1}$, ..., $w_{c,B}$, а также любые веса, внутренние для совместных признаковых преобразований. Задейство-

вав ту же самую компактную нотацию для наших признаковых преобразований, что и в уравнении (10.9), мы можем выразить каждую из этих моделей компактнее:

$$\text{model}_c(\mathbf{x}, \Theta_c) = \dot{\mathbf{f}}^T \mathbf{W}_c . \tag{10.39}$$

На рис. 10.12 показана прототипическая многоклассовая классификация с использованием рассмотренной нотации. Мы можем настроить параметры каждой модели индивидуально, применяя подход "один против всех", или одновременно, минимизируя единую совместную стоимостную функцию над всеми вместе. Для выполнения последнего подхода полезно сначала повторно выразить все C моделей вместе, уложив все C весовых векторов $\mathbf{w}_c$ по столбцам в весовую $(B+1)\times C$-матрицу $\mathbf{W}$, получив совместную модель в виде

$$\text{model}(\mathbf{x}, \Theta) = \dot{\mathbf{f}}^T \mathbf{W}, \tag{10.40}$$

где множество Θ содержит линейно-комбинационные веса в $\mathbf{W}$, а также любые параметры, внутренние для самих наших признакововых преобразований. Для того чтобы настроить веса нашей совместной модели так, чтобы правило слияния

$$y_p = \underset{c=0,1,\ldots,C-1}{\arg\max} \left[\text{model}\left(\mathbf{f}_p, \mathbf{W}\right)\right], \qquad p = 1, 2, \ldots, P \tag{10.41}$$

соблюдалось как можно лучше, мы минимизируем соответствующую многоклассовую стоимость этой модели над параметрами в Θ, т. е. стоимость многоклассового softmax

$$g(\Theta) = \frac{1}{P}\sum_{p=1}^{P}\left[\log\left(\sum_{c=0}^{C-1} e^{\dot{\mathbf{f}}_p^T \mathbf{w}_c}\right) - \dot{\mathbf{f}}_p^T \mathbf{w}_{y_p}\right]. \tag{10.42}$$

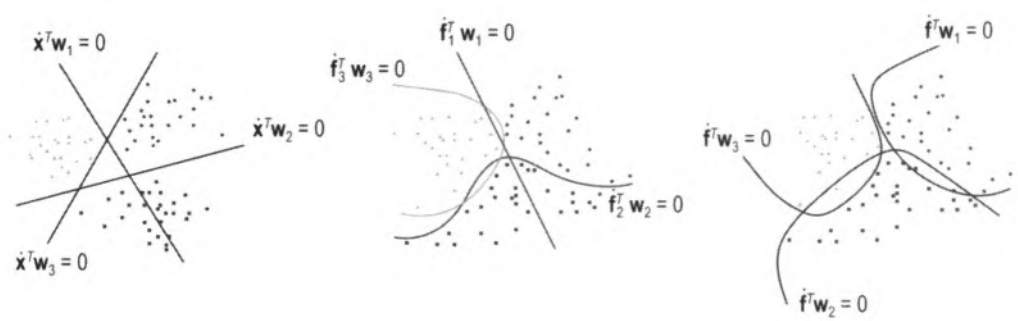


Рис. 10.12. Иллюстрации многоклассовых классификаторов на обобщенном наборе данных с $C = 3$ классами. Линейная многоклассовая классификация (*слева*). Нелинейная многоклассовая классификация, где каждый классификатор использует собственные особые нелинейные признаковые преобразования (*в центре*). Нелинейная многоклассовая классификация, в которой все классификаторы используют одинаковые нелинейные признаковые преобразования (*справа*)

10.5.2. Инженерия признаков

Задача определения нелинейных признаков посредством визуального анализа является еще более сложной в многоклассовой окружающей обстановке, чем в случае

двухклассовой классификации, подробно описанной в предыдущем разделе. Приведем простой пример такого рода инженерии признаков, отмечая, что в общем случае мы хотим *усваивать* такие признаковые преобразования автоматически (подробности *см. в следующей главе*).

Пример 10.6. Многоклассовые данные с эллиптическими границами

В этом примере мы конструируем нелинейные признаки для выполнения многоклассовой классификации на наборе данных, показанном на рис. 10.13, *слева сверху*, состоящей из $C = 3$ классов, которые, судя по всему, (грубо) разделяемы эллиптическими границами. Здесь точки, окрашенные в синий, красный и зеленый цвета, имеют значения меток соответственно 0, 1 и 2.

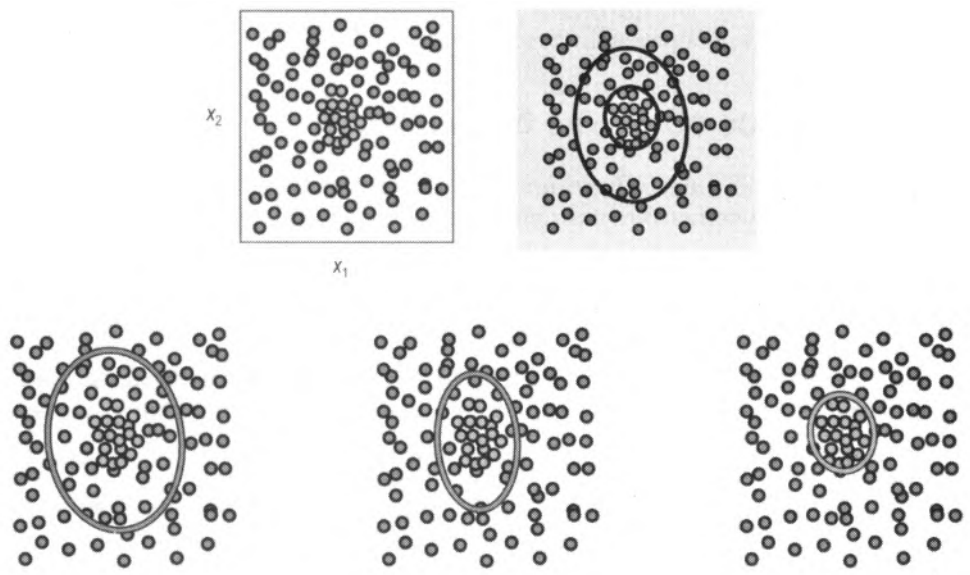


Рис. 10.13. Иллюстрация примера 10.6

Поскольку данные не центрированы в начале координат, мы должны использовать полное разложение входа, состоящего из признаков формы $x_1^i x_2^j$, где $i + j \leq 2$, по многочлену второй степени. Это дает модель на основе многочлена второй степени:

$$\text{model}(\mathbf{x}, \Theta) = w_0 + x_1 w_1 + x_2 w_2 + x_1 x_2 w_3 + x_1^2 w_4 + x_2^2 w_5 . \qquad (10.43)$$

Используя эту нелинейную модель, мы минимизируем стоимость многоклассового softmax в уравнении (10.42) посредством градиентного спуска и строим график соответствующей объединенной многоклассовой границы на рис. 10.13, *справа сверху*, где каждый участок окрашен в соответствии с предсказанием, сделанным окончательным классификатором.

Мы также строим график результирующей двухклассовой границы, полученной каждым отдельным классификатором в нижнем ряду рис. 10.13, окрашивая каждую

границу в соответствии с классифицированием "один против всех", выполненным в каждом случае. Здесь, как и в линейном случае, описанном в *разд. 7.3*, мы видим, что, хотя каждая двухклассовая подзадача не может быть решена правильно, при объединении посредством уравнения (10.41) результирующая многоклассовая классификация по-прежнему может быть очень хорошей.

10.5.3. Имплементация на Python

Общая нелинейная модель в уравнении (10.40) может быть имплементирована так, как описано в *разд. 10.3.3*, поскольку это та же самая общая нелинейная модель, которую мы используем с нелинейной многовыходной регрессией. Следовательно, как и в случае многовыходной регрессии, нам не нужно изменять имплементацию совместных нелинейных стоимостных функций многоклассовой классификации, введенных в *главе 7*: от нас требуется лишь надлежаще определить наше нелинейное преобразование (преобразования) на Python.

10.6. Нелинейное обучение без учителя

В этом разделе мы обсудим общее нелинейное расширение нашей фундаментальной методологии обучения без учителя, представленной в *разд. 8.3*: автокодировщик.

10.6.1. Принципы моделирования

В *разд. 8.3* мы описали линейный автокодировщик — элегантный способ определения оптимального линейного подпространства для представления множества среднецентрированных N-мерных входных точек данных $\left\{\mathbf{x}_p\right\}_{p=1}^{P}$. Для того чтобы определить проекцию наших данных на K-мерное подпространство, охватываемое столбцами $N \times K$-матрицы $\mathbf{C}$, мы сначала *кодируем* наши данные на подпространстве с помощью кодировочной модели

$$\text{model}_e(\mathbf{x}, \mathbf{C}) = \mathbf{C}^T\mathbf{x}, \tag{10.44}$$

которая принимает N-мерный вход $\mathbf{x}$ и возвращает K-мерный выход $\mathbf{C}^T\mathbf{x}$. Затем мы *декодируем*, чтобы получить проекцию на подпространство как

$$\text{model}_d(\mathbf{v}, \mathbf{C}) = \mathbf{C}\mathbf{v}. \tag{10.45}$$

Выход из декодировщика, будучи проекцией на подпространство, лежащее в исходном N-мерном пространстве, сам является N-мерным.

Композиция этих двух шагов дает линейную автокодировочную модель

$$\text{model}(\mathbf{x}, \mathbf{C}) = \text{model}_d\left(\text{model}_e(\mathbf{x}, \mathbf{C}), \mathbf{C}\right) = \mathbf{C}\mathbf{C}^T\mathbf{x}, \tag{10.46}$$

которая при правильной настройке $\mathbf{C}$ создает линейное подпространство, которое чрезвычайно хорошо представляет данные

$$\mathbf{C}\mathbf{C}^T\mathbf{x}_p \approx \mathbf{x}_p, \qquad p = 1, 2, ..., P \tag{10.47}$$

или, что эквивалентно, оказывает влияние на вход максимально близко к тождественному преобразованию, т. е.

$$\mathbf{C}\mathbf{C}^{T} \approx \mathbf{I}_{N\times N}. \tag{10.48}$$

Для того чтобы восстановить эту идеальную настройку для $\mathbf{C}$, мы можем минимизировать, например, величину ошибки желаемого эффекта, измеряемую методом наименьших квадратов, в уравнении (10.47):

$$g(\mathbf{C}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\mathbf{C}^{T}\mathbf{x}_p - \mathbf{x}_p\right\|_2^2. \tag{10.49}$$

Для того чтобы ввести здесь нелинейность, т. е. определить нелинейную поверхность (также именуемую *топологическим многообразием*) для проецирования данных (как показано на рис. 10.14, *справа*), мы можем просто заменить линейные кодировочные/декодировочные модели в уравнениях (10.44) и (10.45) нелинейными версиями обобщенной формы:

$$\begin{aligned}\text{model}_e(\mathbf{x}, \Theta_e) &= \mathbf{f}_e(\mathbf{x});\\ \text{model}_d(\mathbf{v}, \Theta_d) &= \mathbf{f}_d(\mathbf{v}).\end{aligned} \tag{10.50}$$

Здесь $\mathbf{f}_e$ и $\mathbf{f}_d$ — это (в общем случае) нелинейные векторные функции, причем Θ_e и Θ_d обозначают их параметрические множества. С помощью этой нотации мы можем записать общую нелинейную автокодировочную модель просто как

$$\text{model}(\mathbf{x}, \Theta) = \mathbf{f}_d(\mathbf{f}_e(\mathbf{x})), \tag{10.51}$$

где параметрическое множество Θ теперь содержит все параметры как Θ_e, так и Θ_d.

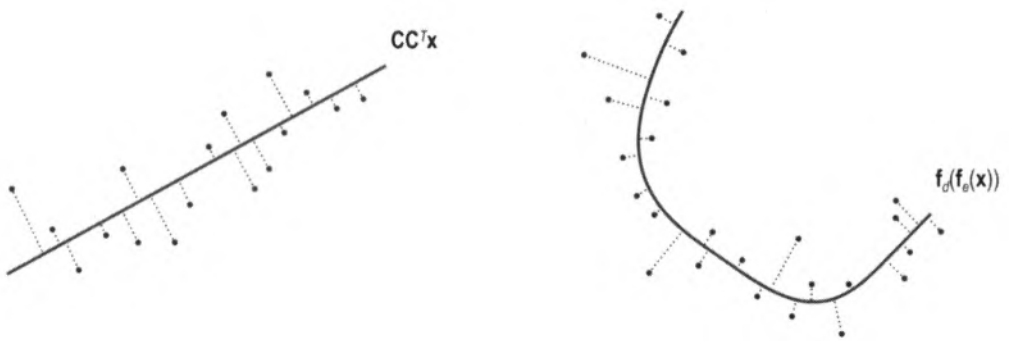


Рис. 10.14. Иллюстрации линейного (*слева*) и нелинейного (*справа*) автокодировщиков

Как и в линейной версии, здесь наша цель состоит в том, чтобы надлежаще сконструировать пару кодировщик/декодировщик и надлежаще настроить параметры Θ, чтобы определить соответствующее нелинейное многообразие для данных как

$$\mathbf{f}_d(\mathbf{f}_e(\mathbf{x}_p)) \approx \mathbf{x}_p, \qquad p = 1, 2, ..., P. \tag{10.52}$$

Для настройки параметров в Θ мы можем минимизировать, например, величину ошибки желаемого эффекта, измеряемую методом наименьших квадратов в уравнении (10.52):

$$g(\Theta) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{f}_d\left(\mathbf{f}_e\left(\mathbf{x}_p\right)\right) - \mathbf{x}_p\right\|_2^2 . \tag{10.53}$$

10.6.2. Инженерия признаков

Учитывая, что и кодировочная, и декодировачная модели содержат нелинейные признаки, которые должны быть определены, и композиционный принцип, которым модель формируется в уравнении (10.51), конструирование признаков с помощью визуального анализа может быть затруднено даже на чрезвычайно простом примере, подобном тому, который мы сейчас обсудим.

Пример 10.7. Округлое топологическое многообразие

В этом примере мы используем симулированный набор из $P = 20$ двумерных точек данных для усвоения округлого топологического многообразия посредством нелинейной автокодировочной схемы, описанной в текущем разделе. Этот набор данных показан на рис. 10.15, *слева*, где мы видим, что он имеет почти совершенный округлый контур.

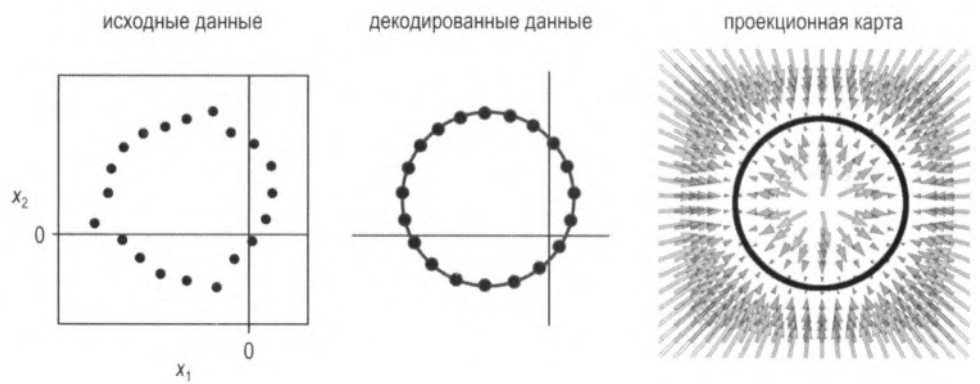


Рис. 10.15. Иллюстрация примера 10.7. Исходные данные распределены примерно по окружности (*слева*). Окончательное декодирование исходных данных на определенное округлое топологическое многообразие (*в центре*). Проекционная карта, показывающая, как точки в соседнем пространстве проецируются на окончательное усвоенное топологическое многообразие (*справа*)

Для того чтобы сконструировать нелинейную автокодировочную модель для этого набора данных, вспомним, что окружность в двумерном пространстве координат (рис. 10.16, *слева*) можно полностью охарактеризовать ее центральной точкой $\mathbf{w} = [w_1 \quad w_2]^T$ и радиусом r. Тогда вычитание $\mathbf{w}$ из любой точки $\mathbf{x}_p$ на окружности центрирует данные в начале координат (рис. 10.16, *справа*).

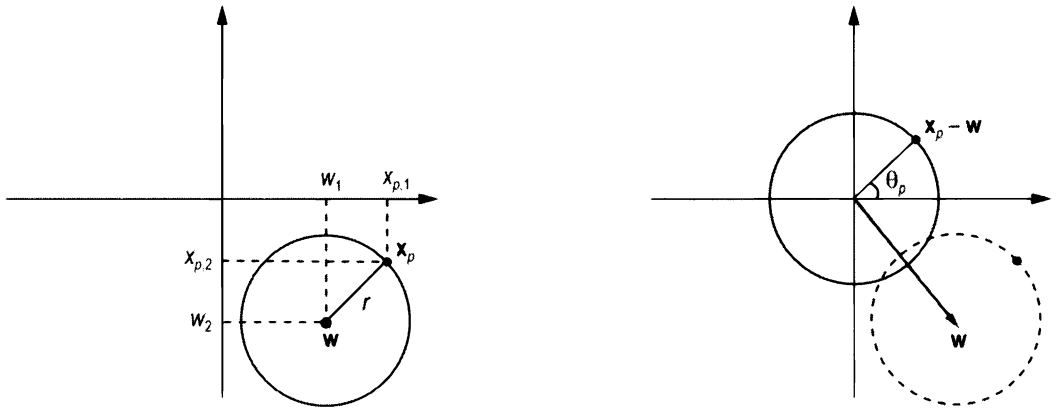


Рис. 10.16. Иллюстрация примера 10.7. Окружность в двумерном пространстве координат характеризуется своей центральной точкой **w** и своим радиусом r (*слева*). Когда она центрирована в начале координат, любая точка на окружности может быть представлена углом, созданным между ее соединителем с началом координат и горизонтальной осью (*справа*)

После центрирования любая двумерная точка $\mathbf{x}_p - \mathbf{w}$ на окружности может быть закодирована как (скалярный) угол θ_p между отрезком прямой, соединяющим ее с началом координат, и горизонтальной осью. Математически говоря, мы имеем

$$\mathbf{f}_e(\mathbf{x}_p) = \theta_p = \arctan\left(\frac{x_{p,2} - w_2}{x_{p,1} - w_1}\right). \tag{10.54}$$

Для того чтобы сконструировать декодировщик, начиная с θ_p, мы можем перестроить $\mathbf{x}_p$ как

$$\mathbf{f}_d(\theta_p) = \begin{bmatrix} r\cos(\theta_p) + w_1 \\ r\sin(\theta_p) + w_2 \end{bmatrix}. \tag{10.55}$$

Взятая вместе эта пара кодировщик/декодировщик определяет соответствующую нелинейную автокодировочную модель общей формы, приведенную в уравнении (10.51), с множеством параметров $\Theta = \{w_1, w_2, r\}$.

На рис. 10.15, *в центре* показано окончательное усвоенное топологическое многообразие вместе с декодированными данными, найденными путем минимизирования стоимостной функции из уравнения (10.53) посредством градиентного спуска. В правой части рисунка представлено восстановленное топологическое многообразие в виде черной окружности (с красным контуром для удобства визуализации); это иллюстрирует то, как точки в пространстве притягиваются (или *проецируются*) к восстановленному топологическому многообразию в виде векторного поля.

10.7. Заключение

В этой главе мы описали *инженерию нелинейных признаков* для задач обучения с учителем и без учителя. Инженерия нелинейных признаков предусматривает конструирование нелинейных моделей посредством философской рефлексии либо визуального анализа данных. Хотя инженерия нелинейных признаков сама по себе является очень полезным набором навыков, более значительная ценность этой главы заключается во введении общего каркаса нелинейного моделирования, включая общие нелинейные модели, формализации и концепции, которые лежат в фундаменте остальных глав книги.

10.8. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:

github.com/jermwatt/machine_learning_refined.

10.1. Моделирование волны

Повторить эксперимент, описанный в примере 10.1, включая построение графиков, показанных на рис. 10.2.

10.2. Моделирование роста численности популяции

На рис. 10.17 представлена популяцию дрожжевых клеток, растущих в ограниченной камере (данные взяты из работы [46]). Это общий контур, обнаруживаемый с данными о росте популяции, когда исследуемый организм начинает только с нескольких элементов и ограничен в росте тем, насколько быстро он может размножаться, и ресурсами, имеющимися в окружающей его среде. Вначале такая популяция растет экспоненциально, но быстро останавливается, когда достигает максимального объема своей среды обитания.

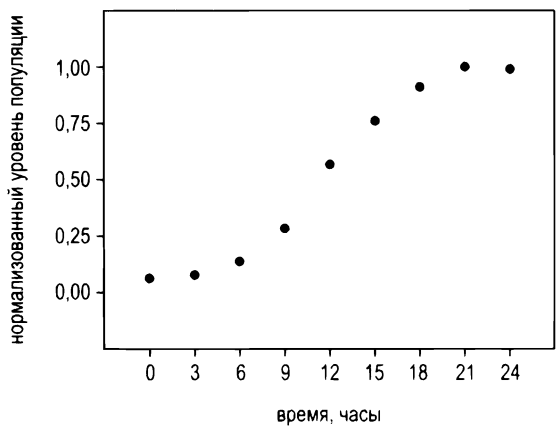


Рис. 10.17. Иллюстрация упражнения 10.2

Предложить *одно* нелинейное признаковое преобразование для этого набора данных и вписать в него соответствующую модель, используя стоимостную функцию наименьших квадратов и градиентный спуск. Позаботиться о том, чтобы данные на входе были *стандартно нормализованы* (*см. разд. 9.3*). Построить график данных вместе с окончательной подгонкой, обеспечиваемой вашей моделью, вместе на одном рисунке.

10.3. Эксперимент Галилея

Повторить эксперимент, описанный в примере 10.2, включая построение графиков, показанных на рис. 10.5.

10.4. Закон Мура

Гордон Мур, один из основателей корпорации Intel, предсказал в своей статье 1965 года [47], что число транзисторов на интегральной схеме будет удваиваться примерно каждые два года. Эта гипотеза, которую сегодня называют законом Мура, за последние пять десятилетий оказалась достаточно точной. Поскольку вычислительная мощность компьютеров напрямую связана с числом транзисторов в их процессорах, закон Мура обеспечивает трендовую модель для предсказания вычислительной мощности будущих микропроцессоров. На рис. 10.18 построен график числа транзисторов нескольких микропроцессоров в сопоставлении с годом их выпуска, начиная с Intel 4004 в 1971 году с 2300 транзисторами и заканчивая Intel Xeon E7, представленным в 2014 году, с более чем 4,3 млрд транзисторов.

А) Предложить одно признаковое преобразование для набора данных закона Мура, показанного на рис. 10.18, так, чтобы преобразованные входные/выходные данные были связаны линейно. Подсказка: для того чтобы произвести линейную взаимосвязь, вам придется преобразовать выход, а не вход.

Б) Сформулировать и минимизировать функцию наименьших квадратов для соответствующих весов и вписать свою модель в данные в исходном пространстве данных, как показано на рис. 10.18.

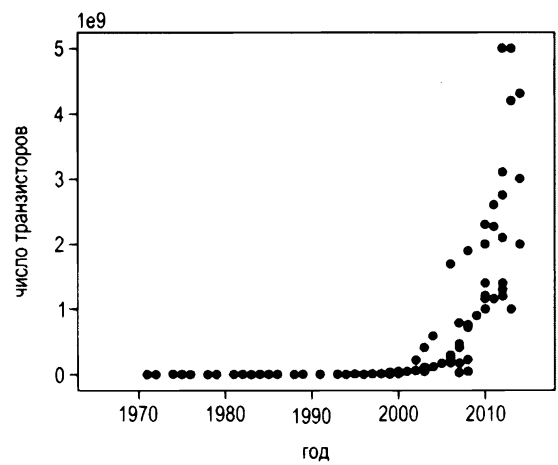


Рис. 10.18. Иллюстрация упражнения 10.4

10.5. Закон Ома

Закон Ома, предложенный немецким физиком Георгом Симоном Омом после серии экспериментов, проведенных им в 1820-х годах, связывает величину (магнитуду) тока в гальванической цепи с суммой всех возбуждающих сил в цепи, а также с длиной цепи. Хотя он не опубликовал никакого отчета о своих экспериментальных результатах, легко проверить его закон, используя простую экспериментальную установку, показанную на рис. 10.19, *слева*, которая очень похожа на ту, что он тогда использовал. Спиртовая лампа нагревает цепь, создавая электродвижущую силу, которая создает ток в катушке, отклоняющий стрелку компаса. Тангенс угла отклонения прямо пропорционален силе тока, проходящего через цепь. Сила тока, обозначаемая через I, изменяется в зависимости от длины провода, используемого для замыкания цепи (пунктирная кривая). На рис. 10.19, *справа* построен график силы тока I (в терминах тангенса угла отклонения), когда цепь замкнута проводом длиной x (в сантиметрах), для пяти различных значений x. Данные, показанные здесь на графике, взяты из [48].

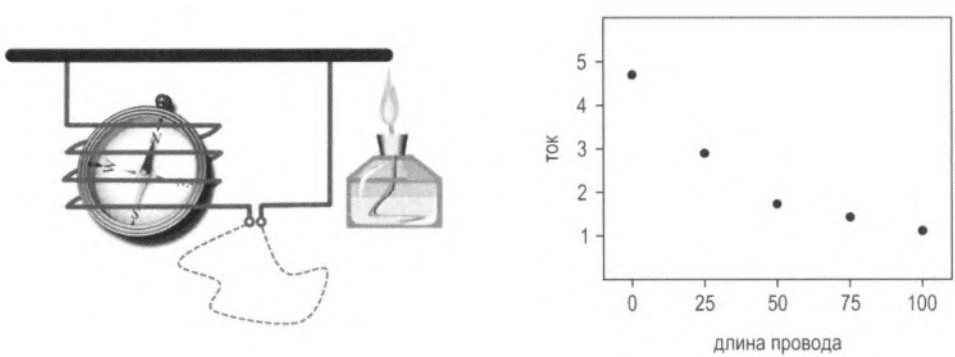


Рис. 10.19. Иллюстрация упражнения 10.5. Экспериментальная установка для проверки закона Ома. Черные и коричневые провода состоят соответственно из константана и меди (*слева*). Измерения тока для пяти разных длин замыкающего провода (*справа*)

А) Предложить одно нелинейное преобразование исходных данных такое, чтобы преобразованные входные/выходные данные были связаны линейно.

Б) Сформулировать надлежащую стоимостную функцию наименьших квадратов, используя ваши преобразованные данные, минимизировать ее для восстановления идеальных параметров, вписать вашу предложенную модель в данные и показать ее в исходном пространстве данных.

10.6. Моделирование волн

Повторить эксперимент, описанный в примере 10.3. От вас не требуется воспроизводить иллюстрации на рис. 10.7. Вместо этого следует использовать график истории стоимостной функции с целью обеспечения вашей способности усвоить точную подгонку к данным.

10.7. Эллиптическая граница решения

Повторить эксперимент, описанный в примере 10.5. От вас не требуется воспроизводить иллюстрации на рис. 10.11. Вместо этого следует использовать график истории стоимостной функции с целью обеспечения вашей способности усвоить точную подгонку к данным.

10.8. Конструирование признаков для двухклассового классификационного набора данных

Предложить нелинейную модель для набора данных, показанного на рис. 10.20, и выполнить нелинейную двухклассовую классификацию. Ваша модель должна достичь совершенной классификации на этом наборе данных.

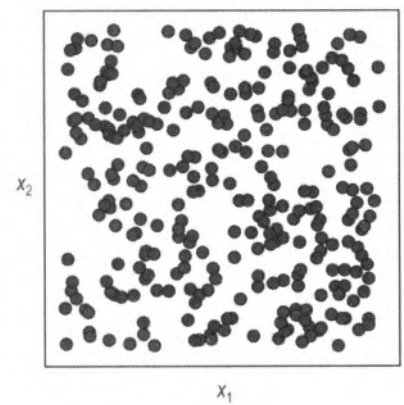


Рис. 10.20. Иллюстрация упражнения 10.8

10.9. Округлое топологическое многообразие

Повторить эксперимент, описанный в примере 10.7, и сообщить оптимальные значения, найденные для w_1, w_2 и r.

10.10. Еще одно нелинейное расширение метода главных компонент (PCA)

Как мы видели в *разд. 8.3*, минимизирование стоимости наименьших квадратов в основе PCA

$$g(\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_P, \mathbf{C}) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\mathbf{w}_p - \mathbf{x}_p\right\|_2^2 \tag{10.56}$$

над охватывающим $N \times K$-множеством $\mathbf{C}$ и соответствующими весовыми $K \times 1$-векторами $\mathbf{w}_1$, $\mathbf{w}_2$, ..., $\mathbf{w}_P$ определяет собственное K-мерное линейное подпространство для множества входных точек $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_P$.

В качестве альтернативы расширению автокодировщиков, чтобы обеспечить возможность PCA улавливать нелинейные подпространства (как подробно описано в *разд. 10.5*), мы можем расширить стоимость наименьших квадратов в основе PCA. Эта более ограниченная версия нелинейного PCA, часто именуемая *ядром*

PCA, является базисом для аналогичных нелинейных расширений других задач обучения без учителя, включая метод K средних. В данном упражнении вы исследуете фундаментальные принципы, связанные с этим нелинейным расширением.

Для этого мы начнем с выбора множества из B нелинейных признаковых преобразований f_1, f_2, ..., f_B, которые не имеют внутренних параметров, и обозначим преобразование точки через $\mathbf{x}_p$, используя все это множество признаковых преобразований как

$$\mathbf{f}_p = \begin{bmatrix} f_1(\mathbf{x}_p) \\ f_2(\mathbf{x}_p) \\ \vdots \\ f_B(\mathbf{x}_p) \end{bmatrix}. \tag{10.57}$$

Затем, вместо того чтобы усваивать линейное подпространство для наших входных данных, мы усваиваем одно для этих преобразованных входных данных путем минимизирования

$$g(\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_P, \mathbf{C}) = \frac{1}{P}\sum_{p=1}^{P} \left\| \mathbf{C}\mathbf{w}_p - \mathbf{f}_p \right\|_2^2 . \tag{10.58}$$

Обратите здесь внимание на то, что поскольку каждое $\mathbf{f}_p$ имеет размер $B \times 1$, наше охватывающее множество $\mathbf{C}$ должно иметь размер $B \times K$.

А) Допустив, что множество из B выбранных признаковых преобразований не имеет внутренних параметров, описать классическое решение на основе PCA задачи минимизирования стоимостной функции в уравнении (10.58). Подсказка: *см. разд. 8.4*.

Б) Предположим, что у нас есть набор входных точек, грубо распределенных по единичной окружности в двумерном пространстве (т. е. $N = 2$), и мы используем два признаковых преобразования $f_1(\mathbf{x}) = x_1^2$ и $f_2(\mathbf{x}) = x_2^2$. Какое подпространство мы найдем как в исходном пространстве, так и в пространстве преобразованных признаков, если будем использовать только первую главную компоненту из классического решения на основе PCA для представления наших данных? Нарисовать картину того, как это выглядит в обоих пространствах.

10.11. Нелинейное расширение *K* средних

Та же идея, введенная в предыдущем упражнении, может быть использована для расширения кластеризации методом K средних (*см. разд. 8.4*) до нелинейной конфигурации, а также является базисом для так называемого *ядра* K средних. Это можно сделать, сначала отметив, что алгоритмы и PCA, и K средних имеют одинаковую стоимостную функцию наименьших квадратов. Однако при использовании K средних минимизация этой стоимостной функции *ограничена* таким образом, что каждый весовой вектор $\mathbf{w}_p$ является стандартным базисным вектором (*см. разд. 8.7*).

А) Расширить задачу K средних точно так, как это было показано в предыдущем упражнении с PCA. Сравнить это с тем же видом кластеризации, выполненной на исходном входе, и описать словами, что именно кластеризуется в каждом экземпляре.

Б) Предположим, что у нас есть набор данных двумерных входных точек, распределенных примерно в двух кластерах: один кластер состоит из точек, распределенных примерно на единичной окружности, а другой — из точек, распределенных примерно на окружности радиуса два; обе окружности центрированы в начале координат. Используя два признаковых преобразования $f_1(\mathbf{x}) = x_1^2$ и $f_2(\mathbf{x}) = x_2^2$, указать, какие кластеры мы найдем при правильном выполнении K средних с $K = 2$, как в исходном пространстве, так и в пространстве преобразованных признаков? Нарисовать картину того, как это выглядит в обоих пространствах.

11 Принципы усвоения признаков

11.1. Введение

В *главе 10* мы увидели, как линейные алгоритмы машинного обучения с учителем и без учителя могут быть расширены для выполнения нелинейного усвоения посредством нелинейных функций (или признаковых преобразований), которые мы конструировали сами, визуально исследуя данные. Например, мы выразили общую нелинейную модель для регрессии как взвешенную сумму B нелинейных функций наших входных данных в виде

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + ... + f_B(\mathbf{x})w_B, \quad (11.1)$$

где f_1, ..., f_B — нелинейные параметризованные или непараметризованные функции (или признаки) данных, а все w с w_0 по w_B (вместе с любыми дополнительными весами, внутренними для нелинейных функций) представлены в весовом множестве Θ.

В этой главе мы подробно рассмотрим фундаментальные инструменты и принципы *усвоения признаков* (или автоматическую инженерию признаков), которые позволяют нам автоматизировать эту задачу и усваивать соответствующие признаки из самих данных, а не *конструировать* их самостоятельно. В частности, мы обсудим приемы выбора формы нелинейных преобразований с f_1 по f_B, число B задействуемых преобразований, а также то, как параметры в Θ настраиваются автоматически и для *любого набора данных*.

11.1.1. Пределы конструирования нелинейных признаков

Как мы уже описывали в предыдущих главах, *признаки* — это те определяющие характеристики конкретного набора данных, которые обеспечивают оптимальное усвоение. В *главе 10* мы увидели, как качество математических признаков, которые мы можем сконструировать сами, зависит от уровня наших знаний о явлении, которое мы изучаем. Чем больше мы понимаем (как интеллектуально, так и интуитивно) процесс, генерирующий данные, которые у нас есть под рукой, тем лучше мы можем сконструировать признаки самостоятельно. В одном крайнем случае, где мы имеем почти совершенное понимание процесса, генерирующего наши данные, причем это знание приходит из значительной интуитивной, экспериментальной и математической рефлексии, признаки, которые мы конструируем, обеспечивают поч-

ти совершенную результативность. Однако чаще всего мы знаем лишь несколько фактов или, возможно, вообще ничего не знаем о данных, которые анализируем. Вселенная огромна и сложна, и у нас есть твердое понимание в отношении того, как работает лишь малая ее часть.

Большинство (в особенности современных) наборов данных машинного обучения имеют гораздо больше, чем два входа, что делает визуализацию бесполезной в качестве инструмента для инженерии признаков. Но даже в редких случаях, когда визуализация данных возможна, мы не можем полагаться просто на наши собственные навыки распознавания шаблонов. Возьмем, к примеру, два набора данных, проиллюстрированных на рис. 11.1. Набор данных слева — это регрессионный набор данных с одномерным входом, и набор данных справа — это двухклассовый классификационный набор данных с двумерным входом. Истинная опорная нелинейная модель, используемая для генерирования данных в каждом конкретном случае, показана пунктирными черными линиями. Мы, люди, обычно обучены распознавать *на глаз* только простейшие нелинейные шаблоны, в том числе созданные элементарными функциями (например, многочленами низкой степени, экспоненциальными функциями, синусоидами) и с простыми контурами (например, квадратами, окружностями, эллипсами). Ни один из шаблонов, показанных на рисунке, не совпадает с такими простыми нелинейными функциями. Отсюда, независимо от того, удастся ли визуализировать набор данных, инженерия надлежащих нелинейных признаков человеком может оказаться затрудненной, если вообще не невозможной.

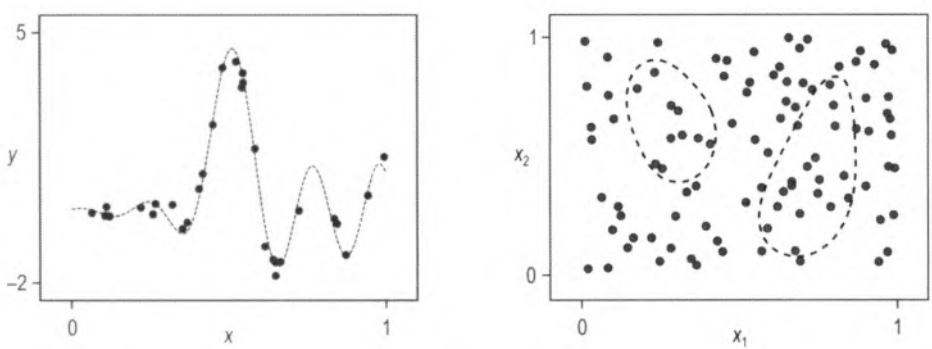


Рис. 11.1. Регрессионный (*слева*) и двухклассовый классификационный (*справа*) наборы данных, которые явно демонстрируют нелинейное поведение. Истинная опорная нелинейная функция, используемая для генерирования данных в каждом конкретном случае, показана пунктирной линией черного цвета

Именно указанная трудность служит мотивом для фундаментальных инструментов *обучения признакам*, описанных в этой главе. Одним словом, эти технологии *автоматизируют* процесс идентифицирования соответствующих нелинейных признаков для произвольных наборов данных. С такими инструментами нам больше не нужно конструировать подходящие нелинейности, по крайней мере так, как мы конструировали нелинейные признаки в предыдущей главе. Вместо этого мы стремимся определить их допустимые формы с помощью обучения. По сравнению

с нашими собственными ограниченными возможностями в распознавании нелинейных шаблонов инструменты обучения признакам способны идентифицировать практически любой нелинейный шаблон, присутствующий в наборе данных, независимо от его входной размерности.

11.1.2. Краткий обзор главы

Цель автоматизации нелинейного обучения амбициозна и, возможно, на первый взгляд пугает, поскольку существует бесконечное разнообразие нелинейностей и нелинейных функций на выбор. Каким образом нам в общем случае автоматически разобрать эту бесконечность, чтобы определить соответствующую нелинейность для конкретного набора данных?

Первый шаг, как мы увидим в *разд. 11.2*, состоит в том, чтобы организовать преследование цели автоматизации, сначала поместив фундаментальные строительные блоки этой бесконечности в *управляемые коллекции* (относительно простых) нелинейных функций. Эти коллекции часто именуются универсальными аппроксиматорами, из которых широкой популярностью пользуются три штамма и которые мы вводим здесь: фиксированно-контурные аппроксиматоры, искусственные нейронные сети и деревья. После введения понятия универсальных аппроксиматоров мы обсудим фундаментальные концепции, лежащие в основе того, как они задействуются, включая необходимость *валидационной ошибки* в качестве измерительного инструмента в *разд. 11.3*, описание *перекрестной валидации* и *компромисса между смещением и дисперсией* в *разд. 11.4*, автоматическую настройку нелинейной сложности посредством бустинга и регуляризации соответственно в *разд. 11.5* и *11.6*, а также понятие тестовой ошибки в *разд. 11.7* и бутсрап-агрегирования в *разд. 11.9*.

11.1.3. Метафора со сложностным наборным диском для усвоения признаков

Конечная цель усвоения признаков — это парадигма для надлежащего и автоматического усвоения признаков для *любого* набора данных, независимо от типа задачи. Она транслируется — формально говоря — в автоматическое определение как надлежащей формы общей нелинейной модели в уравнении (11.1), так и надлежащей *параметрической настройки* этой модели независимо от тренировочных данных и типа задачи. Мы можем рассматривать эту задачу метафорически, во-первых, как *строительство* и, во-вторых, как *автоматическую настройку* "сложностного наборного диска", как показано на рис. 11.2 для простого нелинейно-регрессионного набора данных (впервые использованного в примере 10.1). Эта концептуализация в форме сложностного наборного диска для усвоения признаков визуально изображает задачу усвоения признаков на высоком уровне как наборный диск, который должен быть создан и автоматически настроен для определения надлежащей степени модельной сложности, необходимой для представления явления, порождающего конкретный набор данных.

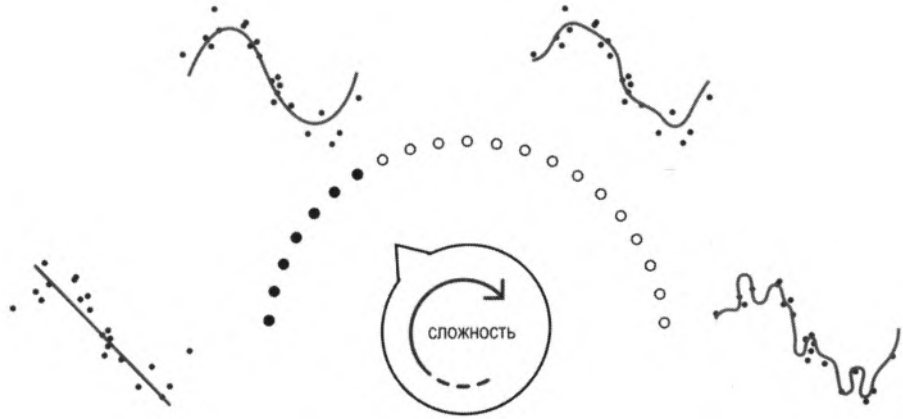


Рис. 11.2. Визуальное изображение усвоения признаков как *строительства* и *автоматической настройки* "сложностного наборного диска", который в широком смысле управляет формой нелинейной модели в уравнении (11.1), а также настройкой ее параметров и, следовательно, сложностью модели по отношению к тренировочным данным

Поворот этого сложностного диска до конца влево соответствует, вообще говоря, выбору модели с наименьшей нелинейной сложностью (т. е. линейной модели, как показано визуально на рисунке). По мере того как диск поворачивается слева направо, опробываются различные модели возрастающей сложности относительно тренировочных данных. Если повернуть его слишком далеко вправо, то результирующая модель будет слишком сложной (или слишком "извилистой") по отношению к тренировочным данным (как показано визуально на двух небольших фрагментах справа от диска). Когда же он установлен "как надо" (как показано визуально на маленьком изображении поверх сложностного диска, которое находится на втором месте слева), то результирующая модель очень хорошо представляет данные, а также опорное явление, которое их генерирует.

Хотя диск настройки сложности и является упрощенным описанием процедуры усвоения признаков, мы увидим, что эта метафора тем не менее полезна, поскольку она поможет нам организовать наше понимание разнообразного набора идей, участвующих в ее надлежащем осуществлении.

11.2. Универсальные аппроксиматоры

В предыдущей главе мы описали, как самостоятельно конструировать надлежащие нелинейные признаки, чтобы они совпадали с шаблонами, которые мы разглядели в простых наборах данных. Однако очень редко на практике удается конструировать совершенные или даже высокорезультативные нелинейные признаки, полностью полагаясь на наше собственное понимание набора данных, независимо от того, достигается ли это визуализацией данных, философской рефлексией или практическим опытом.

В этом разделе мы избавимся от нереалистичного допущения о том, что надлежащие нелинейные признаки можно сконструировать так, как описано в предыдущей

главе, и заменим его столь же нереалистичным допущением, которое имеет гораздо более практические последствия (как мы увидим в следующих разделах), о том, что у нас есть *полный* и *бесшумный* доступ к явлению, генерирующему наши данные. Здесь мы увидим, в случае, когда мы имеем такой неограниченный доступ к данным, что абсолютно совершенные признаки могут быть усвоены автоматически путем совмещения элементов из множества базовых признаковых преобразований, именуемых *универсальными аппроксиматорами*. В этом разделе мы также увидим элементарные образчики трех наиболее популярных универсальных аппроксиматоров, а именно *фиксированно-контурные* аппроксиматоры, *нейронные сети* и *деревья*.

Для простоты мы ограничимся рассмотрением нелинейной регрессии и двухклассовой классификации, которые, как мы видели в *главе 10*, имеют одинаковую нелинейную модель, сформированную в виде линейной комбинации B нелинейных признаковых преобразований входа

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + ... + f_B(\mathbf{x})w_B . \qquad (11.2)$$

Напомним, что в случае нелинейной двухклассовой классификации мы просто пропускаем нелинейно-регрессионную модель в уравнении (11.2) через математическую знаковую функцию, чтобы делать двоичные предсказания. Хотя наш фокус внимания в этом разделе будет на этих двух контролируемых задачах, поскольку общая нелинейная модель в уравнении (11.2) используется практически во всех других формах нелинейного усвоения, включая многоклассовую классификацию (*см. разд. 10.4*) и обучение без учителя (*см. разд. 10.6*), главный упор будет сделан на том, что представленная здесь история соблюдается в более общем плане для всех задач машинного обучения.

11.2.1. Совершенные данные

Начнем мы с того, что вообразим невозможное: *совершенный* набор данных для регрессии. Такой набор данных имеет две важные характеристики: он *совершенно бесшумен* и *бесконечно велик*. Первая характеристика, совершенная бесшумность, означает, что мы можем полностью доверять качеству всех до единой его пар "вход — выход". Бесконечная величина означает, что у нас есть неограниченный доступ к каждой паре "вход — выход" ($\mathbf{x}_p$, y_p) набора данных, которая может существовать. В совокупности такой набор данных *совершенно* описывает явление, которое их генерирует. На рис. 11.3, *сверху* проиллюстрированы виды такого совершенного набора данных в простейшем случае, когда входные/выходные данные связаны линейно.

Начиная с левой части рисунка, показан *реалистичный* набор данных (тот, с которым мы имеем дело на практике), который является одновременно *шумным* и *малым*. В центе изображен тот же самый набор данных, но с удаленным шумом из каждого выхода. В правой части рисунка представлен *совершенный* набор данных, в который добавлены все недостающие на линии точки в бесшумные данные в средней части; в результате данные выглядят как непрерывная прямая (или гиперплоскость в более высоких размерностях).

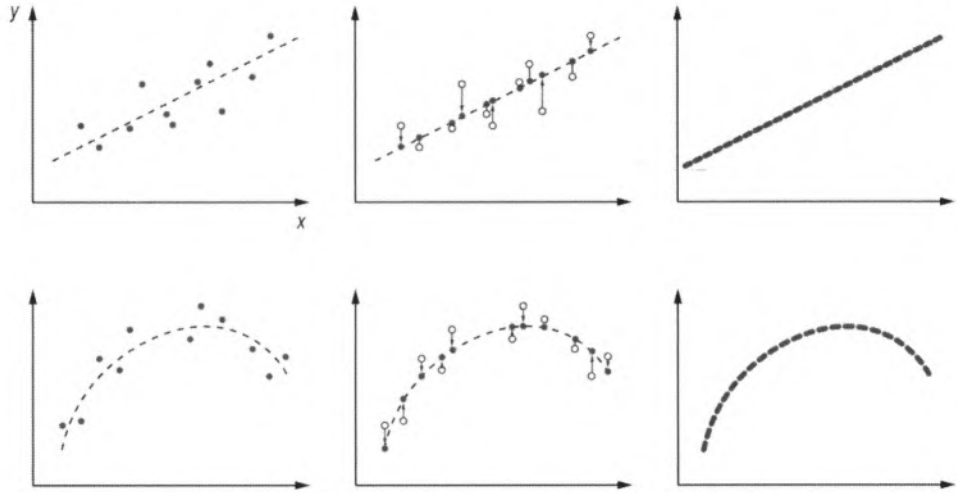


Рис. 11.3. Прототипический реалистичный линейно-регрессионный набор данных представляет собой шумный и (относительно) малый набор точек, которые могут быть грубо смоделированы прямой линией (*слева сверху*). Тот же самый набор данных, в котором все шумы были удалены из каждого выхода (*в центре сверху*). Совершенный линейно-регрессионный набор данных, где у нас бесконечно много точек лежат точно на прямой (*справа сверху*). Прототипический реалистичный нелинейно-регрессионный набор данных представляет собой шумный и (относительно) малый набор точек, которые могут быть грубо смоделированы нелинейной кривой (*слева снизу*). Весь шум удален из выхода, создавая бесшумный набор данных (*в центре снизу*). Совершенный нелинейно-регрессионный набор данных, где у нас бесконечно много точек лежат точно на кривой (*справа снизу*)

На рис. 11.3, *снизу* представлен аналогичный переход для прототипического нелинейно-регрессионного набора данных, в котором совершенные данные (показанные в правой части рисунка) прорезают *непрерывную* нелинейную кривую (или поверхность, в более высоких размерностях).

В случае двухклассовой классификации совершенный набор данных (по умолчанию использующий значения меток $y_p \in \{-1, +1\}$ будет иметь одинаковые характеристики: он *абсолютно бесшумен* и *бесконечно велик*. Однако в этом случае совершенные данные будут выглядеть не как *непрерывная* кривая или сама поверхность, а как шаговая функция с непрерывной нелинейной границей между ее верхними и нижними шагами. Это показано на рис. 11.4, который очень точно отражает то, что мы видели с регрессией на рис. 11.3.

Одним словом, совершенный регрессионный набор данных — это непрерывная функция с неизвестным уравнением. По этой причине мы будем ссылаться на наши совершенные данные, используя функциональную нотацию $y(\mathbf{x})$, означающую, что пара данных, определенная на входе **x**, может быть записана либо как $(\mathbf{x}, y(\mathbf{x}))$, либо аналогичным образом как (**x**, y). В том же ключе совершенный двухклассовый классификационный набор данных может быть представлен в качестве шаговой функции sign(y(**x**)) с непрерывной границей, определяемой $y(\mathbf{x})$. Важно иметь

в виду, что из функционального обозначения $y(\mathbf{x})$ не следует, что мы знаем формулу замкнутой *формы*, связывающую пары "вход — выход" совершенного набора данных; мы это не знаем! И действительно, наша следующая цель состоит в том, чтобы понять, как такая формула может быть выведена для адекватного представления совершенного набора данных.

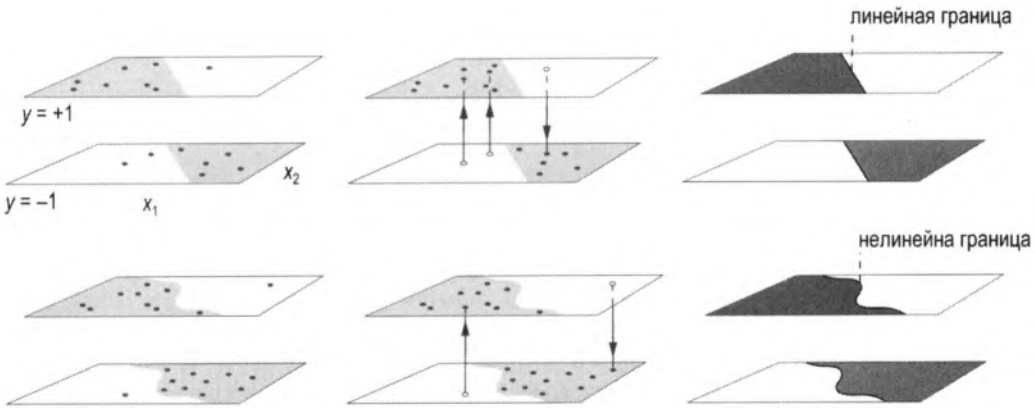


Рис. 11.4. Прототипический реалистичный линейный двухклассовый классификационный набор данных представляет собой шумный и (относительно) малый набор точек, который может быть грубо смоделирован с помощью шаговой функции с линейной границей (*слева сверху*). Продолжаем удалять весь шум из данных, возвращая истинные значения меток в наши шумные точки (*в центре сверху*). Совершенный линейный двухклассовый классификационный набор данных, где у нас бесконечно много точек лежит строго на шаговой функции с линейной границей (*справа сверху*). Прототипический реалистичный нелинейный двухклассовый классификационный набор данных представляет собой шумный и (относительно) малый набор точек, которые могут быть грубо смоделированы с помощью шаговой функции с нелинейной границей (*слева снизу*). Продвигаемся вперед, чтобы удалить весь шум из данных, создавая бесшумный набор данных (*в центре снизу*). Совершенный нелинейный двухклассовый классификационный набор данных, где у нас бесконечно много точек лежит точно на шаговой функции с нелинейной границей (*справа снизу*)

11.2.2. Аналогия с охватывающим множеством для универсальной аппроксимации

Здесь мы будем использовать наши знания и интуицию об основных линейно-алгебраических понятиях, таких как векторы, охватывающие множества и т. п. (*см. разд. 8.2*), чтобы понять лучше то, каким образом можно комбинировать нелинейные функции для моделирования совершенных регрессионных и классификационных данных. В частности, мы увидим, как векторы и нелинейные функции очень похожи, когда речь заходит о понятиях линейной комбинации и охватывающих множеств.

Линейные комбинации векторов и функций

Для начала будем считать, что у нас есть множество из B векторов $\{\mathbf{f}_1, \mathbf{f}_2, ..., \mathbf{f}_B\}$, каждый из которых имеет длину N. Мы называем его *охватывающим* множеством векторов. Тогда, с учетом конкретного множества весов с w_1 по w_B, линейная комбинация

$$\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + ... + \mathbf{f}_B w_B = \mathbf{y} \tag{11.3}$$

определяет новый N-мерный вектор $\mathbf{y}$. Это проиллюстрировано в верхнем ряду рис. 11.5 для конкретного множества векторов и весов, где $B = 3$ и $N = 3$.

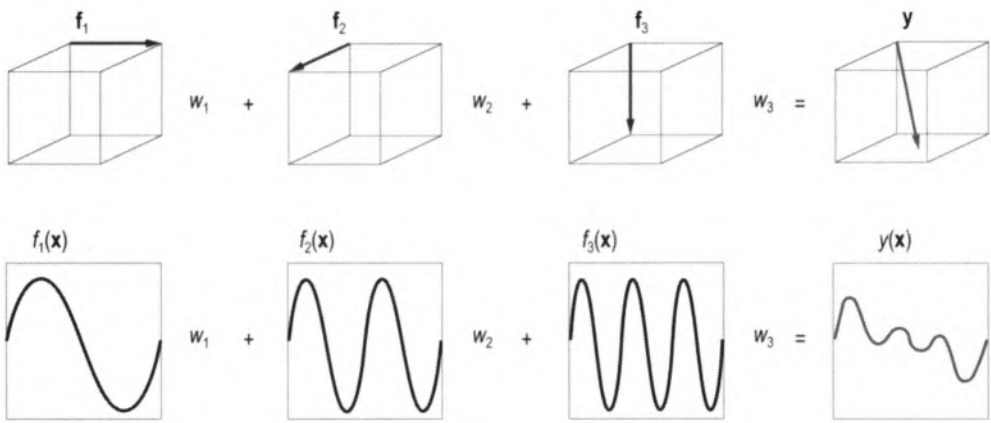


Рис. 11.5. Конкретная линейная комбинация векторов $\mathbf{f}_1$, $\mathbf{f}_2$ и $\mathbf{f}_3$ (показана черным цветом) создает новый вектор $\mathbf{y}$ (показанный синим цветом; *сверху*). Совершенно аналогичным образом конкретная линейная комбинация трех функций $f_1(\mathbf{x})$, $f_2(\mathbf{x})$ и $f_3(\mathbf{x})$ (показана черным цветом) создает новую функцию $y(\mathbf{x})$ (показанную синим цветом; *снизу*)

Арифметика нелинейных функций работает совершенно аналогичным образом: с учетом охватывающего множества из B нелинейных функций $\{f_1(\mathbf{x}), f_2, ..., f_B(\mathbf{x})\}$ (где вход $\mathbf{x}$ является N-мерным, а выход — скалярным) и соответствующего множества весов, линейная комбинация

$$w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + ... + f_B(\mathbf{x})w_B = y(\mathbf{x}) \tag{11.4}$$

определяет новую функцию $y(\mathbf{x})$. Это показано в нижнем ряду рис. 11.5 для конкретного набора функций и весов, где $B = 3$ и $N = 1$.

Обратите внимание на сходство между векторной и функциональной арифметикой в уравнениях (11.3) и (11.4): взятие определенной линейной комбинации множества векторов создает новый вектор с качествами, унаследованными от каждого вектора $\mathbf{f}_b$ в множестве, точно так же, как взятие линейной комбинации множества функций создает новую функцию, принимающую качества каждой функции $f_b(\mathbf{x})$ в этом множестве. Одним из отличий между двумя линейно-комбинационными формулами является наличие *параметра смещения* w_0 в уравнении (11.4). Этот параметр смещения можно было бы включить в одну из нелинейных функций и не делать явным (добавляя константную функцию в смесь), но мы решили оставить

его прямо перед линейной комбинацией функций (как мы делали с линейными моделями в предыдущих главах). Единственная цель этого параметра смещения состоит в том, чтобы переместить нашу линейную комбинацию функций вертикально вдоль выходной оси.

Емкость охватывающих множеств

Вычисление вектора **y** в уравнении (11.3) для заданного множества весов $w_1, ..., w_B$ является тривиальным делом. С другой стороны, обратная задача, т. е. отыскание весов, заданных вектором **y**, несколько труднее. Выражаясь алгебраически, мы хотим найти такие веса $w_1, ..., w_B$, где векторная аппроксимация

$$\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + ... + \mathbf{f}_B w_B \approx \mathbf{y} \tag{11.5}$$

соблюдается как можно лучше. Это показано на простом примере в верхнем ряду рис. 11.6.

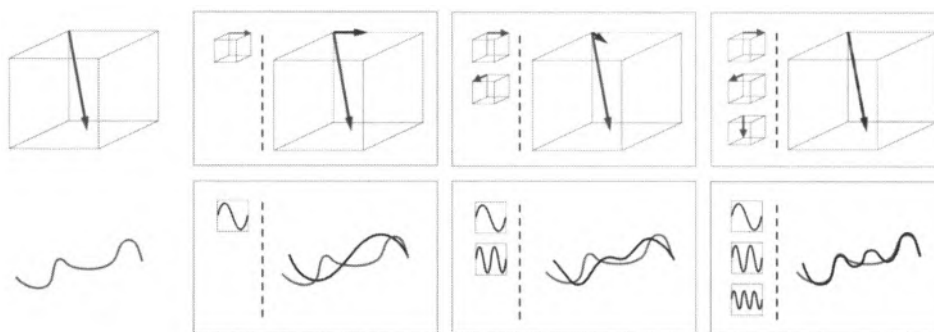


Рис. 11.6. Трехмерный вектор **y** (показан красным цветом на *первом сверху* рисунке) аппроксимируется с помощью одного (*второй сверху* рисунок), двух (*третий сверху* рисунок) и трех (*четвертый сверху* рисунок) охватывающих векторов (здесь стандартные базисные векторы). По мере увеличения числа охватывающих векторов мы можем аппроксимировать **y** все с большей точностью. Та же самая концепция применима и к функциям. Непрерывная функция со скалярным входом $y(\mathbf{x})$ (показана красным цветом) аппроксимируется с помощью одной (*второй снизу* рисунок), двух (*третий снизу* рисунок) и трех (*четвертый снизу* рисунок) охватывающих функций (здесь синусные волны варьирующейся частоты). По мере увеличения числа функций мы можем аппроксимировать $y(\mathbf{x})$ все с большей точностью

Качество *векторной аппроксимации* в уравнении (11.5) зависит от трех важнейших и взаимосвязанных факторов: во-первых, от разнообразия (т. е. линейной независимости) охватывающих векторов; во-вторых, от числа B используемых векторов (в общем случае, чем больше мы делаем B, тем лучше); и в-третьих, от того, насколько хорошо мы настраиваем веса $w_1, ..., w_B$ посредством минимизации соответствующей стоимости[1].

[1] Например, здесь мы можем использовать стоимость наименьших квадратов

$$g(w_1, w_2, ..., w_B) = \|\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + ... + \mathbf{f}_B w_B - \mathbf{y}\|_2^2. \tag{11.6}$$

Первые два фактора определяют *ранг* или *емкость* охватывающего множества, т. е. меру для диапазона векторов **y**, которые мы, скорее всего, можем представить с помощью такого охватывающего множества. Охватывающее множество с низкой емкостью, т. е. состоящее из неразнообразного и/или малого числа охватывающих векторов, может аппроксимировать только крошечную часть тех, которые присутствуют во всем векторном пространстве. С другой стороны, охватывающее множество с *большой* емкостью может представлять более широкую полосу пространства. Понятие емкости (capacity) для охватывающего множества векторов проиллюстрировано для конкретного охватывающего множества на рис. 11.7, *сверху*.

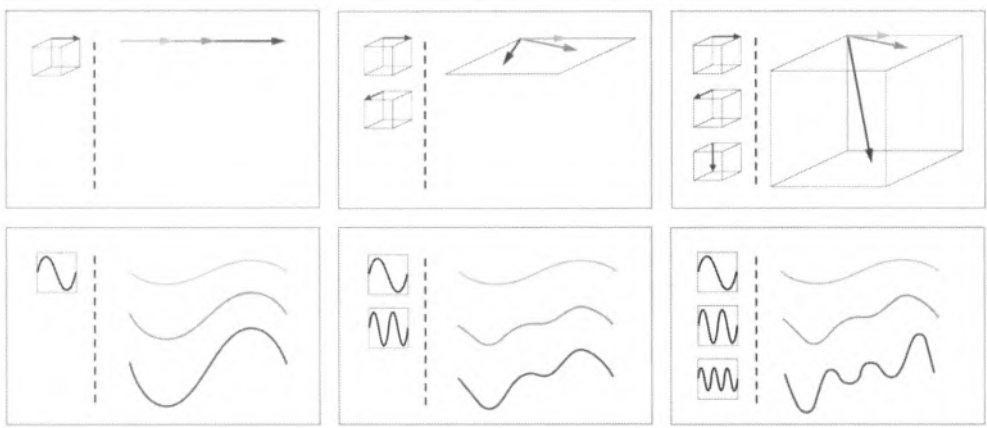


Рис. 11.7. По мере увеличения числа (разнообразных) векторов в охватывающем множестве, от одного в левой панели до двух и трех соответственно в средней и правой панелях, мы увеличиваем емкость охватывающего множества. Это отражается в растущем разнообразии образцовых векторов, создаваемых с использованием каждого охватывающего множества (показанного разными оттенками синего; *сверху*). Та же самая концепция применима и к функциям. По мере того как мы увеличиваем число (разнообразных) функций в охватывающем множестве, от одной функции в левой панели до двух и трех функций соответственно в средней и правой панелях, мы увеличиваем емкость охватывающего множества. Это отражается в растущем разнообразии выборочных функций, создаваемых с использованием каждого охватывающего множества (показанного разными оттенками синего; *снизу*)

Переключив наше внимание с векторов на функции, следует отметить, что вычисление функции $y(\mathbf{x})$ в уравнении (11.4) для *заданного* множества весов w_1, ..., w_B является прямым. Как и в векторном случае, мы можем обратить эту задачу вспять и попытаться найти такие веса w_1, ..., w_B для заданной $y(\mathbf{x})$, что функциональная аппроксимация

$$w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + ... + f_B(\mathbf{x})w_B \approx y(\mathbf{x}) \qquad (11.7)$$

соблюдается как можно лучше. Это иллюстрируется простым примером на рис. 11.6, *снизу*.

Качество соблюдения этой *функциональной аппроксимации* зависит от трех важнейших и взаимосвязанных факторов: во-первых, от разнообразия охватывающих

функций; во-вторых, от числа B используемых функций; и в-третьих, от того, насколько хорошо мы настраиваем веса w_0, ..., w_B (а также любые параметры, внутренние для наших нелинейных функций) посредством минимизации соответствующей стоимости[2].

По аналогии с векторным случаем первые два фактора определяют емкость охватывающего множества функций. *Малоемкое* охватывающее множество, которое использует неразнообразный и/или малый массив нелинейных функций, способен представлять только малый диапазон нелинейных функций. С другой стороны, охватывающее множество с *большой* емкостью может представлять более широкую полосу функций. Понятие емкости для охватывающего множества функций иллюстрируется для конкретного охватывающего множества на рис. 11.7, *снизу*.

Иногда охватывающие функции f_1, ..., f_B параметризуются, т. е. они сами имеют внутренние параметры. Непараметризованная охватывающая функция очень похожа на охватывающий вектор, поскольку они оба свободны от параметров. Параметризованная охватывающая функция, со своей стороны, может сама принимать разные контуры и, таким образом, обладать высокой емкостью. То же самое нельзя

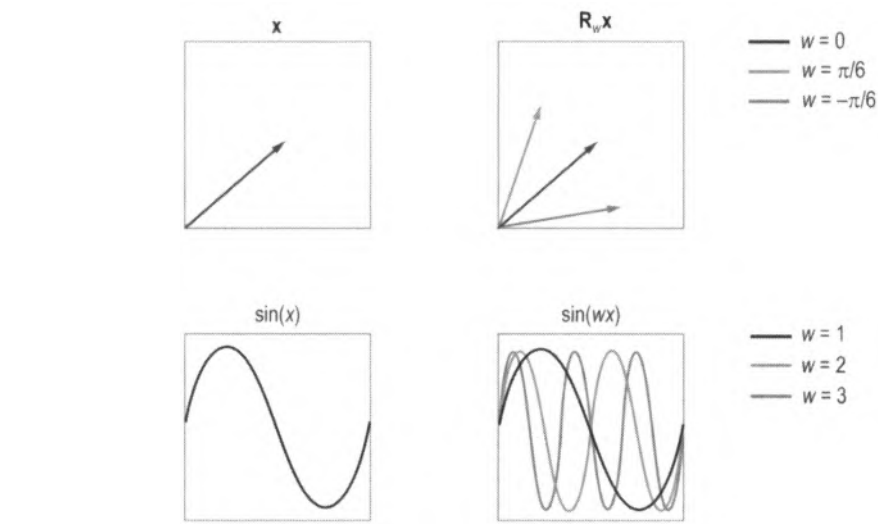


Рис. 11.8. Обычный охватывающий вектор (*слева сверху*). Непараметризованная охватывающая функция (*слева снизу*). Параметризованная охватывающая функция с одним внутренним параметром. Изменяя значение этого внутреннего параметра, можно сделать так, чтобы он принимал разные контуры (*справа снизу*). Параметризованный охватывающий вектор (предварительно умноженный на матрицу поворота в уравнении (11.9)) меняет направление в зависимости от того, как задан параметр w (*справа сверху*)

---

[2] Например, здесь мы можем использовать стоимость наименьших квадратов

$$g(w_0, w_1, ..., w_B) = \int_{\mathbf{x}\in\mathcal{D}} (w_0 + f_1(\mathbf{x})w_1 + ... + f_B(\mathbf{x})w_B - y(\mathbf{x}))^2 \, d\mathbf{x}, \qquad (11.8)$$

где $\mathcal{D}$ — это любая желаемая часть входной области значений.

сказать об охватывающих векторах и непараметризованных охватывающих функциях. Эта концепция проиллюстрирована на рис. 11.8, где слева показан обычный охватывающий вектор $\mathbf{x} = [1 \quad 1]^T$ (*слева сверху*) вместе с непараметризованной охватывающей функцией, т. е. $\sin(\mathbf{x})$ (*слева снизу*). Справа снизу изображена параметризованная функция $\sin(wx)$, которая может представлять более широкий диапазон разных функций, поскольку ее внутренний параметр w корректируется. Рассуждая аналогично, мы также можем параметризовать охватывающий вектор $\mathbf{x}$, например, умножив его на матрицу поворота

$$\mathbf{R}_w = \begin{bmatrix} \cos(w) & -\sin(w) \\ \sin(w) & \cos(w) \end{bmatrix}. \tag{11.9}$$

Это позволяет ему поворачиваться в плоскости и представлять диапазон разных векторов в зависимости от того, как задан угол поворота w.

Универсальная аппроксимация

В случае векторной аппроксимации в уравнении (11.5), если мы выбираем $B \geq N$ векторов для нашего охватывающего множества и по крайней мере N из них линейно независимы, то наше охватывающее множество имеет максимальную емкость, и поэтому мы можем аппроксимировать *каждый* N-мерный вектор $\mathbf{y}$ с *любой* заданной прецизионностью, при условии, что мы надлежаще настраиваем параметры линейной комбинации. Такое множество охватывающих векторов, которых бесконечно много для N-мерного векторного пространства, может аппроксимировать (или в этом случае совершенно представлять) каждый вектор *универсально*, и отсюда оно иногда называется *универсальным аппроксиматором*. Например, простой стандартный базис (см. упражнение 8.1) для векторного пространства является часто встречающимся примером охватывающего множества, которое является *универсальным аппроксиматором*. Понятие универсальной аппроксимации векторов проиллюстрировано на рис. 11.9, *сверху*.

То же самое относится и к функциональной аппроксимации в уравнении (11.7). Если мы выберем правильный вид охватывающих функций, то наше охватывающее множество получит максимальную емкость, и поэтому мы можем аппроксимировать *каждую* функцию $y(\mathbf{x})$ с *любой* заданной прецизионностью при условии, что мы надлежаще настраиваем параметры линейной комбинации. Такое множество охватывающих функций, у которых бесконечно много разновидностей, может аппроксимировать каждую функцию *универсально*, и поэтому оно часто называется *универсальным аппроксиматором*. Это понятие универсальной аппроксимации функций показано на рис. 11.9, *снизу*.

Одно из отличий между векторным и функциональным режимами универсальной аппроксимации состоит в том, что для аппроксимации заданной функции с произвольной прецизионностью может потребоваться бесконечно много охватывающих функций (в то время как для первого всегда достаточно установить B бóльшим или равным N).

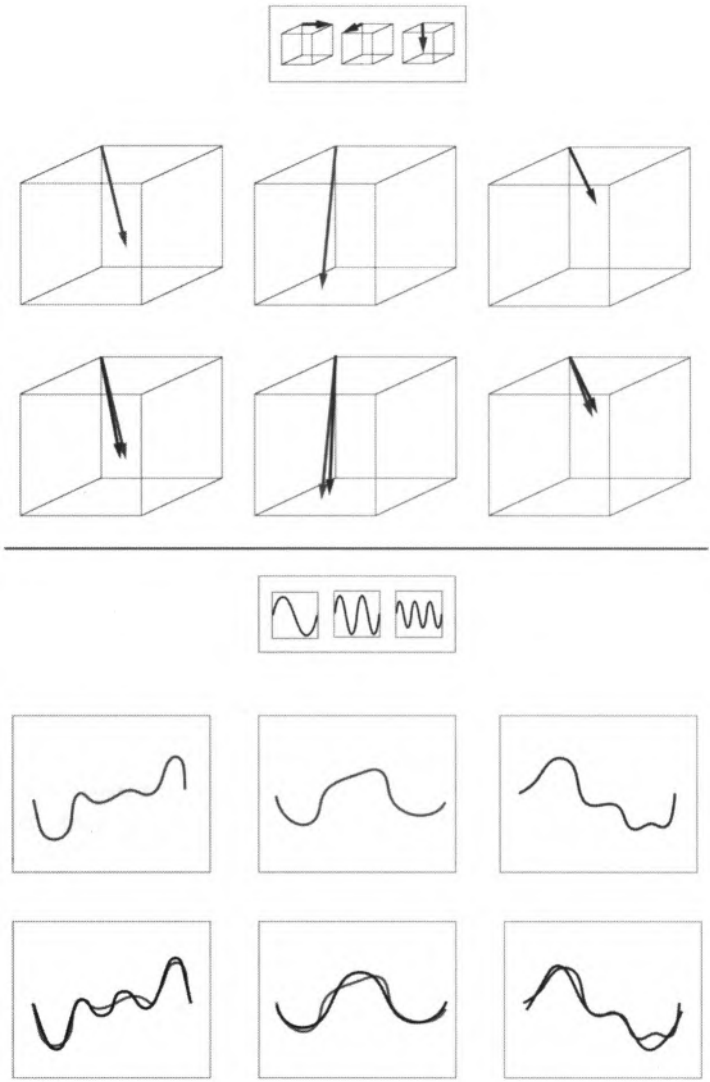


Рис. 11.9. Универсальная аппроксимация, проиллюстрированная в векторном случае (*верхняя часть* рисунка). Охватывающее множество универсального аппроксиматора, состоящее из трех векторов, показанных черным цветом (*верхний ряд*). Три примера аппроксимируемых векторов, окрашенных красным, розовым и синим цветами слева направо (*средний ряд*). Аппроксимация каждого вектора из среднего ряда с использованием охватывающего множества из верхнего ряда, показанная черным цветом в каждом случае (*нижний ряд*). Можно добиться совершенной аппроксимации, но для целей визуализации здесь она показана с небольшим сдвигом. Аналогичный сценарий универсальной аппроксимации, проиллюстрированный для функций (*нижняя часть* рисунка). Охватывающее множество универсального аппроксиматора, состоящее из трех функций (на практике может потребоваться гораздо больше охватывающих функций, чем показано здесь, *верхний ряд*). Три примера аппроксимируемых функций, окрашенных красным, розовым и синим цветами слева направо (*средний ряд*). Аппроксимация каждой функции в среднем ряду с использованием охватывающего множества в верхнем ряду, показанная черным цветом в каждом случае (*нижний ряд*)

11.2.3. Популярные универсальные аппроксиматоры

Когда речь идет об аппроксимировании функций, то существует огромное разнообразие охватывающих множеств, которые являются *универсальными аппроксиматорами*. Действительно, как и в векторном случае, в случае функций существует бесконечно много универсальных аппроксиматоров. Однако для целей организации, условностей, а также различных технических вопросов универсальные аппроксиматоры, используемые в машинном обучении, часто распределяют по трем главным категориям, именуемым *фиксированно-контурными аппроксиматорами*, *нейронными сетями* и *деревьями*. Мы приведем только самые базовые образчики каждой из этих трех категорий, на которые мы будем ссылаться в оставшейся части главы. Каждое из этих популярных семейств имеет свои уникальные практические сильные и слабые стороны в качестве универсального аппроксиматора, широкий спектр технических деталей для разведывания и условности использования.

Пример 11.1. Семейство фиксированно-контурных универсальных аппроксиматоров

Семейство *фиксированно-контурных* функций состоит из групп нелинейных функций без внутренних параметров, популярным примером которых являются *многочлены*[3]. Во время работы только с одним входом это подсемейство фиксированно-контурных функций состоит из

$$f_1(x) = x, \quad f_2(x) = x^2, \quad f_3(x) = x^3 \quad \text{и т. д.}, \tag{11.10}$$

где D-й элемент принимает форму $f_D(x) = x^D$. Сочетание первых D структурных единиц из этого подсемейства часто именуется *многочленом степени D*. Существует бесконечное число этих функций (по одной для каждого положительного целого числа D), и они *естественным образом* упорядочены по степени. Тот факт, что эти функции не имеют настраиваемых внутренних параметров, придает каждой из них *фиксированный контур* (рис. 11.10, *сверху*).

С двумя входами x_1 и x_2 общая многочленная единица степени D принимает аналогичную форму

$$f_h(x_1, x_2) = x_1^p x_2^q, \tag{11.11}$$

где p и q — неотрицательные целые числа, а $p + q \leq D$. Классически многочлен степени D — это линейная комбинация всех таких единиц. Кроме того, определение в уравнении (11.11) непосредственно обобщается и на более многомерный вход. Фиксированно-контурные аппроксиматоры подробнее рассматриваются в *главе 12*.

---

[3] Многочлены были первыми доказанными универсальными аппроксиматорами, что было показано в 1885 году с помощью так называемой аппроксимационной теоремы Стоуна — Вейерштрасса (см., например, [49]).

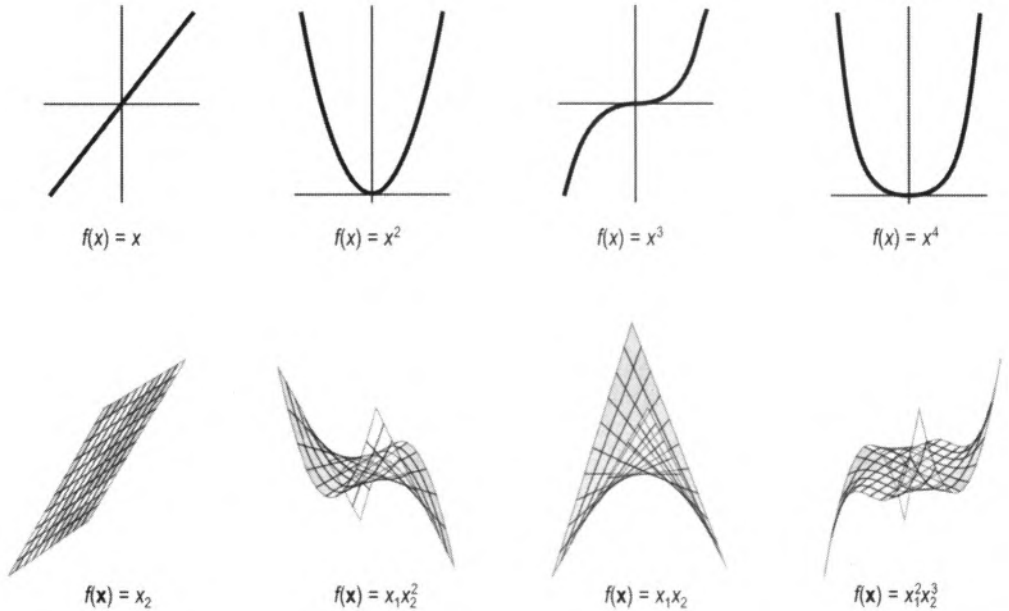


Рис. 11.10. Четыре структурные единицы из многочленного семейства универсальных фиксированно-контурных аппроксиматоров с одноразмерным (*сверху*) и двухразмерным (*снизу*) входами

Пример 11.2. Нейросетевое семейство универсальных аппроксиматоров

Еще одним популярным семейством универсальных аппроксиматоров являются *нейронные сети*[4]. В широком смысле нейронные сети состоят из *параметризованных* функций[5], позволяющих им принимать различные контуры (в отличие от описанных ранее фиксированно-контурных функций, каждая из которых принимает один-единственный фиксированный контур).

Простейшее подсемейство нейронных сетей состоит из параметризованных элементарных функций (например, tanh) формы

$$f_b(x) = \tanh\left(w_{b,0} + w_{b,1}x\right), \tag{11.12}$$

где внутренние параметры $w_{b,0}$ и $w_{b,1}x$ b-й структурной единицы позволяют ей принимать разнообразные контуры. На рис. 11.11, *сверху* проиллюстрирован этот факт случайной установкой значений двух ее внутренних параметров и построением графика результата.

Для построения нейросетевых признаков, принимающих более высокоразмерный вход, мы берем линейную комбинацию входа и пропускаем результат через нели-

[4] То, что нейронные сети являются универсальными аппроксиматорами, было показано еще в конце 1980-х — начале 1990-х годов [50–52].

[5] Эволюционный шаг между фиксированно-контурными и нейросетевыми структурными единицами, т. е. сетевые единицы, внутренние параметры которых рандомизируются и фиксируются, также являются универсальными аппроксиматорами [53, 54].

нейную функцию (здесь, tanh). Например, элемент f_b для общего N-мерного входа принимает форму

$$f_b(\mathbf{x}) = \tanh\left(w_{b,0} + w_{b,1}x_1 + ... + w_{b,N}x_N\right). \quad (11.13)$$

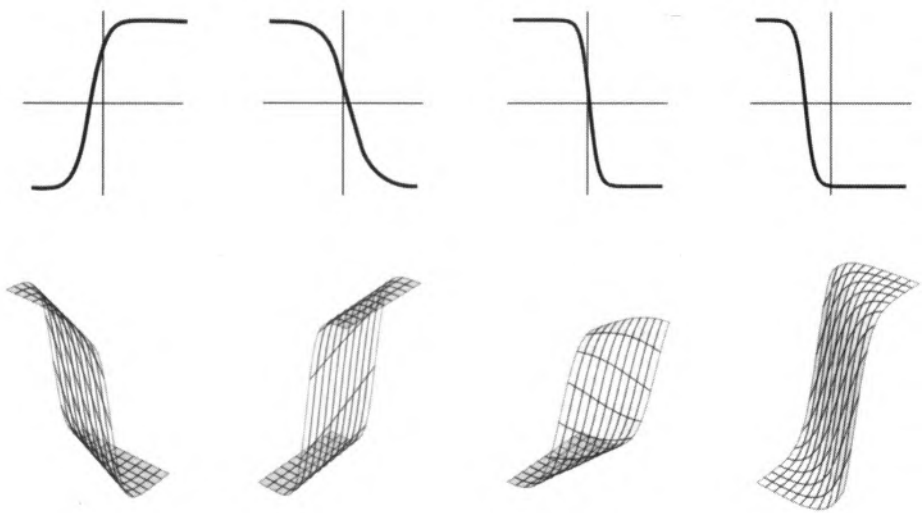


Рис. 11.11. В отличие от фиксированно-контурных аппроксиматоров, нейросетевые структурные единицы являются гибкими и могут принимать разнообразные контуры в зависимости от того, как мы задаем их внутренние параметры. Показаны четыре такие структурные единицы, принимающие $N = 1$ (*сверху*) и $N = 2$ (*снизу*) размерный вход, внутренние параметры которого задаются случайными в каждом экземпляре

Как и в случае с более низкоразмерным примером в уравнении (11.12), каждая функция в уравнении (11.13) может принимать самые разные формы, как показано в нижнем ряду рис. 11.11, в зависимости от того, как мы настраиваем ее внутренние параметры. Нейросетевые аппроксиматоры подробнее описаны в *главе 13*.

Пример 11.3. Древесное семейство универсальных аппроксиматоров

Подобно нейронным сетям, одиночный элемент из семейства древесных универсальных аппроксиматоров[6] может принимать самые разнообразные формы. Простейший вид дерева состоит из дискретных шаговых функций или, как их чаще называют, *пней*, чьи трещины пролегают вдоль одной-единственной размерности входного пространства. Пень с одномерным входом x можно записать как

$$f_b(x) = \begin{cases} v_1 & \text{при } x \le s; \\ v_2 & \text{при } x > s, \end{cases} \quad (11.14)$$

где s называется *точкой ветвления*, в которой пень меняет значения, а v_1 и v_2 — это значения, принимаемые соответственно двумя сторонами пня, которые мы на-

[6] Давно известно, что деревья являются универсальными аппроксиматорами. См., например, [49, 55].

зываем *листьями* пня. Древесный универсальный аппроксиматор — это множество таких пней, каждая структурная единица которых имеет свою собственную уникальную точку ветвления и листовые значения.

На рис. 11.12, *сверху* построены графики четырех экземпляров такого пня. Более высокоразмерные пни следуют этому *одномерному* шаблону. Сначала выбирается точка ветвления s вдоль одной входной размерности. Затем каждой стороне ветвления назначается одно листовое значение, как показано на рис. 11.12, *снизу* для двумерного входа. Древесные аппроксиматоры подробнее описаны в *главе 14*.

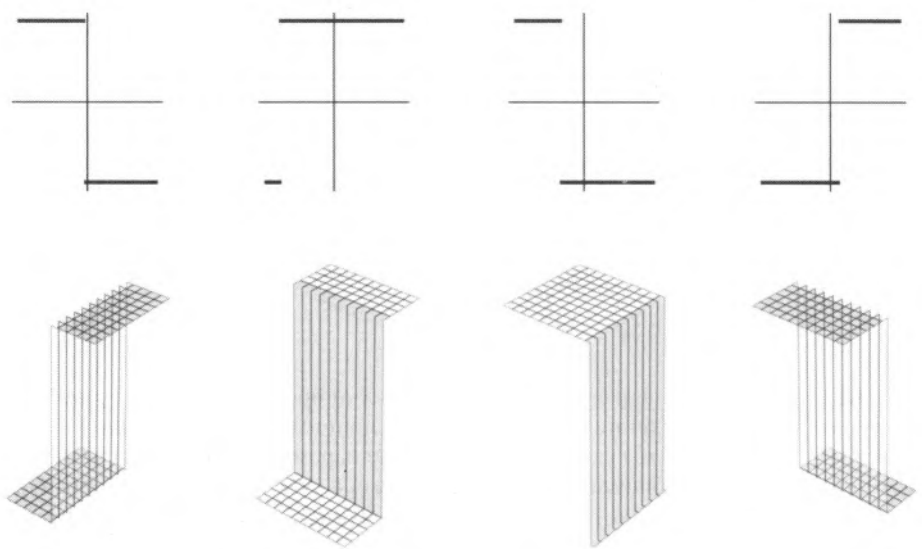


Рис. 11.12. Древесные структурные единицы могут принимать разнообразные контуры в зависимости от того, как назначаются их точки ветвления и листовые значения. Четыре экземпляра одномерного (*сверху*) и двумерного (снизу) пня

При формировании базовой нелинейной модели на основе универсального аппроксиматора

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + \ldots + f_B(\mathbf{x})w_B \qquad (11.15)$$

мы всегда используем структурные единицы из *одного-единственного* типа универсальных аппроксиматоров (например, всех фиксированно-контурных, нейросетевых или древесных единиц). Другими словами, мы не "смешиваем и сопоставляем", беря по несколько единиц из каждого главного семейства. Как мы увидим в настоящей главе, а также в последующих главах, ограничивая модельные признаковые преобразования одним семейством, мы можем (в каждом из трех случаев) лучше оптимизировать процесс обучения и эффективнее справляться с уникальными техническими экстравагантностями каждого семейства, связанными с фундаментальными задачами масштабирования, имеющими отношение к фиксированно-контурным единицам, невыпуклостью стоимостных функций, связанных с нейросетевыми единицами, и дискретной природой древесных единиц.

11.2.4. Емкостный и оптимизационный наборные диски

С помощью любого из главенствующих универсальных аппроксиматоров, введенных ранее (будь то фиксированно-контурные, нейросетевые или древесные), мы можем достичь универсальной аппроксимации с любой заданной прецизионностью при условии, что обобщенная нелинейная модель в уравнении (11.15) обладает достаточно большой *емкостью* (что может быть обеспечено достаточно большим B) и что ее параметры достаточно хорошо настраиваются посредством *оптимизации* связанной с ней стоимостной функции. Понятия *емкости* (capacity) и *оптимизации* такой нелинейной модели концептуально представлены на рис. 11.13 в виде двух наборных *дисков*.

Емкостный наборный диск визуально подытоживает величину емкости, которую мы допускаем в данной модели, причем каждая отметка на диске обозначает отдельную модель, построенную из структурных единиц универсального аппроксиматора. При установке диска в крайнее левое положение мы допускаем как можно меньшую емкость, т. е. задействуем *линейную* модель. Перемещая емкостный диск слева направо (по часовой стрелке), мы корректируем модель, добавляя все больше и больше емкости до тех пор, пока диск не будет установлен в крайнее правое положение. Когда диск установлен в это крайнее правое положение, мы представляем себе, что допускаем в нашей модели бесконечный объем емкости (например, используя бесконечное число структурных единиц из конкретного семейства универсальных аппроксиматоров).

Оптимизационный наборный диск визуально подытоживает то, насколько хорошо мы минимизируем стоимостную функцию данной модели, емкость которой уже установлена. Установка диска в крайнее левое положение обозначает начальную точку любого метода локальной оптимизации, который мы используем. Когда мы поворачиваем оптимизационный диск слева направо (по часовой стрелке), мы представляем себе, что двигаемся все дальше вдоль конкретного оптимизационного прогона, который мы используем для надлежащей настройки модельных параметров, причем последний шаг визуально ассоциировн с установкой диска в крайнее правое положение, и мы представляем, что успешно минимизировали связанную с моделью стоимостную функцию.

Обратите внимание, что при такой концептуализации каждая пара настроек (емкостного и оптимизационного наборных дисков) создает уникально настроенную модель: совокупная архитектура/дизайн модели определяется емкостным диском, а множество конкретных значений модельных параметров определяется оптимизационным диском. Например, одна конкретная настройка может соответствовать модели, состоящей из $B = 10$ нейросетевых единиц, параметры которой задаются путем выполнения 1000 шагов градиентного спуска, в то время как другая настройка соответствует модели, состоящей из $B = 200$ нейросетевых единиц, параметры которой задаются путем выполнения только 5 шагов градиентного спуска.

Держа в уме эти две концептуализации, мы можем рассуждать о концепции универсальной аппроксимации непрерывной функции как о повороте *обоих* наборных дисков до конца вправо (рис. 11.13, *снизу*). То есть для того чтобы аппроксимировать заданную непрерывную функцию с помощью универсального аппроксимато-

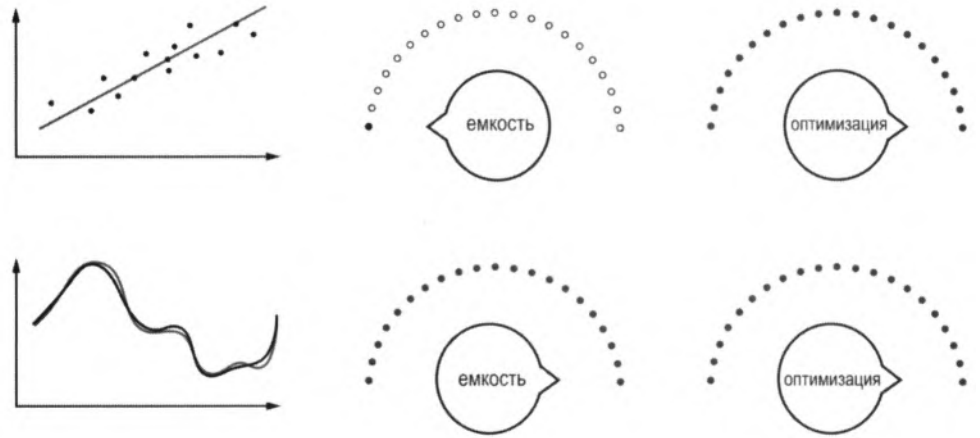


Рис. 11.13. Визуализация прецизионности емкости и оптимизации модели в виде двух наборных дисков. Когда емкостный диск установлен в крайнее левое положение, мы имеем низкоемкостную линейную модель; когда он установлен в крайнее правое положение, мы допускаем в модель максимальную (возможно, бесконечную) емкость. Оптимизационный диск, развернутый влево, обозначает начальную точку оптимизации, а до конца вправо — окончательный шаг успешной оптимизации. В случае с линейной регрессией (как мы видели в предыдущих главах) мы поворачиваем емкостный диск до конца влево и оптимизационный диск до конца вправо, чтобы найти наилучший возможный набор параметров для низкоемкостной линейной модели (нарисованной синим цветом), который соответствует регрессионным данным (*сверху*). В случае с универсальной аппроксимацией непрерывной функции (нарисованной черным цветом) мы поворачиваем оба диска вправо, допуская в модель бесконечную емкость и настраивая ее параметры путем оптимизирования до полного завершения (*снизу*)

ра, мы устанавливаем нашу модельную емкость как можно *большей* (возможно, бесконечно), поворачивая емкостный диск до конца вправо, и оптимизируем соответствующую стоимостную функцию, поворачивая оптимизационный диск до конца вправо.

В отличие от того рода линейного обучения, которое мы рассматривали в предыдущих главах (см. рис. 11.13, *сверху*), мы поворачиваем наш емкостный диск до конца влево (задействуя линейную модель), но при этом устанавливаем наш оптимизационный диск в крайнее правое положение. Оптимизируя до полного завершения, мы определяем надлежащее смещение и наклон нашей линейной модели при выполнении, например, линейной регрессии, как показано на рисунке.

Теперь мы рассмотрим ряд простых примеров универсальной аппроксимации с использованием различных *почти совершенных* регрессионных и двухклассовых классификационных наборов данных, где мы поворачивает как емкостный диск, так и оптимизационный диск далеко вправо. Здесь "почти совершенный" означает очень мелко отобранный крупный набор данных (в отличие от совершенного бесконечно крупного набора данных). Случай, когда набор данных действительно бесконечно велик ($P = \infty$), теоретически потребовал бы бесконечной вычислительной мощности для того, чтобы минимизировать соответствующую стоимостную функцию.

Пример 11.4. Универсальная аппроксимация почти совершенных регрессионных данных

На рис. 11.14 проиллюстрирована универсальная аппроксимация почти совершенного регрессионного набора данных, состоящего из P = 10 000 равномерно отобранных точек из опорной синусоидальной функции, определенной на единичном интервале. В левом, среднем и правом столбцах рисунка показан результат подгонки к этим данным все большего числа соответственно многочленных, нейросетевых и древесных единиц. По мере увеличения числа единиц в каждом случае (сверху вниз) емкость каждой соответствующей модели увеличивается, что позволяет получить более оптимальную универсальную аппроксимацию.

Обратите внимание на то, что здесь для хорошего представления данных требуется гораздо меньше единиц как многочленных, так и нейросетевых аппроксиматоров по сравнению с дискретными пневыми единицами. Это обусловлено тем, что члены первой из упомянутых в первую очередь ближе напоминают гладкую синусоидаль-

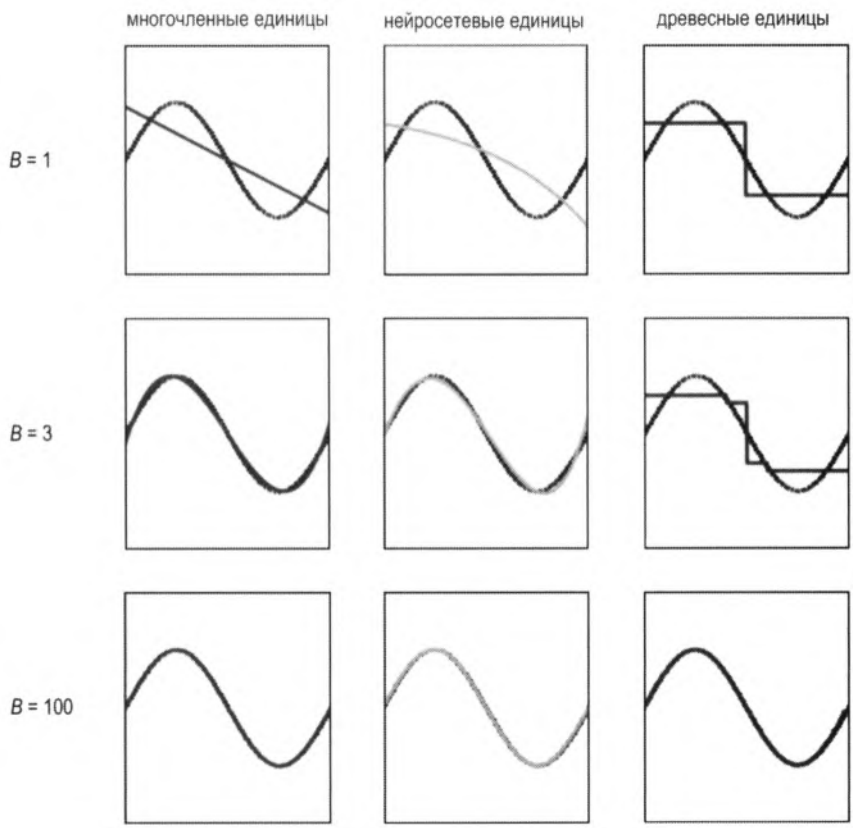


Рис. 11.14. Иллюстрация примера 11.4. Модели, построенные из многочленов (*слева*), нейронных сетей (*в центре*) и пней (*справа*), соответствуют почти совершенному регрессионному набору данных. При визуализации древесных моделей в правом столбце мы связали каждый дискретный шаг вертикальной линией только для целей визуализации. По мере того как все больше и больше единиц добавляется в модель, каждая из них становится способной вписываться в набор данных с возрастающей прецизионностью

ную функцию, которая генерировала данные. В общем случае это явление справедливо: в то время как любой тип универсального аппроксиматора может быть использован для аппроксимации совершенного (или почти совершенного) набора данных настолько близко, насколько это необходимо, некоторые универсальные аппроксиматоры требуют меньше единиц, чем другие, в зависимости от контура опорной функции, которая генерировала набор данных.

Пример 11.5. Универсальная аппроксимация почти совершенных классификационных данных

На рис. 11.15, *сверху* показаны четыре примера почти совершенных двухклассовых классификационных данных с точки зрения персептрона (т. е. сверху), каждый из

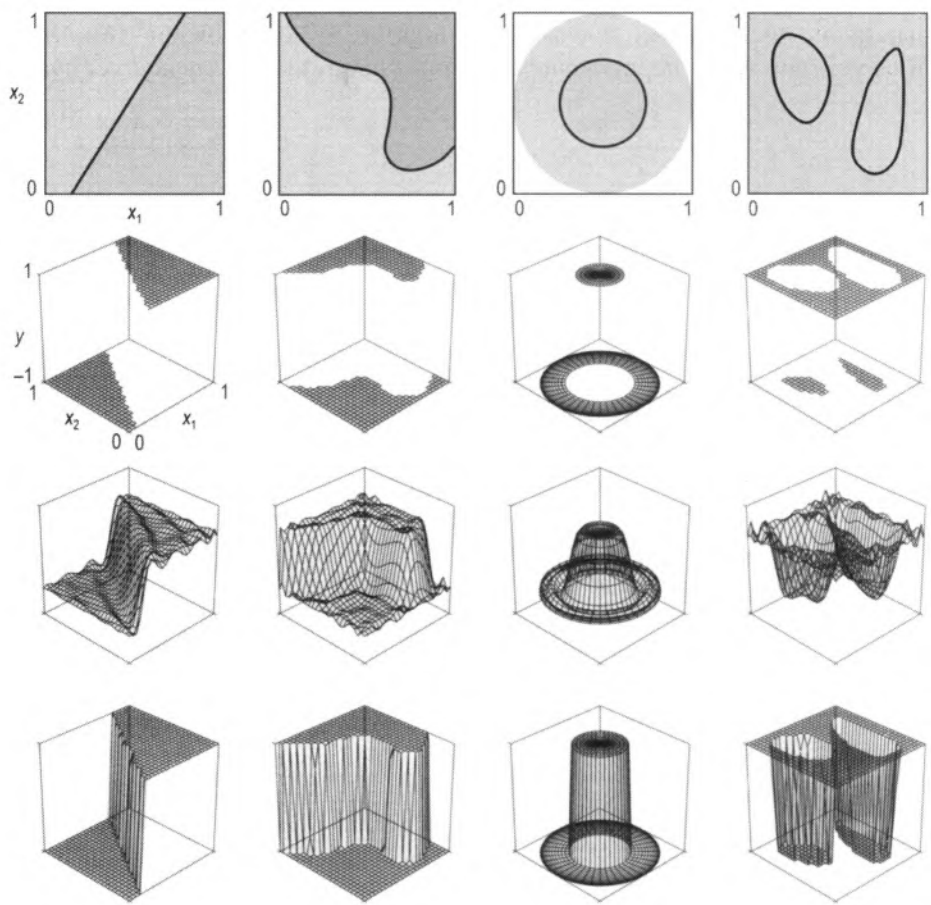


Рис. 11.15. Иллюстрация примера 11.5. Четыре экземпляра почти совершенных двухклассовых классификационных данных (*первый ряд*). Соответствующие данные показаны с другой (т. е. регрессионной) точки зрения (*второй ряд*). Соответствующие многочленные аппроксимации для каждого набора данных с $B = 30$ единицами в каждом случае с использованием стоимости наименьших квадратов (*третий ряд*) и стоимости мягкого максимума (*четвертый ряд*). Аппроксимации, показанные в последнем ряду, пропускаются через функцию tanh перед визуализацией

которых состоит из P = 10 000 точек. В каждом случае точки, окрашенные розовым цветом, имеют значение +1 метки, а точки, окрашенные голубым цветом, имеют значение –1 метки. Во втором ряду этого рисунка нанесены соответствующие наборы данных, показанные с регрессионной точки зрения (т. е. сбоку).

Каждый из этих почти совершенных наборов данных может быть эффективно аппроксимирован с помощью любого из трех категорий универсальных аппроксиматоров, рассмотренных в *разд. 11.2.3* при условии, что емкость каждой модели достаточно увеличена и соответствующие параметры надлежаще настроены. В третьем и четвертом рядах рисунка показана результирующая подгонка при задействовании $B = 30$ многочленных аппроксиматоров с использованием соответственно стоимости наименьших квадратов и мягкого максимума.

11.3. Универсальная аппроксимация реальных данных

В предыдущем разделе мы видели, как нелинейная модель из структурных единиц одного-единственного универсального аппроксиматора может быть построена так, чтобы она плотно аппроксимировала любой *совершенный набор данных*, если мы достаточно увеличим его емкость и надлежаще настроим модельные параметры, минимизируя соответствующую стоимостную функцию. В этом разделе мы выясним, как универсальная аппроксимация переносится на случай *реальных данных*, т. е. данных, которые конечны по размеру и потенциально зашумлены. Затем мы узнаем о новом измерительном инструменте, именуемом *валидационной ошибкой*, который позволит нам эффективно задействовать универсальные аппроксиматоры с реальными данными.

11.3.1. Прототипические примеры

Здесь мы разведаем применения универсальных аппроксиматоров для представления реальных данных, рассмотрев два простых примера: регрессионный и двухклассовый классификационный наборы данных. Проблемы, с которыми мы столкнемся в этих примерах, отражают сложности, которые подстерегают нас в целом при задействовании моделей на основе универсального аппроксиматора с реальными данными, независимо от типа проблемы.

Пример 11.6. Универсальная аппроксимация реальных регрессионных данных

В этом примере мы иллюстрируем использование универсальных аппроксиматоров на реальном регрессионном наборе данных, основанном на почти совершенных синусоидальных данных, представленных в примере 11.4. Для того чтобы просимулировать реальную версию этого набора данных, мы случайно отобрали $P = 21$ его точек и добавили немного случайного шума в выход (т. е. компонент y) каждой точки (рис. 11.16).

На рис. 11.17 показана полностью настроенная нелинейная подгонка модели, задействующей многочленные (верхний ряд), нейросетевые (средний ряд) и древесные (нижний ряд) структурные единицы, к этим данным. Обратите внимание, как

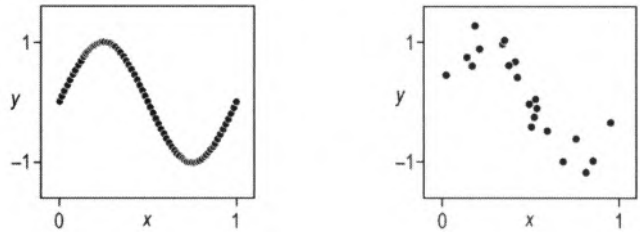


Рис. 11.16. Иллюстрация примера 11.6. Исходный почти совершенный синусоидальный набор данных из примера 11.4 (*слева*). Реальный регрессионный набор данных, сформированный путем добавления случайного шума в выход малого подмножества точек почти совершенного набора данных (*справа*)

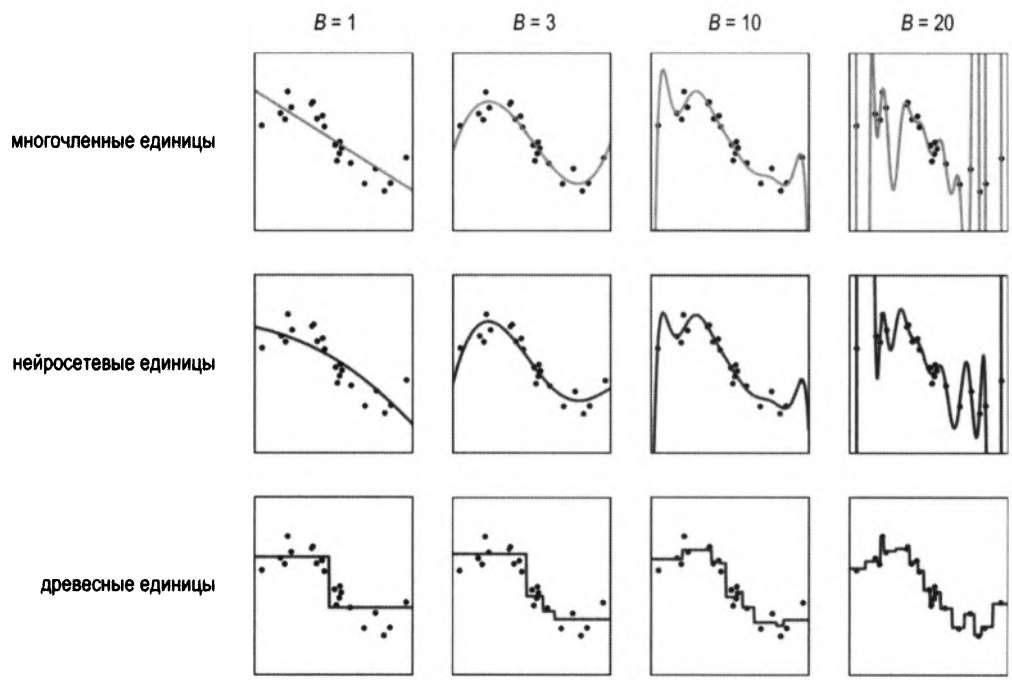


Рис. 11.17. Иллюстрация примера 11.6

в случае с каждым универсальным аппроксиматором все три модели *недообучены* к данным при использовании только $B = 1$ единиц в каждом случае (левая крайняя графа). Недообучение к данным является прямым следствием использования низкоемкостных моделей, производящих подгонки, *недостаточно сложные* для опорных данных, которые они стремятся аппроксимировать. Также обратите внимание на то, как каждая модель совершенствуется по мере увеличения B, но только до определенного момента, после которого каждая *настроенная* модель становится чересчур сложной и начинает выглядеть довольно необузданной, и очень сильно отличаясь от синусоидального явления, которое сгенерировало данные изначально. Это особенно заметно в многочленном и нейросетевом случаях, когда к моменту достижения $B = 20$ единиц (крайняя правая графа) обе модели чрезвычайно осцил-

лируют и становятся чересчур сложными. Такие *переобученные* модели, хотя и хорошо представляют текущие данные, очевидно, будут способствовать слабым предсказателям будущих данных, генерируемых тем же самым процессом.

На рис. 11.18 построены графики нескольких многочленных моделей, показанных на рис. 11.17, вместе с соответствующим стоимостным значением наименьших

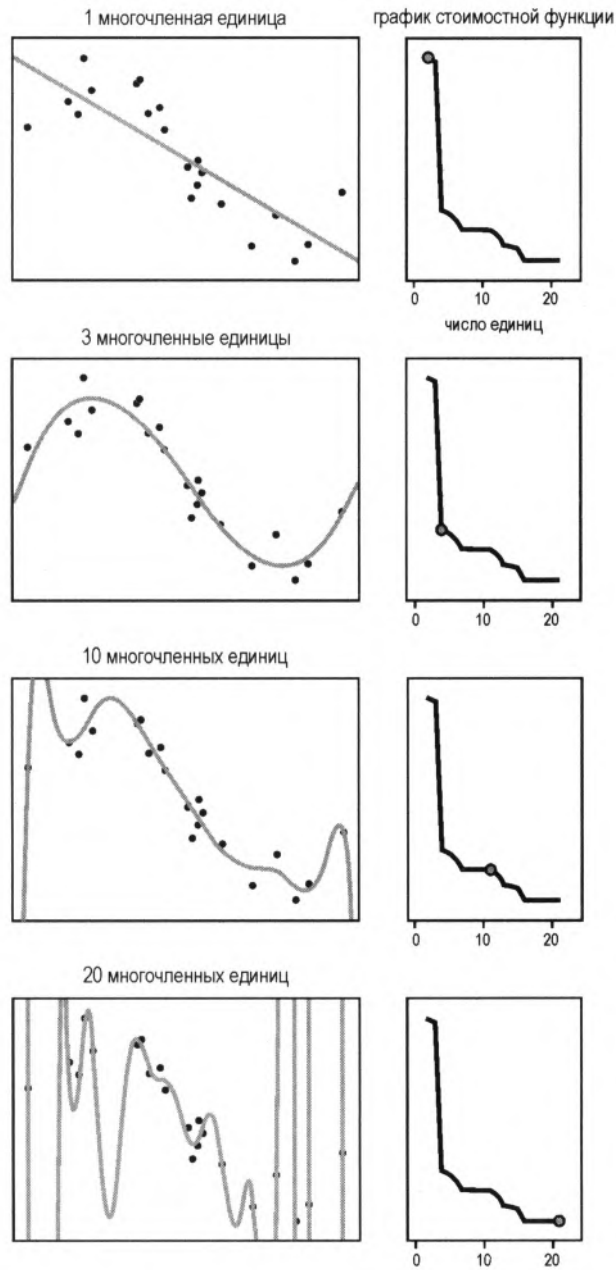


Рис. 11.18. Иллюстрация примера 11.6

квадратов для каждой из них. Добавляя больше многочленных структурных единиц, мы увеличиваем емкость нашей модели, и после оптимизации каждой модели до полного завершения результирующие настроенные модели достигают все более низкого стоимостного значения. Однако результирующая подгонка, обеспечиваемая каждой полностью настроенной моделью (после некоторой точки), становится слишком сложной и начинает *ухудшаться* с точки зрения того, как она представляет общее регрессионное явление. В качестве измерительного инструмента стоимостное значение говорит нам только о том, насколько хорошо настроенная модель вписывается в *тренировочные данные*, но не способна ничего нам сообщить, когда наша настроенная модель становится слишком сложной.

Пример 11.7. Универсальная аппроксимация реальных классификационных данных

В этом примере проиллюстрировано применение моделей на базе универсального аппроксиматора на реальном двухклассовом классификационном наборе данных, основанном на почти совершенном наборе данных из примера 11.5. Здесь мы просимулировали реалистичную версию этих данных, случайно отобрав $P = 99$ его точек и добавив немного классификационного шума, перевернув метки пяти из этих точек, как показано на рис. 11.19.

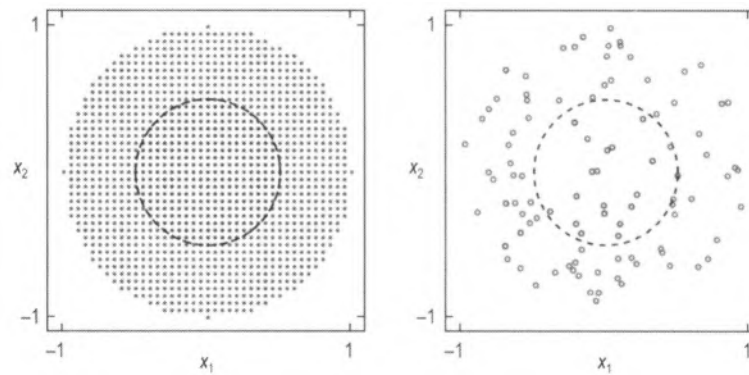


Рис. 11.19. Иллюстрация примера 11.7. Исходный почти совершенный классификационный набор данных из примера 11.5 с истинной округлой границей, используемой для генерирования данных, показанных пунктирной линией черного цвета (это та граница, которую мы надеемся восстановить с помощью классификации) (*слева*). Реальный набор данных, сформированный из шумного подмножества этих точек (*справа*)

На рис. 11.20 показаны нелинейные границы решения, обеспечиваемые полностью настроенными моделями, задействующими многочленную (верхний ряд), нейросетевую (средний ряд) и древесную структурные единицы (нижний ряд). В начале, где $B = 2$ (крайняя левая графа), все три настроенные модели недостаточно сложны и, таким образом, *недообучены* к данным, обеспечивая классификацию, которая во всех случаях просто классифицирует все пространство как принадлежащее синему классу. После этого и до определенного момента граница решения, предоставляемая каждой моделью, совершенствуется по мере добавления все большего числа

единиц, причем $B = 5$ многочленных единиц, $B = 3$ нейросетевые единицы и $B = 5$ древесных единиц обеспечивают разумные аппроксимации желаемой округлой границы решения. Однако вскоре после того, как мы достигнем этих количеств единиц, каждая настроенная модель становится слишком сложной и *переобученной* к тренировочным данным, а граница решения каждой из них отклоняется от истинной округлой границы, центрированной в начале координат. Как и в случае с регрессией в примере 11.6, в случае классификации возникают проблемы как недообучения, так и переобучения, независимо от типа используемого универсального аппроксиматора.

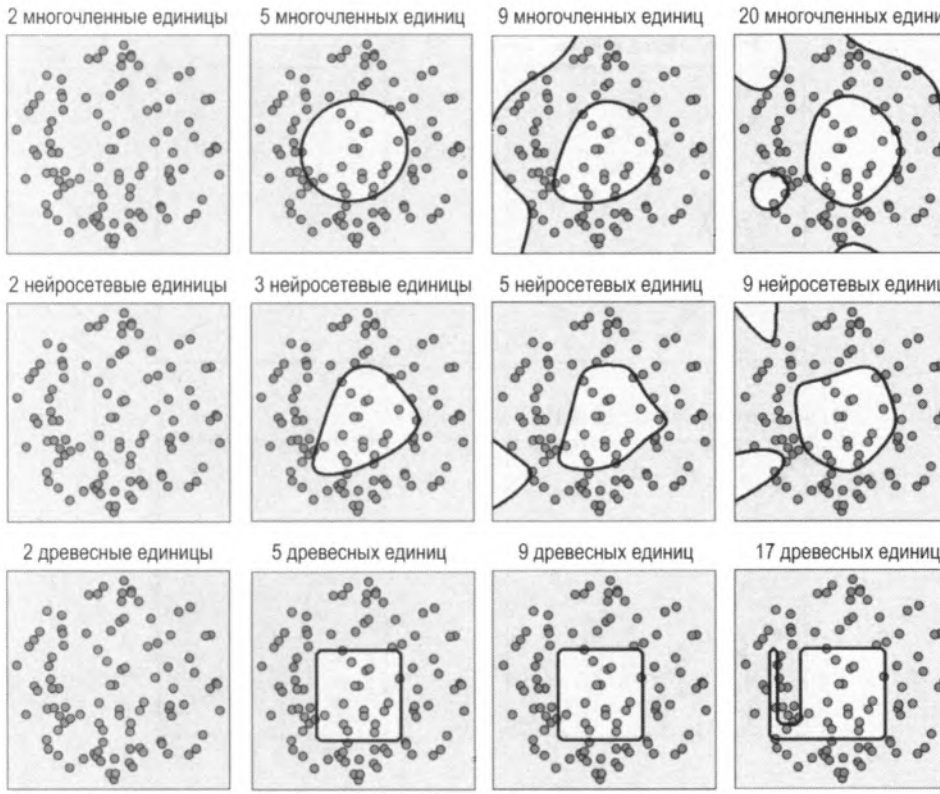


Рис. 11.20. Иллюстрация примера 11.7

На рис. 11.21 построены графики нескольких нейросетевых моделей, показанных в среднем ряду рис. 11.20, вместе с соответствующим значением стоимости двухклассового мягкого максимума, которого каждая из них достигает. Как и ожидалось, увеличение емкости модели за счет добавления большего числа нейросетевых единиц всегда (при настройке параметров каждой модели путем полной минимизации) *уменьшает* значение стоимостной функции (так же как и при совершенных или почти совершенных данных). Однако результирующая классификация после определенного момента на самом деле становится *хуже* с точки зрения того, как она (усвоенная граница решения) представляет общее классификационное явление.

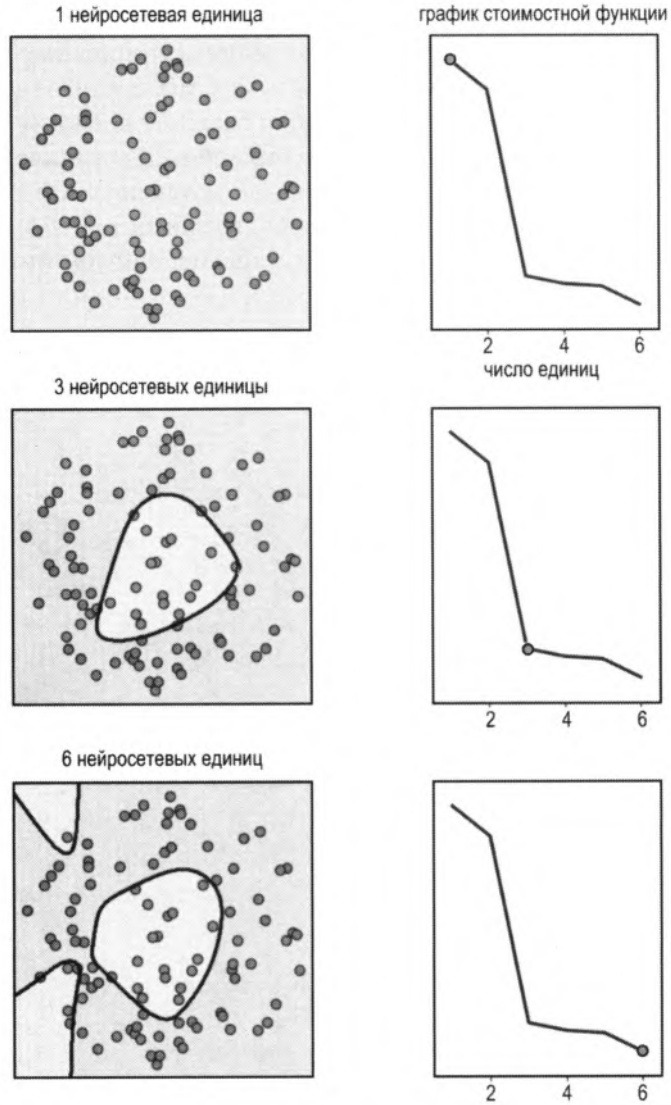


Рис. 11.21. Иллюстрация примера 11.7

В итоге примеры 11.6 и 11.7 показывают, что, в отличие от случая с совершенными данными, при задействовании моделей на основе универсального аппроксиматора с реальными данными мы должны быть осторожны в плане установки емкости нашей модели, а в плане качества настройки ее параметров посредством оптимизации связанной с ней стоимости. Эти два простых примера также показывают, как стоимость, связанная с тренировочными данными (также именуемая *тренировочной ошибкой*), не дотягивает до "звания" надежного инструмента для измерения того, насколько хорошо настроенная модель представляет явление, лежащее в основе реального набора данных. Обе эти трудности возникают в общем случае и подробнее обсуждаются далее.

11.3.2. Емкостный и оптимизационный наборные диски повторно

Прототипические примеры, описанные в *разд. 11.3.1*, иллюстрируют то, как с реальными данными мы не можем (в противовес случаю совершенных данных) просто взять и повернуть наши емкостный и оптимизационный наборные диски (представленные в *разд. 11.2.4*) до конца вправо, т. к. это приводит к чрезмерно сложным моделям, которые не могут хорошо представлять опорное явление, генерирующее данные. Обратите внимание, что мы контролируем *сложность* настроенной модели (или, грубо говоря, то, насколько "волнистой" является результирующая подгонка в настроенной модели) только *косвенно* тем, как мы устанавливаем наши емкостный и оптимизационный диски, и априори не очевидно, как мы должны устанавливать их одновременно, чтобы достичь верного уровня модельной сложности для конкретного набора данных. Однако мы можем несколько облегчить эту проблему настройки наборного диска, зафиксировав один из двух дисков и настроив другой. Установка одного наборного диска в крайнее правое положение наделяет другой диск исключительным контролем над сложностью настроенной модели (и превращает его, грубо говоря, в *сложностный диск*, описанный в *разд. 11.1.3*). То есть, фиксируя один из двух дисков в крайнем правом положении, по мере того как мы поворачиваем незафиксированный диск слева направо, мы увеличиваем сложность нашей окончательной настроенной модели. В этом состоит общий принцип при применении моделей на основе универсального аппроксиматора к реальным данным, который не проявляется в случае совершенных данных.

Для того чтобы получить более четкое интуитивное понимание об этом принципе, сначала предположим, что мы повернули наш оптимизационный диск до конца вправо (это означает, что независимо от набора данных и модели, которую мы используем, мы всегда настраиваем ее параметры, минимизируя соответствующую стоимостную функцию до полного завершения). Затем с совершенными данными, как показано рис. 11.22, *сверху*, когда мы поворачиваем наш емкостный диск слева направо (например, добавляя больше структурных единиц), результирующая настроенная модель обеспечивает все более и более оптимальное представление данных.

Однако с реальными данными (рис. 11.22, *снизу*), если начать с установки нашего емкостного диска в крайнее левое положение, то результирующая настроенная модель будет недостаточно сложной для явления, лежащего в основе наших данных. Мы говорим, что такая настроенная модель *недообучена*, т. к. она плохо вписывается в данные[7]. Поворот емкостного диска слева направо *увеличивает* сложность каждой настроенной модели, обеспечивая все более и более оптимальное представление данных и лежащего в их основе явления. Однако при некотором положении емкостного диска наступает момент, когда соответствующая настроенная модель

[7] Обратите внимание, что в то время как простое визуальное изображение здесь иллюстрирует недообученную модель как линейную ("неволнистую") функцию — что довольно часто встречается на практике — недообученная модель может быть довольно "волнистой". Независимо от контура, который принимает настроенная модель, мы говорим, что она недообученна, если она плохо представляет тренировочные данные, т. е. если она имеет высокую тренировочную ошибку.

становится *слишком сложной*. Действительно, после этого момента, когда сложность каждой настроенной модели дико неуместна для рассматриваемого явления, мы говорим, что начинается *переподгонка*. Эта терминология используется из-за того, что, хотя очень сложные модели отлично вписываются в имеющиеся данные, они делают это за счет плохого представления лежащего в их основе явления. По мере дальнейшего поворота нашего емкостного диска вправо результирующие настроенные модели будут становиться все более сложными и все менее репрезентативными для истинного опорного явления.

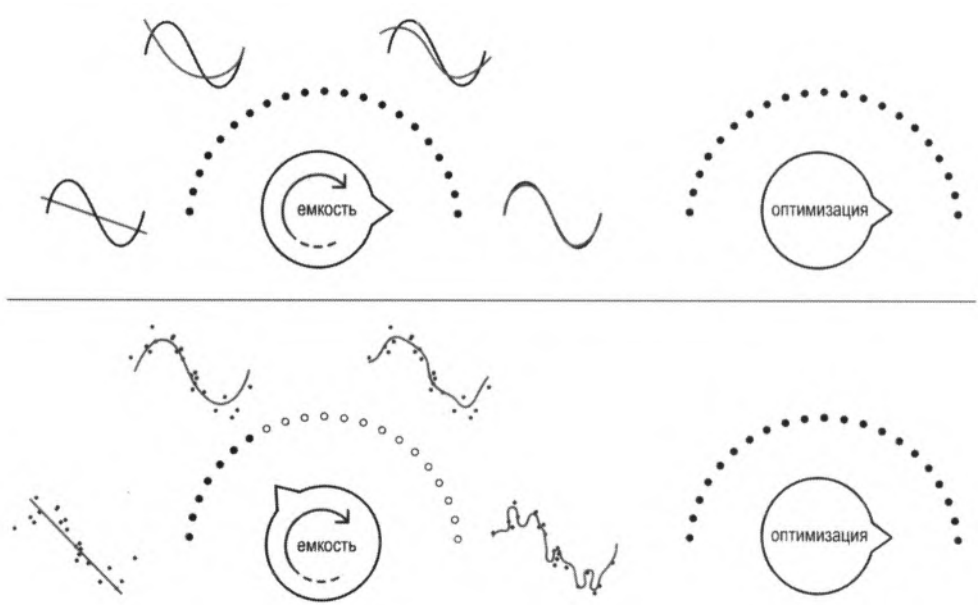


Рис. 11.22. Если мы устанавливаем наш оптимизационный диск в крайнее правое положение с совершенными данными, то по мере увеличения емкости нашей модели, поворачивая емкостный диск слева направо, соответствующее представление будет становиться все лучше и лучше (*сверху*). С реальными данными наблюдается аналогичный эффект; однако здесь, когда мы поворачиваем емкостный диск дальше вправо, каждая настроенная модель будет тяготеть к тому, чтобы становиться все сложнее и сложнее, в конечном итоге оказываясь переподогнанной к данным (*снизу*)

Теперь предположим, что мы установили наш емкостный диск в крайнее правое положение, используя высокоемкостную модель, и настраиваем ее параметры, поворачивая наш оптимизационный диск очень медленно слева направо. В случае совершенных данных (рис. 11.23, *сверху*) этот подход создает настроенные модели, которые с каждым разом всё точнее представляют явление. С другой стороны, с реальными данными (рис. 11.23, *снизу*), если начать с установки нашего оптимизационного диска в крайнее левое положение, то это будет, как правило, приводить к низкосложностным *недообученным* настроенным моделям. По мере поворота нашего оптимизационного диска слева направо, при выполнении шагов конкретной схемы локальной оптимизации, наша соответствующая модель будет тяготеть

к увеличению сложности, улучшая свое представление данных. Это улучшение продолжается только до того момента, пока наша соответствующая настроенная модель не станет слишком сложной для явления, лежащего в основе данных, и, следовательно, начинается *переподгонка*. После этого настроенные модели, возникающие в результате поворота диска оптимизации вправо, будут слишком сложны для адекватного представления явления, лежащего в основе данных.

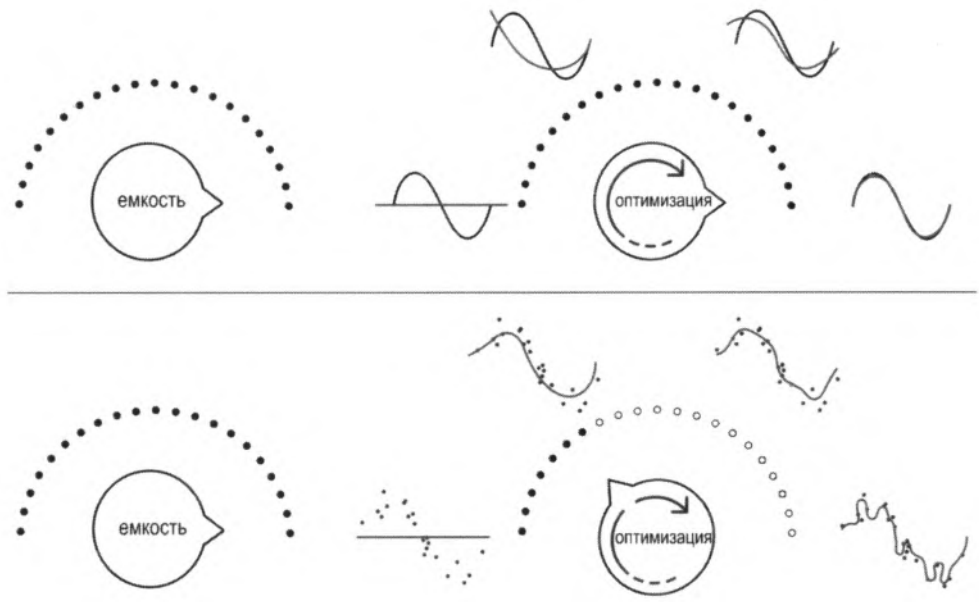


Рис. 11.23. Если мы установим наш емкостный диск в крайнее правое положение с совершенными данными, то по мере увеличения степени, с которой мы оптимизируем нашу модель, поворачивая оптимизационный диск слева направо (начиная с крайнего левого положения, где для простоты мы исходим из того, что все модельные параметры инициализируются в нуле), соответствующее представление становится все лучше и лучше (*сверху*). С реальными данными происходит аналогичный эффект, но только до определенного момента, когда начинается переподгонка (*снизу*)

11.3.3. Мотивирование нового измерительного инструмента

То, как мы устанавливаем наши емкостный и оптимизационный наборные диски, чтобы достичь окончательной настроенной модели, которая обладает *только нужной степенью сложности* для имеющегося набора данных, является главной проблемой, с которой мы сталкиваемся при задействовании моделей на основе универсального аппроксиматора с реальными данными. В примерах 11.6 и 11.7 мы увидели, как тренировочная ошибка не фиксирует момент, когда настроенная модель получает достаточную *сложность* соответственно для задач регрессии и двухклассовой классификации — факт, который в более общем плане справедлив для всех задач нелинейного машинного обучения. Если мы не можем полагаться на тренировочную ошибку, сигнализирующую нам об остановке при надлежащей степени

сложности, необходимой для решения реальных нелинейных задач машинного обучения, то какой измерительный инструмент мы должны использовать вместо этого? Внимательное изучение рис. 11.24 дает ответ!

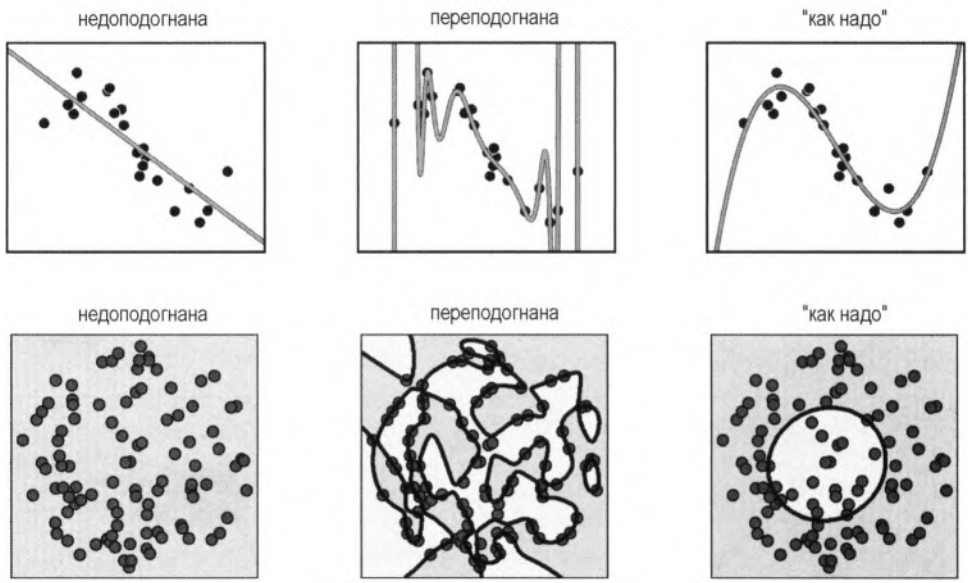


Рис. 11.24. Три модели для регрессионного набора данных из примера 11.6: недообученная модель (*слева сверху*), переобученная модель (*в центре сверху*) и модель "как надо" (*справа сверху*). Три модели для двухклассового классификационного набора данных из примера 11.7: недообученная модель, которая просто классифицирует все как часть синего класса (*слева снизу*), переобученная модель (*в центре снизу*) и подгонка "как надо" (*справа снизу*)

В верхнем ряду этого рисунка показаны три экземпляра моделей, представленных для игрушечного регрессионного набора данных из примера 11.6: полностью настроенная с низкой сложностью (и недообученная) линейная модель (*слева*), высокосложностная (и переподогнанная) 20-степенная многочленная модель (*в центре*) и трехстепенная многочленная модель (*справа*), которая вписывается в данные и опорное генерирующее их явление, "как надо". Что общего между недообученным и переобученным шаблонами, чего нет у модели "как надо"?

Внимательно рассмотрев две левые панели рисунка, мы увидим, что общая проблема как с недообученными, так и с переобученными моделями заключается в том, что, хотя они различаются качеством представления данных, которые у нас *уже есть*, они обе не будут способны адекватно представлять *новые данные* с помощью того же процесса, с помощью которого были получены текущие данные. Другими словами, мы не доверяем ни одной из моделей в предсказании выхода вновь прибывшей входной точки. Полностью настроенная модель "как надо" не страдает от этой проблемы, поскольку она близко аппроксимирует своего рода волнистый синусоидальный шаблон, лежащий в основе данных, и в результате будет хорошо работать в качестве предсказателя для будущих точек данных.

Та же история повторяется на рис. 11.24, *снизу* с нашим двухклассовым классификационным набором данных, использованным ранее в примере 11.7. Здесь показаны полностью настроенная низкосложностная (и недообученная) линейная модель (*слева*), высокосложностная (и переобученная) 20-степенная многочленная модель (*в центре*) и двухстепенная многочленная модель "как надо" (*справа*). Как и в регрессионном случае, недообученная и переобученная модели не способны адекватно представлять опорное явление, генерирующее наши текущие данные, и, как следствие, не смогут адекватно предсказывать значения меток для *новых данных*, генерируемых с помощью того же самого процесса, которым были созданы текущие данные.

В итоге с помощью рассмотренных здесь простых регрессионного и классификационного примеров мы можем приблизительно квалифицировать низкорезультативные модели как такие, которые не позволят нам делать точные предсказания в отношении будущих данных. Но как квантифицировать то, что мы получим в будущем? Рассмотрим это далее.

11.3.4. Валидационная ошибка

Теперь у нас есть неоформальный диагноз для проблематичной результативности недообученной/переобученной моделей: такие модели не точно отражают новые данные, которые мы могли бы получить в будущем. Но как использовать этот диагноз? Мы, конечно же, не имеем доступа к каким-либо новым данным, которые мы получим в будущем. Для того чтобы сделать это понятие полезным, нам нужно перевести его в величину, которую мы всегда сможем измерить, независимо от набора данных/задачи, которую мы решаем, или типа модели, которую мы задействуем.

Если коротко, то универсальный способ сделать это состоит в *подделке*: мы просто удаляем случайную часть наших данных и трактуем ее как "новые данные, которые мы могли бы получить в будущем", как абстрактно показано на рис. 11.25. Другими словами, мы вырезаем случайный фрагмент набора данных, который у нас есть, тренируем наши отобранные модели только на той части данных, которая остается, и *валидируем* (т. е. подвергаем контрольной проверке) результативность каждой модели на этом случайно удаленном фрагменте "новых" данных. Случайная часть

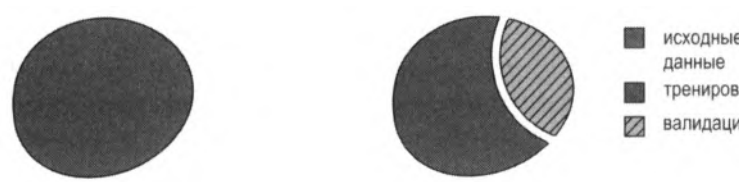


Рис. 11.25. Разбивка данных на тренировочный и валидационный наборы. Исходные данные (*слева*) в виде полностью круглой массы случайным образом разбиваются (*справа*) на два непересекающихся множества. Меньший фрагмент, обычно от 1/10 до 1/3 исходных данных, берется в качестве валидационного набора, а оставшийся — в качестве тренировочного набора

данных, которую мы удаляем для того, чтобы валидировать нашу модель, обычно называется *валидационными данными* (или контрольно-проверочным набором), а оставшаяся часть, которую мы используем для тренировки моделей, таким же образом называется *тренировочными данными* (или тренирующим набором). Модель, обеспечивающая наименьшую ошибку на валидационных данных, т. е. наименьшую валидационную ошибку, считается наилучшим вариантом из отобранных натренированных моделей. Как мы увидим, валидационная ошибка (в отличие от тренировочной ошибки) на самом деле является тем измерительным инструментом, который должным образом подходит для определения качества модели относительно опорного генерирующего данные явления, которое мы хотим уловить.

Не существует точного правила в отношении того, какая часть имеющегося набора данных должна быть оставлена для валидации. На практике обычно от 1/10 до 1/3 данных закрепляется за валидационным набором. Вообще говоря, чем крупнее набор данных и/или он больше представляет истинное явление, из которого отбираются данные, тем крупнее часть исходных данных, которая может быть отнесена к валидационному набору (например, 1/3). Интуиция подсказывает, что если данные достаточно многочисленны/репрезентативны, то тренировочный набор попрежнему точно представляет лежащее в его основе явление, даже после удаления относительно крупного валидационного набора данных. И наоборот, в целом для меньших или менее репрезентативных (т. е. более шумными или слабо распределенными) наборов данных мы обычно берем для валидации меньшую часть (например, 1/10), поскольку относительно крупный тренировочный набор должен сохранять ту малую информацию об опорном явлении, которая была захвачена исходными данными, и мало данных может быть выделено для валидации.

11.4. Наивная перекрестная валидация

Валидационная ошибка дает нам конкретный способ не только измерить результативность отдельной настроенной модели, но и, что важнее, позволяет сравнить эффективность нескольких настроенных моделей различной степени сложности. Произведя тщательный поиск среди набора моделей различной сложности, мы можем легко определить лучшую из них, ту, которая обеспечивает минимальную ошибку в валидационном наборе. Это сравнение моделей, именуемое *перекрестной валидацией*, или иногда *модельным поиском* или *отбором моделей*, является базисом усвоения признаков, поскольку она обеспечивает систематический способ усвоения (в отличие от *инженерии*, как подробно описано в *главе 10*) надлежащей формы, которую нелинейная модель должна принимать для конкретного набора данных.

В этом разделе мы введем понятие *наивной* перекрестной валидации. Она заключается в поиске среди множества моделей варьирующейся емкости, причем каждая модель полностью оптимизирована над тренировочным набором в поисках варианта с минимальной валидационной ошибкой. Хотя наивная валидация является простой в принципе и в имплементации, она в целом обходится очень дорого (с вычислительной точки зрения) и часто приводит к довольно грубому модельному поиску,

в результате которого можно пропустить (или "проскочить через") идеальную степень сложности, желаемую для конкретного набора данных.

11.4.1. Картина в целом

Первый *организованный* подход, который можно было бы принять для определения идеальной степени сложности для конкретного набора данных, заключается в том, чтобы сначала выбрать один универсальный аппроксиматор (например, один из тех простых образчиков, описанных в *разд. 11.2.3*) и построить множество из M моделей общей формы, приведенной в уравнении (11.15), варьируя значение B от 1 до M последовательно:

$$\begin{aligned}
&\text{model}_1(\mathbf{x}, \Theta_1) = w_0 + f_1(\mathbf{x})w_1;\\
&\text{model}_2(\mathbf{x}, \Theta_2) = w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2;\\
&\vdots\\
&\text{model}_M(\mathbf{x}, \Theta_M) = w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + \ldots + f_M(\mathbf{x})w_M;
\end{aligned} \tag{11.16}$$

Этот набор моделей, который мы можем компактно обозначить как $\{\text{model}_m(\mathbf{x}, \Theta_m)\}_{m=1}^{M}$ (или даже компактнее просто как $\{\text{model}_m\}_{m=1}^{M}$), где множество Θ_m состоит из всех этих параметров m-й модели, естественным образом увеличивается в *емкости* от $m = 1$ до $n = M$ (как впервые описано в *разд. 11.2.2*). Если *оптимизировать* каждую из этих моделей до полного завершения, то они также (как обсуждается в *разд. 11.3.2*) увеличатся с точки зрения их *сложности* в отношении тренировочных данных. Отсюда следует, если мы сначала разделим наши исходные данные случайно на тренировочную и валидационную части, как описано в *разд. 11.3.4*, и измерим ошибку всех M полностью натренированных моделей на каждой части данных, то сможем очень легко определить, какая из M моделей обеспечивает идеальную степень сложности для набора данных в целом, отыскав ту, которая достигает минимальной валидационной ошибки.

На рис. 11.26, *сверху* показан обобщенный тип тренировочной (синим цветом) и валидационной (оранжевым цветом) ошибок, которые мы находим на практике в результате следования этой схеме наивной перекрестной валидации. Горизонтальная ось этого графика показывает сложность каждой из наших M полностью оптимизированных моделей, где выходные данные на вертикальной оси обозначают уровень ошибки. Как видно из рисунка, наши модели низкой сложности *недообучены* к данным, что отражается в их высоких тренировочной и валидационной ошибках. По мере дальнейшего увеличения модельной сложности полностью оптимизированные модели достигают меньшей тренировочной ошибки, поскольку увеличение сложности модели позволяет нам постоянно улучшать качество представления тренировочных данных. Этот факт отражен в монотонно убывающей природе (синей) кривой тренировочной ошибки. С другой стороны, в то время как валидационная ошибка наших моделей будет стремиться сначала к уменьшению по мере увеличения сложности, этот тренд продолжается только до точки, где начинается *переподгонка* к тренировочным данным. Как только мы достигаем сложности модели, при которой наступает переподгонка к тренировочным данным, наша ва-

лидационная ошибка снова начинает увеличиваться, поскольку наша модель становится все менее и менее справедливым представлением "данных, которые мы могли бы получить в будущем", генерируемых тем же самым явлением.

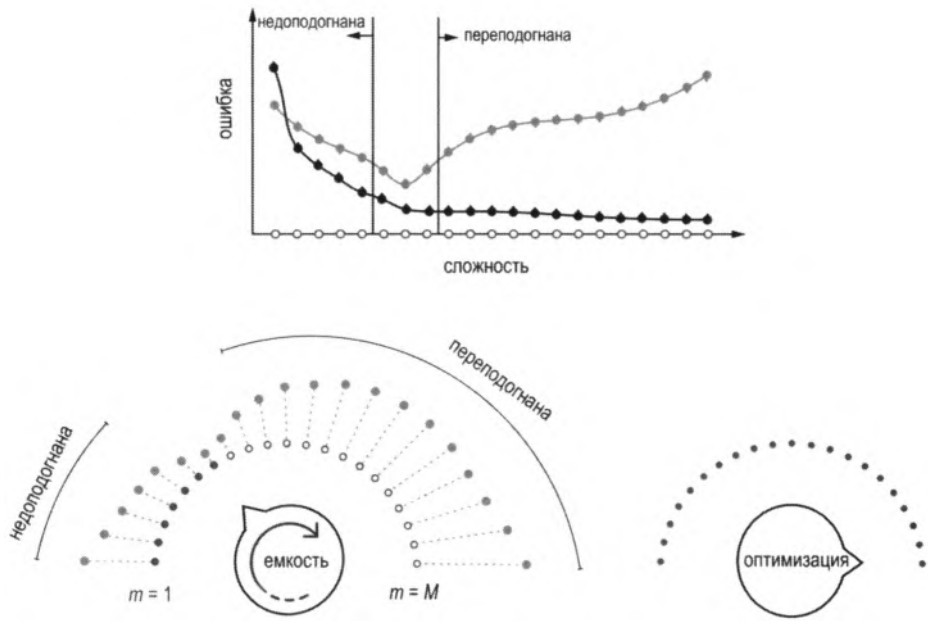


Рис. 11.26. Графики тренировочной (синим цветом) и валидационной (оранжевым цветом) ошибок прототипического набора данных, полученные в результате выполнения наивной перекрестной валидации (*сверху*). Здесь набор моделей, которые в целом увеличиваются по сложности относительно тренировочного набора, формируется путем полной оптимизации набора моделей возрастающей емкости. Низкосложностные модели недообученны к данным, что в типичной ситуации приводит к крупным тренировочной и валидационной ошибкам. В то время как тренировочная ошибка будет монотонно уменьшаться по мере увеличения сложности модели, валидационная ошибка тяготеет к уменьшению только до точки, где начинается переподгонка к тренировочным данным. Наивная перекрестная валидация с использованием нашей концептуализации в виде наборных дисков (*снизу*), где мы поворачиваем емкостный диск слева направо, осуществляя поиск среди диапазона моделей возрастающей емкости модели, которая минимизирует валидационную ошибку, удерживая при этом оптимизационный диск в крайнем правом положении (это значит, что мы оптимизируем каждую модель до полного завершения)

На практике обратите внимание на то, что в то время как тренировочная ошибка обычно следует монотонно убывающему тренду, показанному на рис. 11.26, *сверху*, валидационная ошибка может осциллировать вверх и вниз более одного раза в зависимости от проверяемых моделей. В любом случае мы определяем наилучшую полностью оптимизированную модель из множества, выбирая ту, которая *минимизирует* валидационную ошибку. Эта процедура часто именуется *решением компромисса между смещением и дисперсией*, поскольку она предусматривает определение модели, которая (в идеале) не является ни недообученной (или имеет высокое смещение), ни переобученной (или имеет высокую дисперсию).

В рис. 11.26, *снизу* мы подытоживаем этот наивный подход к перекрестной валидации с использованием концептуализации с емкостным/оптимизационным наборными дисками, впервые представленной в *разд. 11.2.2*. Здесь мы устанавливаем наш *оптимизационный* диск в крайнее правое положение (это означает, что мы оптимизируем каждую модель до полного завершения) и в диапазоне нашего множества из M моделей поворачиваем *емкостный* диск слева направо, начиная с $m = 1$ (слева) и заканчивая $m = M$ (до конца вправо), причем в каждой отметке диска значение m увеличивается на 1. Поскольку в этом случае *емкостный* диск приблизительно управляет сложностью модели, как наглядно показано на рис. 11.22, *снизу*, наш модельный поиск сводится к правильной установке этого диска, равной минимальной валидационной ошибке. Для того чтобы визуально обозначить, как это делается, мы поворачиваем прототипическую кривую валидационной ошибки, показанную на верхней панели рис. 11.26, по часовой стрелке вокруг емкостного диска. Мы тогда можем представить, что мы установили этот диск правильно (и автоматически) равным значению m, обеспечивающему минимальную валидационную ошибку.

Пример 11.8. Наивная перекрестная валидация и регрессия

В этом примере мы проиллюстрируем использование процедуры наивной перекрестной валидации для синусоидального регрессионного набора данных, впервые представленного в примере 11.6. Здесь мы используем 2/3 исходного набора из 21 точки данных для тренировки и остальные 1/3 — для валидации. Сравним множество моделей, являющихся многочленами степени $1 \le m \le 8$. Другими словами, m-я модель из нашего множества $\{\text{model}_m\}_{m=1}^{8}$ является одновходным m-степенным многочленом формы

$$\text{model}_m(\mathbf{x}, \Theta_m) = w_0 + xw_1 + x^2 w_2 + \ldots + x^m w_m. \qquad (11.17)$$

Обратите внимание на то, как это малое множество моделей естественным образом упорядочено в терминах нелинейной емкости, причем более низкостепенные модели имеют меньшую емкость, а более высокостепенные модели — бóльшую емкость.

На рис. 11.27 показана подгонка трех многочленных моделей на исходном наборе данных (первый ряд), тренировочных данных (второй ряд) и валидационных данных (третий ряд). Ошибки как на тренировочных (синим цветом), так и на валидационных (оранжевым цветом) данных показаны на нижнем рисунке для всех моделей. Обратите внимание, что валидационная ошибка находится на самом низком уровне, когда модель представляет собой многочлен четвертой степени. Разумеется, по мере того, как мы используем все больше многочленных единиц, перемещаясь слева направо на рисунке, более высокостепенные модели все лучше вписываются в тренировочные данные. Однако по мере того, как тренировочная ошибка продолжает уменьшаться, соответствующая валидационная ошибка начинает быстро расти по мере того, как соответствующие модели обеспечивают все более и более бедные представления валидационных данных (к моменту $m = 7$ валидацион-

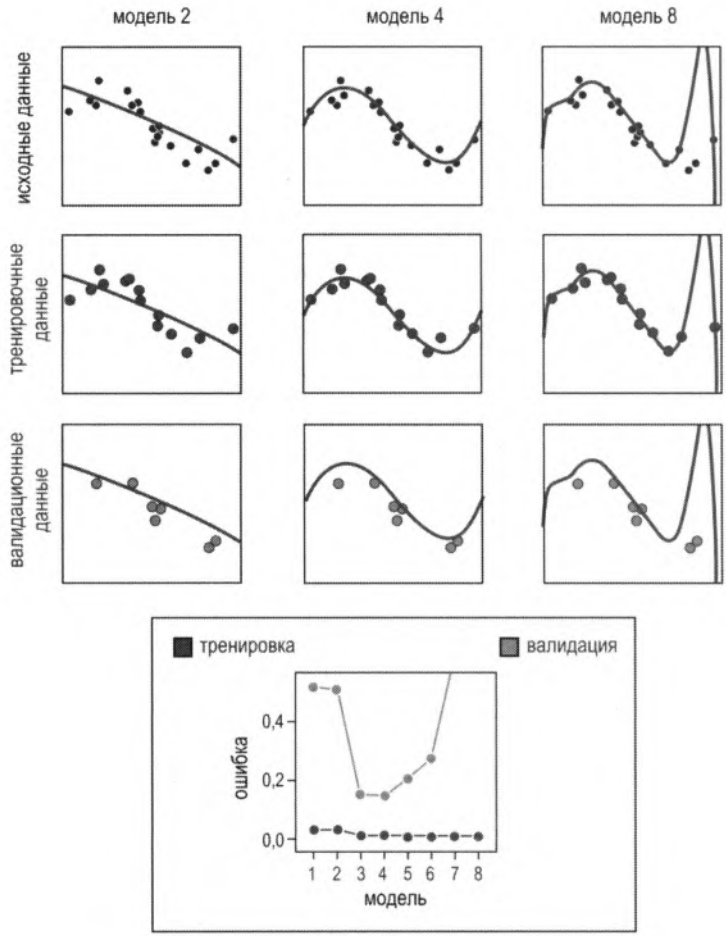


Рис. 11.27. Иллюстрация примера 11.8

ная ошибка становится настолько большой, что мы не строим ее график в том же окне, благодаря чему мы можем надлежаще различать другие значения ошибки).

Пример 11.9. Наивная перекрестная валидация и классификация

В этом примере мы иллюстрируем использование наивного подхода к перекрестной валидации на двухклассовом классификационном наборе данных, впервые показанном в примере 11.7. Здесь мы используем (приближенно) 4/5 исходного набора из 99 точек данных для тренировки и другие 1/5 — для валидации. Для простоты мы задействуем лишь малое множество многочленных моделей, имеющих степени $1 \le m \le 7$. Другими словами, m-я модель из нашего множества $\{\text{model}_m\}_{m=1}^{7}$ является многочленом степени m (с двумерным входом) формы

$$\text{model}_m(\mathbf{x}, \Theta_m) = w_0 + \sum_{0<i+j\le m} x_1^i x_2^j w_{i,j} \,. \qquad (11.18)$$

Эти модели также естественно упорядочены от низкой к высокой емкости по мере того, как мы увеличиваем степень m многочлена.

На рис. 11.28 показана подгонка трех моделей из $\{\text{model}_m\}_{m=1}^{7}$ вместе с исходными данными (первый ряд), тренировочными данными (второй ряд) и валидационными данными (третий ряд). Тренировочная и валидационная ошибки таким же образом показаны на нижнем рисунке для всех моделей. В случае классификации разумнее всего в качестве наших тренировочной/валидационной ошибок использовать число ошибок классифицирования, вычисленное над тренировочным/валидационным наборами, или некоторую функцию этих ошибок классифицирования (например, точность), в отличие от сырого вычисления стоимости классификации.

В этом случае двухстепенная многочленная модель ($m = 2$) обеспечивает наименьшую валидационную ошибку и, следовательно, наилучшую нелинейную границу

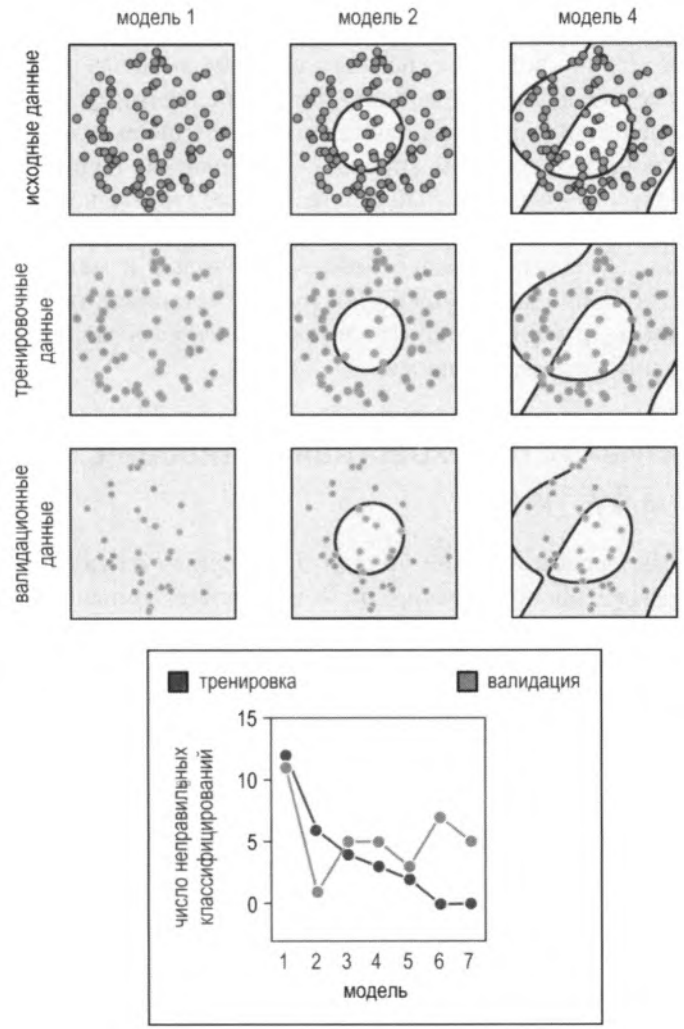


Рис. 11.28. Иллюстрация примера 11.9

решения для всего набора данных. Этот результат, конечно же, имеет и интуитивный смысл, т. к. мы определили округлую границу, используя модель этой формы, при конструировании таких признаков в примере 10.5 из *разд. 10.4.2*. По мере того как сложность возрастает, а тренировочные ошибки продолжают уменьшаться, наши модели становятся переобученными к тренировочным данным, при этом обеспечивая слабое решение для валидационных данных.

11.4.2. Проблемы с наивной перекрестной валидацией

Наивная перекрестная валидация достаточно хорошо работает для простых примеров, подобных описанным выше. Однако, поскольку этот процесс обычно предусматривает опробование диапазона моделей, где каждая модель *полностью* оптимизируется с нуля, наивная перекрестная валидация может обходиться очень дорого с вычислительной точки зрения. Более того, разница в емкости даже между соседними моделями (например, состоящими из m и $m+1$ единиц) может быть довольно большой, что приводит к огромным скачкам в диапазоне модельных сложностей, опробываемых на наборе данных. Другими словами, управление модельной сложностью посредством корректировки *емкостного* диска (при этом наш *оптимизационный* диск повернут до конца вправо — как показано на рис. 11.26, *снизу*) часто допускает только грубый модельный поиск, в результате которого можно легко "проскочить через" идеальную степень модельной сложности. Как мы увидим в следующих двух разделах, гораздо более устойчивые и мелкогранулированные схемы перекрестной валидации можно построить, установив наш *емкостный* диск в крайнее правое положение и контролируя *сложность* модели, тщательно настраивая наш *оптимизационный* диск.

11.5. Эффективная перекрестная валидация посредством бустинга

В этом разделе мы введем понятие *бустинга* — первую из двух фундаментальных парадигм для эффективной перекрестной валидации, описанных в этой главе. В отличие от наивной формы перекрестной валидации, описанной в предыдущем разделе, при перекрестной валидации на основе бустинга выполняем модельный поиск, беря одну высокоемкостную модель и оптимизируя ее *по одной структурной единице за раз*, что приводит к гораздо более эффективной процедуре перекрестной валидации. Хотя вместе с бустингом, в принципе, можно использовать любой универсальный аппроксиматор, этот подход часто применяется в качестве предпочтительного валидационного метода при задействовании древесных универсальных аппроксиматоров (как обсуждается подробнее в *разд. 14.7*).

11.5.1. Картина в целом

Базовый принцип, лежащий в основе перекрестной валидации на основе бустинга, заключается в поступательном построении высокоемкостной модели по *одной структурной единице за раз* при использовании единицы из одного-единственного

типа универсального аппроксиматора (например, один из этих простых образчиков описан в *разд. 11.2.3*):

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + ... + f_M(\mathbf{x})w_M \,. \qquad (11.19)$$

Мы делаем это последовательно за M *раундов*[8], где на каждом раунде добавляем в модель одну структурную единицу, полностью оптимизируя параметры только этой единицы вместе с соответствующим линейно-комбинационным весом, и оставляем эти параметры фиксированными в этих оптимально настроенных значениях навсегда. В качестве альтернативы мы можем рассуждать об этой процедуре как о том, что мы начинаем с высокоемкостной модели формы, приведенной в уравнении (11.19) и за M раундов оптимизируем параметры каждой единицы по одной за раз[9]. В любом случае выполнение бустинга таким образом создает последовательность из M настроенных моделей, которые обычно увеличиваются в *сложности* относительно тренировочного набора данных, который мы компактно обозначаем как $[\text{model}_m]_{m=1}^{M}$, где m-я модель состоит из m настроенных единиц. Поскольку одномоментно оптимизируется только одна структурная единица, бустинг, как правило, обеспечивает вычислительно эффективную мелкозернистую форму модельного поиска (по сравнению с наивной перекрестной валидацией).

Общая процедура бустинга тяготеет к созданию кривых тренировочной/валидационной ошибок, которые обычно выглядят так, как показано на рис. 11.29, *сверху*. Как и в случае с наивным подходом, подробно описанным в предыдущем разделе, здесь мы также, как правило, видим, что тренировочная ошибка уменьшается по мере увеличения m, в то время как валидационная ошибка начинается высоко там, где происходит недообучение, опускается до минимального значения (возможно, осциллируя больше, чем показанный здесь один раз) и снова поднимается вверх, когда начинается переподгонка.

Используя концептуализацию с емкостным/оптимизационным дисками, впервые представленную в *разд. 11.2.4*, мы можем рассуждать о бустинге, как если бы мы начинали с установки нашего *емкостного диска* в крайнее *правое* положение при некотором высоком значении (например, некотором большом значении M) и работали с оптимизационным диском, поворачивая его очень медленно слева направо (рис. 11.29, *снизу*). Как обсуждалось в *разд. 11.3.2* и визуально подытоживалось на рис. 11.23, *снизу*, с реальными данными эта общая конфигурация позволяет нашему оптимизационному диску управлять модельной сложностью. Другими словами, с этой конфигурацией наш оптимизационный диск (грубо говоря) становится своего рода сложностным диском с мелкой зернистостью, который мы намеревались построить в начале главы (*см. разд. 11.1.3*). Повернув наш *оптимизационный диск* до конца влево, мы начинаем наш поиск с настроенной низкосложностной модели (именуемой model_1), состоящей из одной структурной единицы универсального аппроксиматора, параметры которой полностью оптимизированы. По мере прохо-

---

[8] За $M + 1$ раундов, если мы включим w_0.

[9] Это одна из форм координатной оптимизации. См., например, *разд. 2.6*.

ждения раундов бустинга мы постепенно поворачиваем *оптимизационный диск* слева направо (здесь каждая отметка на оптимизационном диске обозначает полную оптимизацию одной дополнительной единицы), оптимизируя (до полного завершения) одну взвешенную единицу нашей исходной высокоемкостной модели из уравнения (11.19), в результате чего на m-м раунде наша настроенная модель (именуемая model_m) состоит из m индивидуально, но полностью настроенных единиц. Наша конечная цель при этом состоит в том, чтобы определить настройку оптимизации (т. е. определить соответствующее число настроенных структурных единиц), которая минимизирует валидационную ошибку.

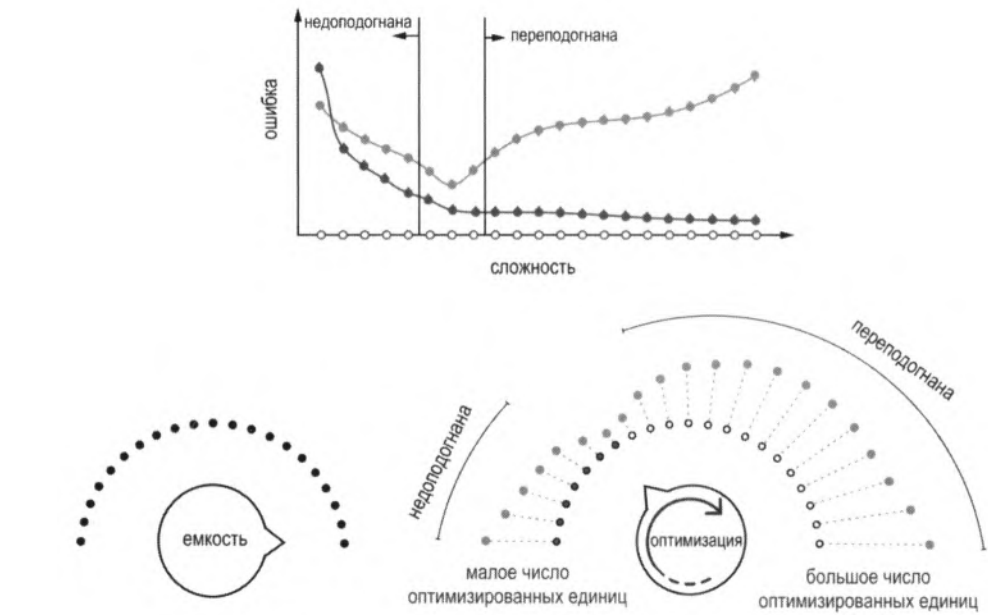


Рис. 11.29. Прототипические кривые тренировочной и валидационной ошибок, связанные с полным прогоном бустинга (*сверху*). В случае с бустингом мы фиксируем наш емкостный диск в крайнем правом положении и начинаем "крутить" наш оптимизационный диск, первоначально установленный в крайнее левое положение (*снизу*). Затем мы медленно поворачиваем наш оптимизационный диск слева направо, причем каждая отметка на оптимизационном диске обозначает полную оптимизацию одной добавочной единицы модели, увеличивая сложность каждой последующей модели, созданной относительно тренировочного набора

Независимо от того, используем ли мы с бустингом фиксированно-контурные, нейросетевые или древесные единицы, мы, естественно, предпочтем единицы с *низкой емкостью*, чтобы зернистость нашего модельного поиска была как можно более мелкой. Когда мы начинаем добавлять единицы по одной за раз, то поворачиваем наш оптимизационный диск по часовой стрелке слева направо. Мы хотим, чтобы этот поворот диска выполнялся как можно более плавно, чтобы мы могли просканировать кривую валидационной ошибки мелкозернисто в поисках ее минимума (рис. 11.30, *слева*). Если на каждом раунде бустинга мы будем использовать *высо-*

коемкостные единицы, то результирующий модельный поиск будет намного грубее, т. к. добавление каждой дополнительной единицы приводит к агрессивному повороту диска слева направо, оставляя большие интервалы в нашем модельном поиске (рис. 11.30, *справа*). Этот вид крупнозернистого поиска может легко привести к тому, что мы проскочим через сложность оптимальной модели. То же самое можно сказать и по поводу того, почему с бустингом мы добавляем только одну единицу за раз, настраивая только ее параметры в каждом раунде. Если бы мы добавляли более одной единицы за раз или перенастраивали *каждый* параметр *каждой* единицы на каждом шаге этого процесса, то нам не только пришлось бы значительно больше вычислять на каждом шаге, но и разница в результативности между последующими моделями могла бы быть довольно большой, и мы могли бы легко упустить идеальную модель.

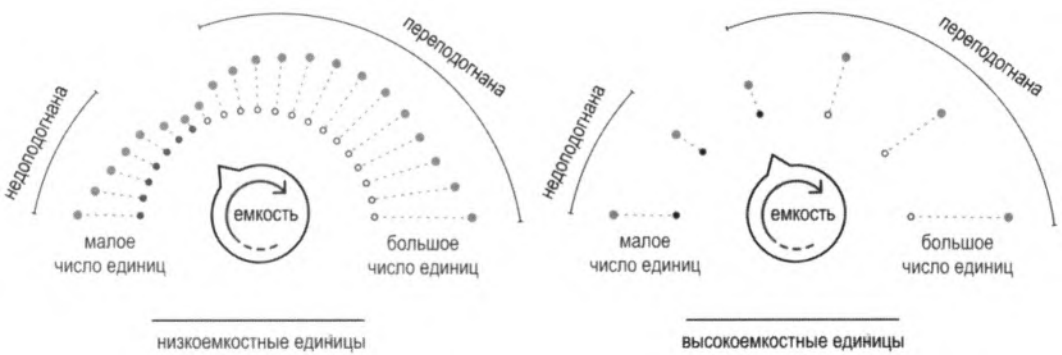


Рис. 11.30. Использование низкоемкостных структурных единиц превращает процедуру бустинга в мелкозернистый поиск оптимальной модельной сложности (*слева*). Использование высокоемкостных единиц превращает бустинг в крупнозернистый поиск оптимальной модельной сложности (*справа*)

11.5.2. Технические детали

Формализуя наше приведенное выше обсуждение темы бустинга, мы начнем с множества из M нелинейных признаков или структурных единиц из отдельного семейства универсальных аппроксиматоров

$$\mathcal{F} = \{f_1(\mathbf{x}),\ f_2(\mathbf{x}),\ ...,\ f_M(\mathbf{x})\}\ . \tag{11.20}$$

Мы добавляем эти единицы последовательно (или по одной за раз), выстраивая последовательность из M настроенных моделей $[\text{model}_m]_{m=1}^{M}$, которые увеличивают сложность по отношению к тренировочным данным, от $m = 1$ до $m = M$, заканчивая обобщенной нелинейной моделью, состоящей из M единиц. Выразим эту окончательную изготовленную посредством бустинга модель несколько иначе, чем в уравнении (11.19), в частности переиндексировав единицы, из которых она построена, как

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_{s_1}(\mathbf{x})w_1 + f_{s_2}(\mathbf{x})w_2 + ... + f_{s_M}(\mathbf{x})w_M\ . \tag{11.21}$$

Здесь мы переиндексировали индивидуальные единицы в f_{s_m}, чтобы обозначить единицу из всей коллекции в $\mathcal{F}$, добавляемому на m-м раунде процесса бустинга. Линейно-комбинационные веса с w_0 по w_M, наряду с любыми дополнительными весами, внутренними для f_{s_1}, f_{s_2}, ..., f_{s_M}, представлены коллективно в весовом множестве Θ.

Процесс бустинга выполняется в общей сложности в M раундах, по одному для каждой из единиц в уравнении (11.21). В каждом раунде мы определяем то, какая единица при ее добавлении в скользящую модель лучше всего снижает ее тренировочную ошибку. Затем мы измеряем соответствующую валидационную ошибку, предоставляемую этим обновлением, и, в конце концов, после завершения всех раундов бустинга, используем наименьшую найденную меру валидационной ошибки, чтобы принять решение о том, какой раунд обеспечил наилучшую совокупную модель.

Ради простоты в описании формальных деталей бустинга мы сосредоточим наше обсуждение на одной-единственной задаче: нелинейная регрессия на тренировочном наборе данных $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$, задействовав стоимость наименьших квадратов. Однако принципы бустинга, как мы увидим, остаются *точно* такими же для других задач автоматического усвоения (например, двухклассовой и многоклассовой классификации) и связанных с ними стоимостей.

Раунд 0 бустинга

Мы начинаем процедуру бустинга с настройки

$$\text{model}_0(\mathbf{x}, \Theta_0) = w_0, \tag{11.22}$$

чье весовое множество $\Theta_0 = \{w_0\}$ содержит один-единственный вес смещения, который мы можем легко настроить, минимизировав соответствующую стоимость только над этой переменной. Держа это в уме, для того чтобы отыскать оптимальное значение для w_0, мы минимизируем стоимость наименьших квадратов

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0(\mathbf{x}_p, \Theta_0) - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0 - y_p\right)^2. \tag{11.23}$$

Это дает оптимальное значение для w_0, которое обозначено через $w_0^\star$. Мы фиксируем вес смещения на этом значении навсегда на протяжении всего процесса.

Раунд 1 бустинга

Настроив один-единственный параметр модели model_0, мы теперь *бустируем* ее сложность путем добавления в нее взвешенной единицы $f_{s_1}(\mathbf{x})w_1$ и в результате получаем модифицированную скользящую модель, которую мы называем model_1:

$$\text{model}_1(\mathbf{x}, \Theta_1) = \text{model}_0(\mathbf{x}, \Theta_0) + f_{s_1}(\mathbf{x})w_1. \tag{11.24}$$

Обратите внимание, что здесь параметрическое множество Θ_1 содержит w_1 и любые параметры, внутренние для единицы f_{s_1}. Для того чтобы определить, какая структурная единица в нашем множестве $\mathcal{F}$ лучше всего снижает тренировочную ошибку, мы прикладываем модель model_1 к данным, минимизируя

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0\left(\mathbf{x}_p, \Theta_0\right) + f_{s_1}\left(\mathbf{x}_p\right)w_1 - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0^\star + f_{s_1}\left(\mathbf{x}_p\right)w_1 - y_p\right)^2 \quad (11.25)$$

для каждой единицы $f_{s_1} \in \mathcal{F}$.

Обратите внимание, что, поскольку вес смещения уже был задан оптимально в предыдущем раунде, нам нужно только настроить вес w_1, а также параметры, внутренние для нелинейной единицы f_{s_1}. Также обратите внимание, в частности, на то, что в нейронных сетях все нелинейные единицы принимают в точности одинаковую форму, и, следовательно, от нас не требуется решать M вариантов задачи оптимизации из уравнения (11.25), по одному для каждой структурной единицы в $\mathcal{F}$, как это было бы при использовании фиксированно-контурных или древесных единиц. Независимо от типа задействованного универсального аппроксиматора, раунд 1 бустинга заканчивается отысканием оптимальных f_{s_1} и w_1, которые мы обозначаем соответственно через $f_{s_1}^\star$ и $w_1^\star$ и оставляем фиксированными, двигаясь вперед.

Раунд $m > 1$ бустинга

В общем случае на m-м раунде бустинга мы начинаем с модели model_{m-1}, состоящей из члена смещения и $m-1$ структурных единиц формы:

$$\text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right) = w_0^\star + f_{s_1}^\star(\mathbf{x})w_1^\star + f_{s_2}^\star(\mathbf{x})w_2^\star + \ldots + f_{s_{m-1}}^\star(\mathbf{x})w_{m-1}^\star. \quad (11.26)$$

Обратите внимание, что параметры этой модели были настроены последовательно, начиная со смещения $w_0^\star$ в раунде 0, $w_1^\star$ и любых параметров, внутренних для $f_{s_1}^\star$, в раунде 1 и т. д., вплоть до $w_{m-1}^\star$ и любых параметров, внутренних для $f_{s_{m-1}}^\star$, в раунде $m-1$.

m-й раунд бустинга следует шаблону, описанному в раунде 1, где мы отыскиваем наилучшую взвешенную единицу $f_{s_m}(\mathbf{x})w_m$ и добавляем ее в нашу скользящую модель, чтобы наилучшим образом снизить ее тренировочную ошибку в наборе данных. В частности, наша m-я модель принимает форму

$$\text{model}_m\left(\mathbf{x}, \Theta_m\right) = \text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right) + f_{s_m}(\mathbf{x})w_m, \quad (11.27)$$

и мы определяем надлежащую единицу, пригодную для добавления в эту модель путем минимизирования

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + f_{s_m}\left(\mathbf{x}_p\right)w_m - y_p\right)^2 =$$

$$= \frac{1}{P}\sum_{p=1}^{P}\left(w_0^\star + w_1^\star f_{s_1}^\star + \ldots + f_{s_{m-1}}^\star\left(\mathbf{x}_p\right)w_{m-1}^\star + f_{s_m}\left(\mathbf{x}_p\right)w_m - y_p\right)^2 \quad (11.28)$$

над w_m и внутренними для f_{s_m} параметрами (если они существуют), которые содержатся в параметрическом множестве Θ_m.

Опять-таки в случае с древесными и фиксированно-контурными аппроксиматорами это влечет за собой решение M (или $M-m+1$, если мы решим проверить только те структурные единицы, которые не использовались в предыдущих раундах) таких задач оптимизации и выбор той, которая имеет наименьшую тренировочную ошибку. В случае нейронных сетей, поскольку каждая структурная единица принимает одинаковую форму, нам нужно решить только одну такую задачу оптимизации.

11.5.3. Ранняя остановка

После завершения всех раундов бустинга обратите внимание на то, как мы сгенерировали последовательность из M настроенных моделей[10] — обозначенных через $[\text{model}_m(\mathbf{x}, \Theta_m)]_{m=1}^{M}$, — которые постепенно увеличивают нелинейную сложность с $m=1$ до $m=M$ и, следовательно, постепенно уменьшают тренировочную ошибку. Это дает нам мелкозернистый контроль при отборе подходящей модели, т. к. скачок в результативности с точки зрения как тренировочной, так и валидационной ошибок между последующими моделями в этой последовательности может быть довольно гладким, если мы используем малоемкостные структурные единицы (как описано в *разд. 11.5.1*).

После завершения бустинга мы выбираем из нашего множества моделей ту, которая обеспечивает наименьшую валидационную ошибку. В качестве альтернативы, вместо выполнения всех раундов бустинга и принятия решения об оптимальной модели постфактум, мы можем попытаться *остановить* процедуру, когда валидационная ошибка впервые начинает увеличиваться. Эта концепция, именуемая *ранней остановкой*, приводит к более эффективной имплементации бустинга с вычислительной точки зрения, но нужно быть осторожным в принятии решения, когда валидационная ошибка действительно достигла своего минимума, поскольку она может осциллировать вверх и вниз несколько раз (как упомянуто в *разд. 11.4*) и не должна принимать простую обобщенную форму, показанную на рис. 11.29, *сверху*. Окончательного решения этой проблемы не существует. Поэтому, когда задействуется ранняя остановка, в типичной ситуации на практике задействуются нерегламентированные решения.

11.5.4. Недорогое, но эффективное улучшение

Легкая корректировка на каждом раунде бустинга в виде добавления индивидуального смещения может значительно улучшить алгоритм. Формально на m-м раунде

---

[10] Мы исключили модель model_0, т. к. она не использует никаких структурных единиц универсального аппроксиматора.

бустинга вместо формирования модели model_m, как показано в уравнении (11.27), мы вносим добавочный вес смещения $w_{0,m}$:

$$\text{model}_m(\mathbf{x}, \Theta_m) = \text{model}_{m-1}(\mathbf{x}, \Theta_{m-1}) + w_{0,m} + f_{s_m}(\mathbf{x})w_m . \quad (11.29)$$

Эта простая корректировка приводит к большей гибкости и в целом к более высокой совокупной результативности, позволяя корректировать структурные единицы "вертикально" в каждом раунде (в случае регрессии) с минимальной стоимостью добавления одной переменной в каждую оптимизационную подзадачу. Обратите внимание, что после завершения настройки оптимальный вес смещения $w^{\star}_{0,m}$ может быть поглощен весами смещения из предыдущих раундов, создавая единый вес смещения $w^{\star}_0 + w^{\star}_{0,1} + \ldots + w^{\star}_{0,m}$ для всей модели.

Это улучшение в особенности полезно при использовании фиксированно-контурных или нейросетевых единиц для бустинга, т. к. оно является избыточным при использовании древесных аппроксиматоров, поскольку в них уже вклеены индивидуальные члены смещения, которые всегда допускают такого рода вертикальную корректировку на каждом раунде бустинга[11].

Для того чтобы это увидеть, обратите внимание, что уравнение (11.14) показывает наиболее часто встречающийся способ выражения пня, принимающего одномерный вход

$$f(x) = \begin{cases} v_2, & \text{если } x \le s; \\ v_2, & \text{если } x > s. \end{cases} \quad (11.30)$$

Функцию $f(x)$ также можно выразить эквивалентным образом как

$$f(x) = w_0 + w_1 h(x), \quad (11.31)$$

где w_0 — индивидуальный параметр смещения для пня; w_1 — ассоциированный вес, который шкалирует функцию $h(x)$, представляющую собой простую шаговую функцию с фиксированными уровнями и ветвлением в $x = s$:

$$h(x) = \begin{cases} 0, & \text{если } x \le s; \\ 1, & \text{если } x > s. \end{cases} \quad (11.32)$$

Выражая пень таким эквивалентным образом, мы получаем возможность увидеть, что каждая единица пня действительно имеет собственный индивидуальный параметр смещения, а это делает излишним добавление индивидуального смещения в каждом раунде при бустинге с участием пней. Та же концепция соблюдается и для пней, принимающих в целом N-мерный вход.

---

[11] На жаргоне машинного обучения бустинг с древесными учениками часто именуется бустированием градиента, или градиентным бустингом. Дальнейшие подробности *см. в разд. 14.5.*

Пример 11.10. Бустинг регрессии с использованием древесных единиц

В этом примере мы используем синусоидальный регрессионный набор данных, впервые показанный в примере 11.6, состоящий из $P = 21$ точки данных, и строим множество из $B = 20$ древесных (пневых) единиц для этого набора данных (*см. разд. 11.2.3*). На рис. 11.31 проиллюстрирован результат $M = 50$ раундов бустинга (многие пни используются многократно). Мы разбили набор данных на тренировочный в размере 2/3 и валидационный в размере 1/3, которые имеют цветовую кодировку соответственно светло-синего и оранжевого цвета. На рисунке показаны результирующие регрессионные подгонки и связанные с ними тренировочная/валидационная ошибки для нескольких раундов бустинга. Этот пример рассматривается далее в *разд. 14.5*.

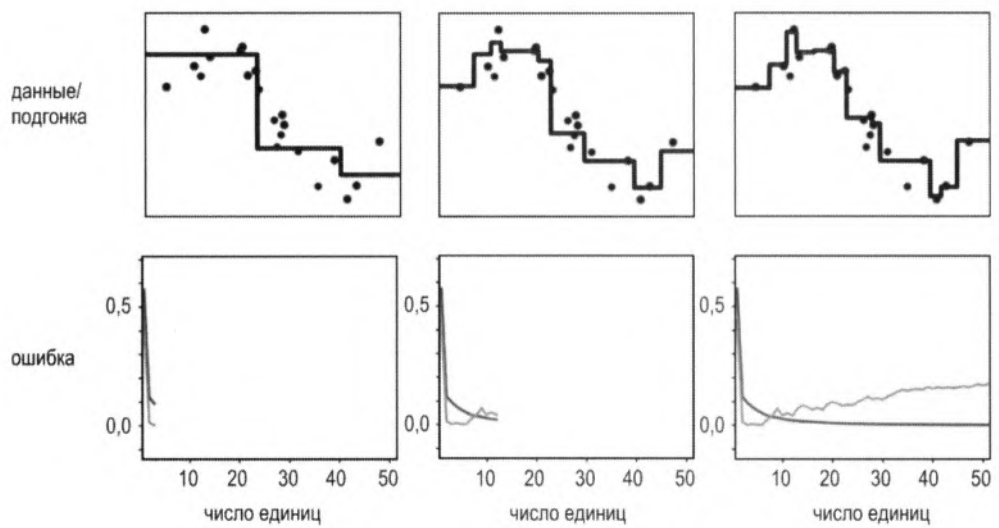


Рис. 11.31. Иллюстрация примера 11.10

Пример 11.11. Бустинг классификации с использованием нейросетевых единиц

В этом примере мы иллюстрируем тот же самый вид бустинга, что и ранее в примере 11.10, но теперь для двухклассовой классификации используется набор данных из $P = 99$ точек данных, который имеет (грубо говоря) округлую границу решения. Этот набор данных впервые был использован в примере 11.7. Мы произвольно разбили набор данных на тренировочный в размере 2/3 и валидационный в размере 1/3, а также задействуем нейросетевые единицы для бустинга. На рис. 11.32 представлены результат $M = 30$ раундов бустинга в терминах нелинейной границы решения и результирующая классификация, а также тренировочная/валидационная ошибки.

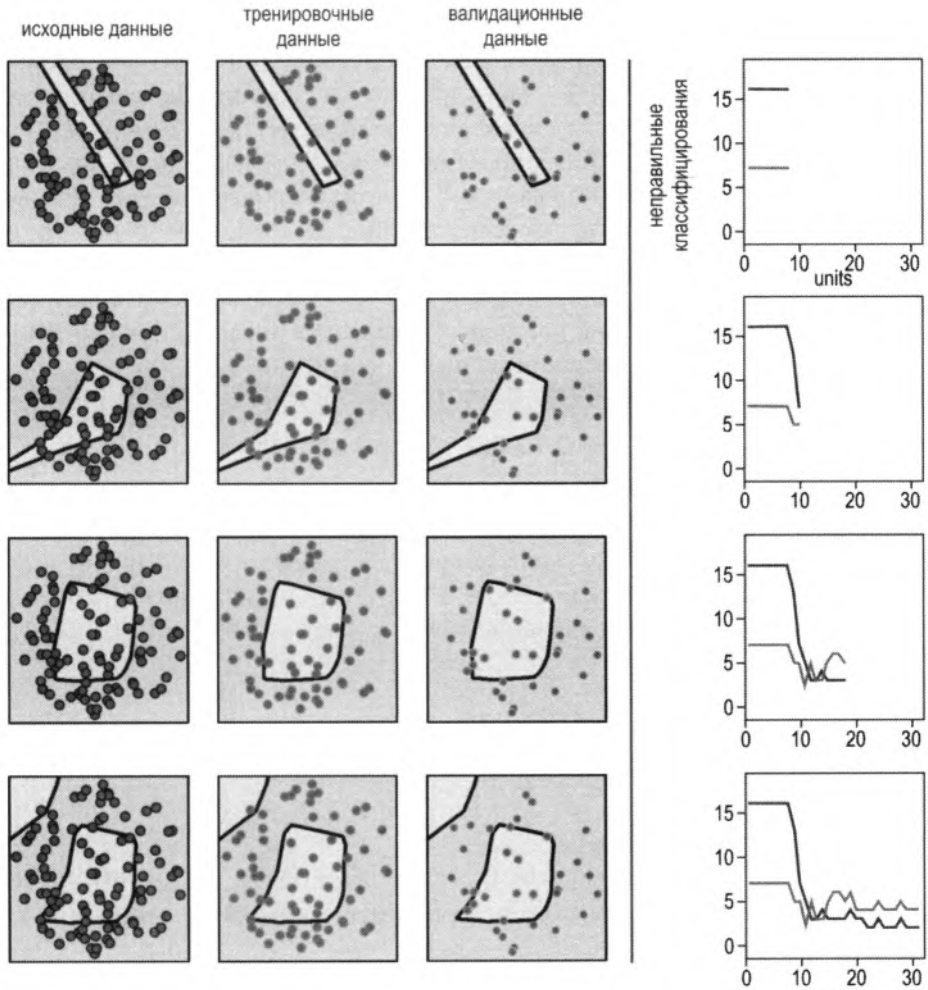


Рис. 11.32. Иллюстрация примера 11.11

11.5.5. Сходство с отбором признаков

Внимательный читатель заметит то, насколько процедура бустинга похожа на ту, что представлена в *разд. 9.6* в контексте отбора признаков. И действительно, в принципе эти два подхода совершенно похожи, за исключением того, что при бустинге мы не отбираем из множества заданных входных признаков, а создаем их сами на основе выбранного семейства универсальных аппроксиматоров. Кроме того, в отличие от отбора признаков, где наша главная забота — *интерпретируемость* результатов *человеком*, мы в первую очередь используем бустинг как инструмент для перекрестной валидации. Это означает, что если мы специально не запретим этому происходить в процессе бустинга, то на самом деле мы можем отбирать одинаковый признак многократно, в случае если он положительно влияет на отыскание модели с минимальной валидационной ошибкой.

Эти два варианта бустинга, т. е. отбор признаков и перекрестная валидация, могут происходить вместе, пусть и, как правило, в контексте линейного моделирования, как подробно описано в *разд. 9.6*. Часто в таких случаях перекрестная валидация используется с линейной моделью как способ автоматического отбора надлежащего числа признаков, при этом интерпретация результирующих отбранных признаков человеком по-прежнему учитывается. С другой стороны, отбор признаков редко осуществляется при задействовании нелинейной модели, основанной на признаках из универсального аппроксиматора; это обусловлено большими трудностями в интерпретации нелинейных признаков человеком. Редким исключением из этого правила является использование древесных единиц, которые в силу своей простой структуры могут быть легко интерпретированы человеком.

11.5.6. Взгляд на регрессию с точки зрения остатков

Здесь мы опишем общепринятую интерпретацию бустинга в контексте регрессии, т. е. последовательной подгонки к *остатку* регрессионного набора данных. Для того чтобы понять, что под этим подразумевается, рассмотрим следующую стоимостную функцию наименьших квадратов, в которую мы вставили бустированную модель на m-м раунде ее разворачивания:

$$g(\Theta_m) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_m\left(\mathbf{x}_p, \Theta_m\right) - y_p\right)^2 . \tag{11.33}$$

Мы можем записать нашу бустированную модель рекурсивно как

$$\text{model}_m\left(\mathbf{x}_p, \Theta_m\right) = \text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + f_m\left(\mathbf{x}_p\right) w_m , \tag{11.34}$$

где все параметры $(m-1)$-й модели (т. е. model_{m-1}) уже настроены. Совместив уравнения (11.33) и (11.34), мы можем переписать стоимость наименьших квадратов как

$$g(\Theta_m) = \frac{1}{P}\sum_{p=1}^{P}\left(f_m\left(\mathbf{x}_p\right) w_m - \left(y_p - \text{model}_{m-1}\left(\mathbf{x}_p\right)\right)\right)^2 . \tag{11.35}$$

Минимизируя эту стоимость, мы стремимся настроить параметры одной добавочной единицы так, чтобы

$$f_m\left(\mathbf{x}_p\right) w_m \approx y_p - \text{model}_{m-1}\left(\mathbf{x}_p\right) \tag{11.36}$$

для всех p или, другими словами, так, чтобы эта полностью настроенная структурная единица аппроксимировала наш первоначальный выход y_p минус вклад предыдущей модели. Эта величина, разница между нашим первоначальным выходом и вкладом $(m-1)$-й модели, часто именуется *остатком*: это то, что остается представить после вычитания того, что было усвоено $(m-1)$-й моделью.

Пример 11.12. Бустинг с точки зрения подгонки к остатку

На рис. 11.33 проиллюстрирован процесс бустинга $M = 20$ нейросетевых единиц в игрушечном регрессионном наборе данных. В верхнем ряду показан набор дан-

ных, а также подгонка, предоставляемая моделью model_m на m-м шаге бустинга для отборных значений m. В нижнем ряду построены графики *остатков* на тех же шагах, а также отражена подгонка, предоставляемая соответствующей m-й единицей f_m. По мере прогрессии бустинга подгонка исходных данных улучшается, в то время как (одновременно) остаток сжимается.

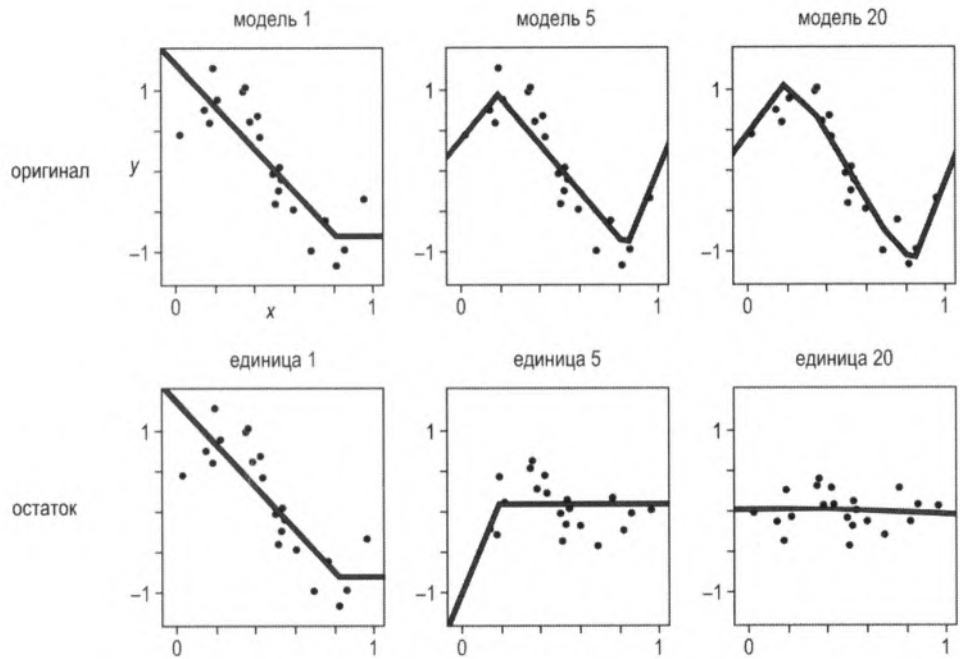


Рис. 11.33. Иллюстрация примера 11.12

11.6. Эффективная перекрестная валидация посредством регуляризации

В предыдущем разделе мы увидели, как с помощью перекрестной валидации на основе бустинга мы автоматически усваиваем надлежащую степень модельной сложности для конкретного набора данных, оптимизируя общую высокоемкостную модель по одной единице за раз. В этом разделе мы введем методику *регуляризации* для эффективной перекрестной валидации. С помощью этого набора подходов мы снова начнем с одной-единственной высокоемкостной модели и скорректируем ее сложность относительно тренировочного набора данных путем тщательной оптимизации. Однако в случае регуляризации мы будем настраивать все структурные единицы *одновременно*, контролируя качество *оптимизации* связанной стоимости для достижения минимального валидационного экземпляра модели.

11.6.1. Картина в целом

На мгновение представьте себе, что у нас есть простой нелинейный регрессионный набор данных, подобный тому, который показан на рис. 11.34, *слева*, и мы используем высокоемкостную модель (относительно природы данных), составленную из суммы *универсальных аппроксиматоров* одного рода, чтобы вписать ее как

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1(\mathbf{x})w_1 + f_2(\mathbf{x})w_2 + \ldots + f_M(\mathbf{x})w_M \,. \qquad (11.37)$$

Предположим, что мы подразделяем эти данные на тренировочную и валидационную части, а затем тренируем нашу высокоемкостную модель, *полностью* оптимизируя стоимость наименьших квадратов над тренировочной частью данных. Другими словами, мы определяем множество параметров для нашей высокоемкостной модели, которые лежат очень близко к глобальному минимуму ассоциированной стоимостной функции. На рис. 11.34, *справа сверху* приведена гипотетическая двумерная иллюстрация стоимостной функции, ассоциированной с нашей высокоемкостной моделью, над тренировочными данными; глобальный минимум обозначен синей графической точкой, а его значение на кривой функции — синим крестиком.

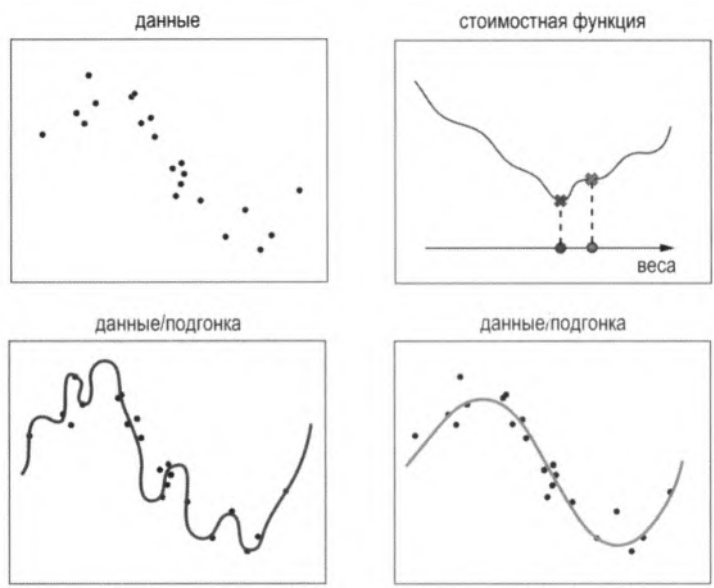


Рис. 11.34. Обобщенный нелинейно-регрессионный набор данных (*слева сверху*). Фигуральная иллюстрация стоимостной функции, связанной с высокоемкостной моделью, над тренировочной частью этих данных (*справа сверху*). Глобальный минимум обозначен здесь синей графической точкой (вместе с его значением, обозначенным синим крестиком), а точка рядом отмечена оранжевым цветом (значение которой показано как оранжевый крестик). Исходные данные и подгонка (синим цветом), предоставляемая моделью с использованием параметров из глобального минимума стоимостной функции и экстремально переобученная к тренировочной части данных (*слева снизу*). Параметры, соответствующие оранжевой точке, показанной на правом верхнем рисунке, минимизируют стоимостную функцию над валидационной частью данных и, таким образом, обеспечивают гораздо более качественную подгонку (обозначенную оранжевым цветом) к данным (*справа снизу*)

Поскольку наша модель обладает высокой емкостью, результирующая подгонка, обеспечиваемая параметрами, лежащими на глобальном минимуме нашей стоимости, произведет настроенную модель, являющуюся чрезмерно сложной и *чересчур* обученной к тренировочной части нашего набора данных. На рис. 11.34, *слева снизу* показан результат подгонки (синим цветом) настроенной модели, обеспечиваемый таким множеством параметров, которая экстремально переобучена к тренировочным данным. *Справа сверху* также представлено множество параметров, лежащих относительно близко к глобальному минимуму, в виде оранжевой графической точки, вычисленное значение функции которой отмечено оранжевым крестиком. Это множество параметров, лежащих в общей окрестности глобального минимума, является тем самым местом, где стоимостная функция минимизируется над валидационной частью наших данных. Вследствие этого соответствующая подгонка (показанная *справа снизу* оранжевым цветом) обеспечивает гораздо более качественное представление данных.

Этот пример иллюстрирует более общий принцип, который мы уже видели в *разд. 11.3.2*: переподгонка происходит как из-за того, что *емкость* ненастроенной модели является лишком высокой, так и из-за того, что соответствущая ей стоимостная функция (над тренировочными данными) *оптимизирована* слишком хорошо, что приводит к чрезмерно сложной настроенной модели. Это явление соблюдается для всех задач машинного обучения (включая регрессию, классификацию и приемы обучения без учителя, такие как автокодировщик) и мотивирует применение общих стратегий перекрестной валидации на основе регуляризации: если надлежащая оптимизация *всех параметров* высокоемкостной модели приводит к переподгонке, то этого можно избежать, оптимизировав означенную модель *несовершенно*, когда валидационная ошибка (не тренировочная ошибка) находится на самом низком уровне. Другими словами, *регуляризация* в контексте перекрестной валидации представляет собой совокупность подходов к перекрестной валидации, при которых мы тщательно настраиваем все параметры высокоемкостной модели, целенаправленно устанавливая их вдали от глобального минимума ассоциированной стоимостной функции. Это делается различными способами, и далее мы подробно остановимся на двух наиболее популярных подходах.

11.6.2. Регуляризация на основе ранней остановки

При регуляризации на основе ранней остановки[12] мы надлежаще настраиваем высокоемкостную модель, выполняя прогон локальной оптимизации (настраивая все параметры модели) и используя множество весов из этого прогона, где модель достигает минимальной валидационной ошибки. Эта идея проиллюстрирована на рис. 11.35, *слева*, где мы задействуем ту же самую прототипическую стоимостную функцию, впервые показанную на рис. 11.34, *справа сверху*. Во время выполнения

[12] Этот регуляризационный подход особенно популярен при задействовании моделей на основе глубоких нейронный сетей, подробно описанных в *разд. 13.7*.

локальной оптимизации мы часто вычисляем тренировочную и валидационную ошибки (например, на каждом шаге оптимизационной процедуры) так, чтобы множество весов, обеспечивающих минимальную валидационную ошибку, могло быть определено с мелкой зернистостью.

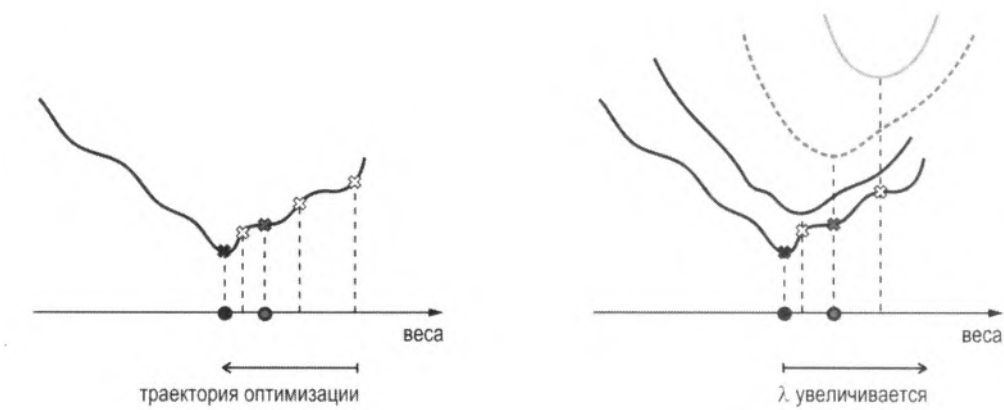


Рис. 11.35. Фигуральная иллюстрация регуляризации на основе ранней остановки применительно к прототипической стоимостной функции высокоемкостной модели. Мы делаем оптимизационный прогон — здесь показано выполнение до полного завершения в глобальном минимуме синим цветом — и выбираем набор весов, которые обеспечивают минимальную валидационную ошибку (показано оранжевым цветом) (*слева*). Фигуральная иллюстрация регуляризации на основе регуляризатора. Добавляя функцию-регуляризатор в стоимость, связанную с высокоемкостной моделью, мы изменяем ее контур, в частности перетаскивая ее глобальный минимум (где происходит переобученное поведение) в сторону, противоположную от ее первоначального местоположения (*справа*). Затем регуляризованная стоимостная функция может быть полностью минимизирована для восстановления весов как можно ближе/дальше от истинного глобального минимума исходной стоимостной функции, в зависимости от выбора регуляризационного параметра λ. Надлежащая настройка этого параметра позволяет восстановить веса, минимизирующие валидационную ошибку

Неважно, останавливается ли процесс оптимизации буквально при достижении минимальной валидационной ошибки (что может быть непросто на практике, учитывая несколько непредсказуемое поведение валидационной ошибки, как впервые отмечалось в *разд. 11.4.2*), или выполняется оптимизация до полного завершения (с подбором наилучшего множества весов впоследствии), в любом случае мы называем этот метод *регуляризацией на основе ранней остановки*. Обратите внимание, что сам метод аналогичен процедуре ранней остановки, описанной для перекрестной валидации на основе бустинга в *разд. 11.5.3*, в том, что мы последовательно увеличиваем сложность модели, пока не будет достигнута минимальная валидация. Однако здесь (в отличие от бустинга) мы контролируем то, насколько хорошо мы оптимизируем модельные параметры *одновременно*, а не по одной единице за раз.

Предположив, что мы запускаем нашу оптимизацию с малого начального значения (что обычно и делаем), мы получим, что соответствующие кривые тренировочной

и валидационной ошибок будут в общем случае[13] выглядеть так, как показано на рис. 11.36, *сверху*. В начале прогона сложность нашей модели (вычисляемой над начальными весами) довольно мала, что обеспечивает большую тренировочную и валидационную ошибку. По мере того как минимизация продвигается вперед, и мы шаг за шагом продолжаем оптимизировать, ошибки как в тренировочной, так и в валидационной части данных уменьшаются, в то время как сложность настроенной модели увеличивается. Этот тренд продолжается до тех пор, пока сложность модели не станет слишком большой и не начнется переподгонка, а валидационная ошибка не увеличится.

В терминах концептуализации с емкостным/оптимизационным наборными дисками, подробно описанной в *разд. 11.3.2*, мы можем рассуждать о регуляризации

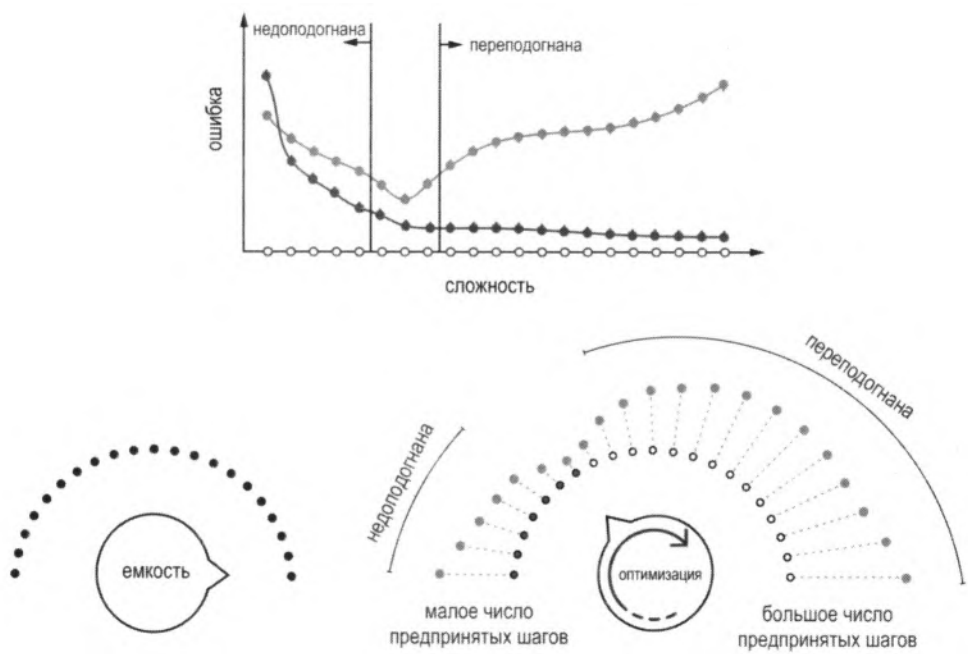


Рис. 11.36. Прототипическая пара кривых тренировочной/валидационной ошибок, связанных с обобщенным прогоном регуляризации на основе ранней остановки (*сверху*). В случае с ранней остановкой мы поворачиваем наш емкостный диск до конца вправо, а оптимизационный диск — до конца влево (*снизу*). Затем мы медленно перемещаем наш оптимизационный диск слева направо, итеративно улучшая подгонку нашей модели к тренировочным данным, настраивая все ее параметры одновременно по одному шагу за раз. С каждым шагом оптимизации мы медленно поворачиваем оптимизационный диск по часовой стрелке слева направо, постепенно увеличивая сложность нашей настроенной модели в поисках настроенной модели с минимальной валидационной ошибкой. Здесь каждая отметка на оптимизационном диске абстрактно обозначает шаг локальной оптимизации

[13] Обратите внимание, что и то и другое может осциллировать на практике в зависимости от используемого метода оптимизации.

(основанной на ранней остановке) как начинающейся с момента установки нашего емкостного диска в крайнее *правое* положение (поскольку мы задействуем высокоемкостную модель) и нашего оптимизационного диска в крайнее *левое* положение (при инициализации нашей оптимизации). С этой конфигурацией — кратко описанной визуально на рис. 11.36, *внизу* — мы позволяем нашему оптимизационному диску напрямую управлять степенью сложности, которую могут принимать наши настроенные модели. Другими словами, при такой конфигурации наш оптимизационный диск становится идеальным сложностным диском, описанным в начале главы в *разд. 11.1.3*. С ранней остановкой мы поворачиваем наш оптимизационный диск слева направо, начиная с нашей инициализации, выполняем прогон локальной оптимизации по одному шагу за раз, выискивая множество параметров, которые обеспечивают минимальную валидационную ошибку для нашей (высокоемкостной) модели.

С имплементированием эффективной процедуры регуляризации на основе ранней остановки связан ряд важных инженерных деталей, которые мы обсудим далее.

- **Разные оптимизационные прогоны могут привести к разным настроенным моделям**. Топология стоимостной функции, связанная с высокоемкостными моделями, может быть довольно сложной. Отсюда, различные инициализации могут создавать разные траектории к потенциально разным минимумам стоимостной функции и создавать соответствующие модели, минимизиующие валидационную ошибку, которые различаются контуром, как показано на рис. 11.37, *сверху*. Однако на практике эти различия, как правило, не влияют на результативность, и результирующие модели можно легко совместить, или упакованы вместе на основе бутстрап-агрегирования (*см. разд. 11.9*), чтобы усреднить любые существенные различия в их индивидуальной результативности.
- **Насколько высокой должна быть задана емкость?** Как узнать, насколько высокой следует устанавливать емкость нашей модели при использовании ранней остановки (или любой другой формы) регуляризации, основанной на перекрестной валидации? В общем случае однозначного ответа нет. Емкость просто должна быть установлена "высоко", по крайней мере настолько, чтобы модель была полностью оптимизирована. Это достигается путем корректировки M (числа структурных единиц в модели) и/или емкости отдельных единиц (например, использованием единиц мелкой или глубокой нейронной сети или древесных единиц, как мы подробно рассмотрим соответственно в *главах 13* и *14*).
- **Локальная оптимизация должна выполняться тщательно**. Следует быть осторожным с типом схемы локальной оптимизации, используемой с перекрестной валидацией на основе ранней остановки.

  Как показано на рис. 11.37, *слева снизу*, в идеале мы хотим плавно поворачивать наш оптимизационный диск слева направо, отыскивая множество модельных сложностей с мелкой зернистостью. Из этого следует, например, что при ранней остановке мы часто избегаем схем локальной оптимизации, которые делают очень большие шаги (например, метод Ньютона, как подробно описано в *главе 4*), поскольку это может привести к грубому и крупнозернистому поиску модель-

ной сложности, который может легко проскочить через модели с минимальной валидацией, как показано на рис. 11.37, *справа снизу*. Локальные оптимизаторы, использующие более мелкие и высококачественные шаги, такие как продвинутые методы первого порядка, подробно описанные в *приложении 1*, часто предпочтительны при задействовании ранней остановки. Более того, при задействовании мини-пакетных/стохастических методов первого порядка (*см. разд. П1.5*) валидационная ошибка должна измеряться *несколько раз за эпоху* во избежание слишком большого числа шагов, предпринимаемых без измерения валидационной ошибки.

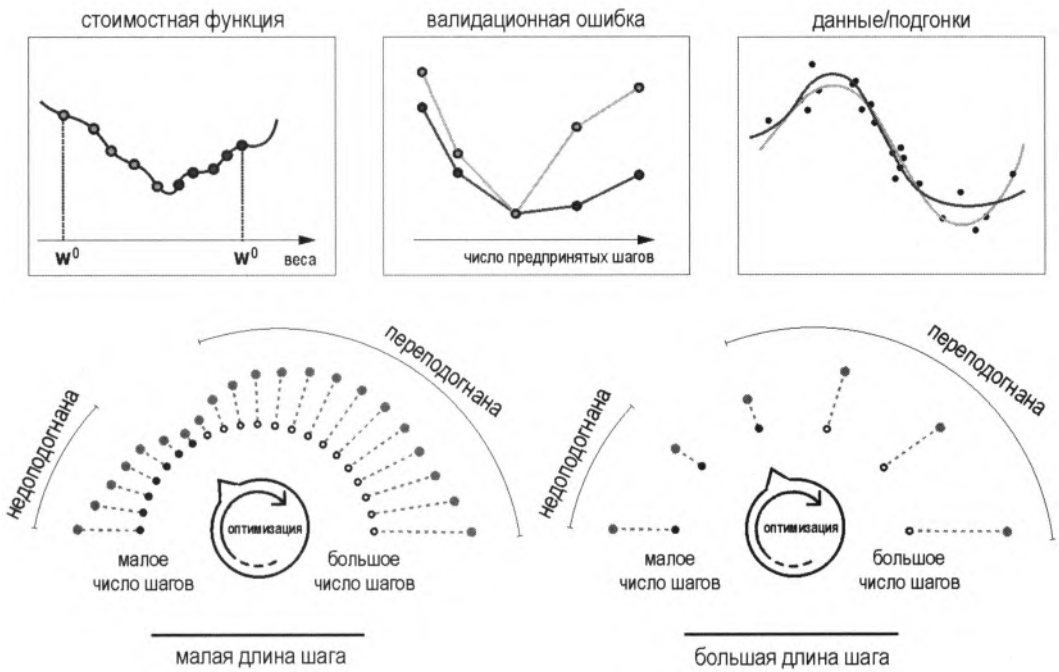


Рис. 11.37. Две тонкости, связанные с регуляризацией на основе ранней остановки. Прототипическая стоимостная функция, ассоциированная с высокоемкостной моделью, с двумя траекториями оптимизации (показанными соответственно красным и зеленым цветами), полученными в результате двух прогонов локальной оптимизации, начинающихся в разных начальных точках (*слева сверху*). Истории валидационной ошибки, соответствующие каждому оптимизационному прогону (*в центре сверху*). В то время как каждый прогон производит разный набор оптимальных весов и разную подгонку к данным (здесь показаны соответственно зеленым и красным цветами, соответствующими каждому прогону), эти подгонки, как правило, одинаково репрезентативны (*справа сверху*). Выполнение оптимизационных шагов с малой длиной превращает процедуру ранней остановки в мелкозернистый поиск оптимальной модельной сложности. С такими маленькими шагами мы плавно поворачиваем оптимизационный диск слева направо в поисках модели, минимизирующей валидационную ошибку (*слева снизу*). Использование шагов с большой длиной позволяет остановить крупнозернистый поиск оптимальной модельной сложности рано (*справа снизу*). С каждым шагом мы агрессивно поворачиваем диск слева направо, выполняя более грубый модельный поиск, который потенциально проскакивает через оптимальную модель

- **Когда валидационная ошибка действительно находится в самой низкой точке?** Хотя, вообще говоря, валидационная ошибка, уменьшаясь в начале процесса оптимизации, в конечном итоге увеличивается (образуя что-то вроде U-образного контура), она, безусловно, может колебаться вверх и вниз во время оптимизации. Поэтому не вполне очевидно, когда валидационная ошибка действительно достигла своей самой низкой точки, если только процесс оптимизации не выполняется до полного завершения. Для того чтобы уладить эту особенность, часто на практике принимается разумное инженерное решение относительно того, когда следует остановиться, основываясь на продолжительности времени, прошедшего с тех пор, как валидационная ошибка перестала уменьшаться. Более того, как упоминалось ранее, на самом деле нет необходимости останавливать локально-оптимизационную процедуру, чтобы задействовать рывок ранней остановки, и можно просто выполнить оптимизатор до полного завершения и выбрать наилучшее множество весов из указанного прогона после завершения.

Заинтересованный читатель может взглянуть на пример 13.14 в качестве простой иллюстрации регуляризации на основе ранней остановки.

11.6.3. Методы на основе регуляризаторов

Регуляризатор — это простая функция, которая может быть добавлена в стоимость машинного обучения для различных целей, например для предотвращения нестабильного усвоения (как мы видели в *разд. 6.4.6*), в качестве естественной части смягчения сценариев обучения на основе машин опорных векторов (*см. разд. 6.5.4*) и на основе многочисленных классов (*см. разд. 7.3.4*), а также для отбора признаков (*см. разд. 9.7*). Как мы увидим, последнее из этих применений (отбор признаков) очень похоже на наше использование регуляризатора здесь.

Добавив простую функцию-регуляризатор, подобную той, что мы видели в предыдущих применениях (например, ℓ_2-норму), в стоимость высокоемкостной модели, мы можем изменить ее контур и, в частности, переместить местоположение ее глобального минимума из ее первоначального местоположения. В общем случае если наша высокоемкостная модель задана как $\text{model}(\mathbf{x}, \Theta)$, связанная с ней стоимостная функция задана как g и регуляризатор как h, то регуляризованная стоимость задается как линейная комбинация g и h:

$$g(\Theta) + \lambda h(\Theta), \tag{11.38}$$

где λ называется *регуляризационным параметром*. Регуляризационный параметр всегда неотрицателен $\lambda \geq 0$ и управляет смесью стоимости и регуляризатора. Когда он задан малым и близким к нулю, регуляризованная стоимость по существу равна просто g, и наоборот, когда он задан очень большим, регуляризатор h доминирует в линейной комбинации (и поэтому при минимизации мы действительно просто минимизируем ее в одиночку). На рис. 11.35, *справа* показано, как изменяется контур фигуральной регуляризованной стоимости (а следовательно, и расположение ее глобального минимума) со значением λ.

Предполагая, что мы начинаем с большого значения λ и апробируем поступательно все меньшие значения (оптимизируя каждую регуляризованную стоимость полностью), соответствующие кривые тренировочной и валидационной ошибок в целом будут выглядеть примерно так, как показано на рис. 11.38, *сверху* (на практике помните, что *валидационная ошибка* может осциллировать и не обязательно должна иметь только одно падение вниз).

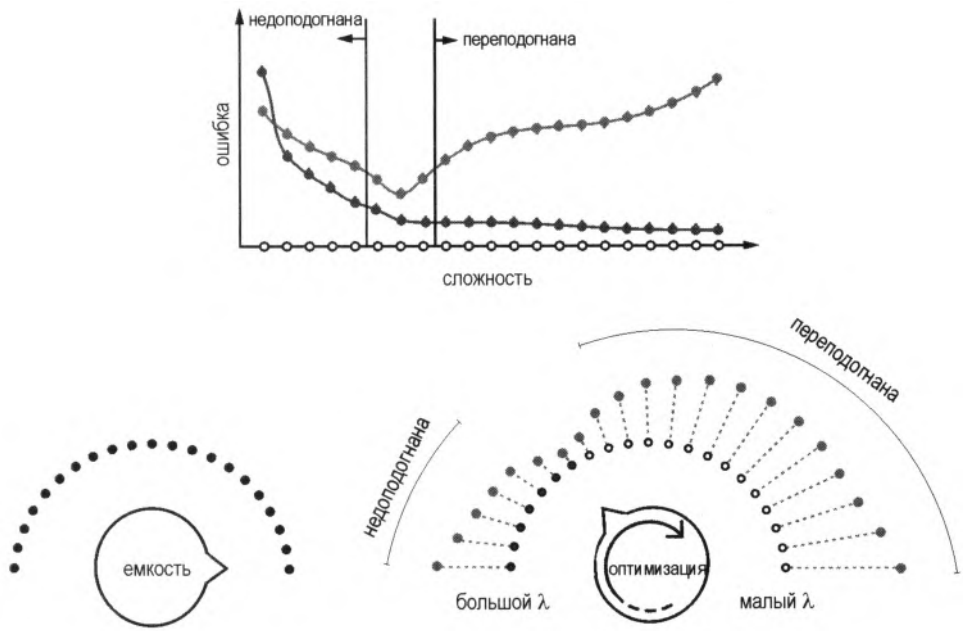


Рис. 11.38. Прототипическая пара кривых тренировочной/валидационной ошибок, связанных с обобщенным прогоном перекрестной валидации на основе регуляризатора (*сверху*). С помощью перекрестной валидации на основе регуляризатора мы устанавливаем наш емкостный диск в крайнее правое положение, а оптимизационный диск — в крайнее левое (начиная с большого значения для нашего регуляризационного параметра λ; *снизу*). Затем мы медленно перемещаем наш оптимизационный диск слева направо, уменьшая значение λ, здесь каждая отметка на оптимизационном диске представляет собой полную минимизацию соответствующей регуляризованной стоимостной функции в уравнении (11.38), улучшая подгонку нашей модели к тренировочным данным. Корректируя значение λ (и полностью минимизируя каждую соответствующую регуляризованную стоимость), мы медленно поворачиваем оптимизационный диск по часовой стрелке слева направо, постепенно увеличивая сложность нашей настроенной модели, в поисках модели с минимальной валидационной ошибкой

В начале этой процедуры при большом значении λ сложность нашей модели довольно мала, т. к. регуляризатор полностью доминирует в регуляризованной стоимости и, таким образом, связанный с ним восстановленный минимум принадлежит регуляризатору, а не самой стоимостной функции. Поскольку набор весов практически не связан с данными, над которыми мы тренируемся, соответствующая модель будет тяготеть к тому, чтобы иметь большие тренировочную и валидационную

ошибки. По мере уменьшения λ параметры, обеспечиваемые полной минимизацией регуляризованной стоимости, будут приближаться к глобальному минимуму самой первоначальной стоимости, и поэтому ошибка как на тренировочной, так и на валидационной части данных уменьшается, в то время как (вообще говоря) сложность настроенной модели возрастает. Этот тренд продолжается вплоть до момента, когда регуляризационный параметр не станет мал настолько, что восстановленные параметры будут лежать слишком близко к первоначальной стоимости, в результате чего соответствующая модельная сложность станет слишком большой. Здесь начинается переподгонка и увеличивается валидационная ошибка.

В терминах схемы емкостного/оптимизационного поворотных дисков, подробно описанной в *разд. 11.3.2*, мы можем рассуждать о перекрестной валидации на основе регуляризатора как о нашем емкостном диске, который вначале установлен в крайнее *правое* положение (поскольку мы задействуем высокоемкостную модель), и нашем оптимизационном диске, который установлен в крайнее *левое* положение (задействуя большое значение λ в нашей регуляризованной стоимости). С помощью этой конфигурации (обобщенной визуально на рис. 11.38, *снизу*) мы позволяем нашему оптимизационному диску непосредственно управлять степенью сложности, которую могут принимать наши настроенные модели. Поворачивая наш оптимизационный диск слева направо, мы *уменьшаем* значение λ и *полностью* минимизируем соответствующие регуляризованные стоимости, отыскивая множество параметров, которые обеспечивают минимальную валидационную ошибку для нашей (высокоемкостной) модели.

С имплементированием эффективной процедуры перекрестной валидации на основе регуляризатора связан ряд важных инженерных деталей, которые мы обсудим далее.

- **Веса смещения часто не включаются в регуляризатор**. Как и в случае линейных моделей, рассмотренных в *разд. 9.7*, часто в регуляризатор включаются только несмещенные веса общей модели. Например, предположим, что мы задействуем фиксированно-контурные универсальные аппроксиматоры, и, следовательно, наше множество параметров Θ содержит одно-единственное смещение w_0 и признаковые веса w_1, w_2, ..., w_B. Если затем мы выполним регуляризацию нашей стоимостной функции $g(\Theta)$, используя квадратичную ℓ_2-норму, то наша регуляризованная стоимость примет форму $g(\Theta)+\lambda\sum_{b=1}^{B} w_b^2$. При задействовании нейросетевых единиц мы следуем тому же самому шаблону, но здесь у нас гораздо больше членов смещения, включения которых в регуляризатор следует избегать. Например, если мы используем единицы формы $f_b(\mathbf{x})=\tanh\left(w_{b,0}+x_1w_{b,1}+\ldots+x_Nw_{b,N}\right)$, то член $w_{b,0}$ — внутренний для единицы — является членом смещения, который мы также не хотим включать в наш регуляризатор. Таким образом, для того чтобы регуляризировать стоимостную функцию, включающую эти единицы, используя квадратичную ℓ_2-норму, мы имеем $g(\Theta)+\lambda\left(\sum_{b=1}^{B} w_b^2+\sum_{b=1}^{B}\sum_{n=1}^{N} w_{b,n}^2\right)$.

♦ **Выбор функции-регуляризатора**. Обратите внимание, что в то время как ℓ_2-норма является очень популярным регуляризатором, в принципе в качестве регуляризатора можно использовать любую простую функцию. Другие популярные варианты функций-регуляризаторов включают регуляризатор на основе ℓ_1-нормы $h(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_{n=1}^{N} |w_n|$, который тяготеет к тому, чтобы производить разреженные веса, и регуляризатор на основе суммарной вариации $h(\mathbf{w}) = \sum_{n=1}^{N-1} |w_{n+1} - w_n|$, который тяготеет к тому, чтобы производить плавно варьирующиеся веса. Мы часто используем простой квадратичный регуляризатор (квадратичную ℓ_2-норму) с целью стимуляции весов к тому, чтобы оставаться *малыми*, как мы естественным образом делаем с двухклассовой и многоклассовой логистической регрессией. Каждый из этих разных типов регуляризаторов имеет тенденцию к тому, чтобы притягивать глобальный минимум суммы к разным частям входного пространства — как показано на рис. 11.39 для квадратичной функции (*слева сверху*), ℓ_1-нормы (*в центре сверху*) и суммарной вариационной нормы (*справа сверху*).

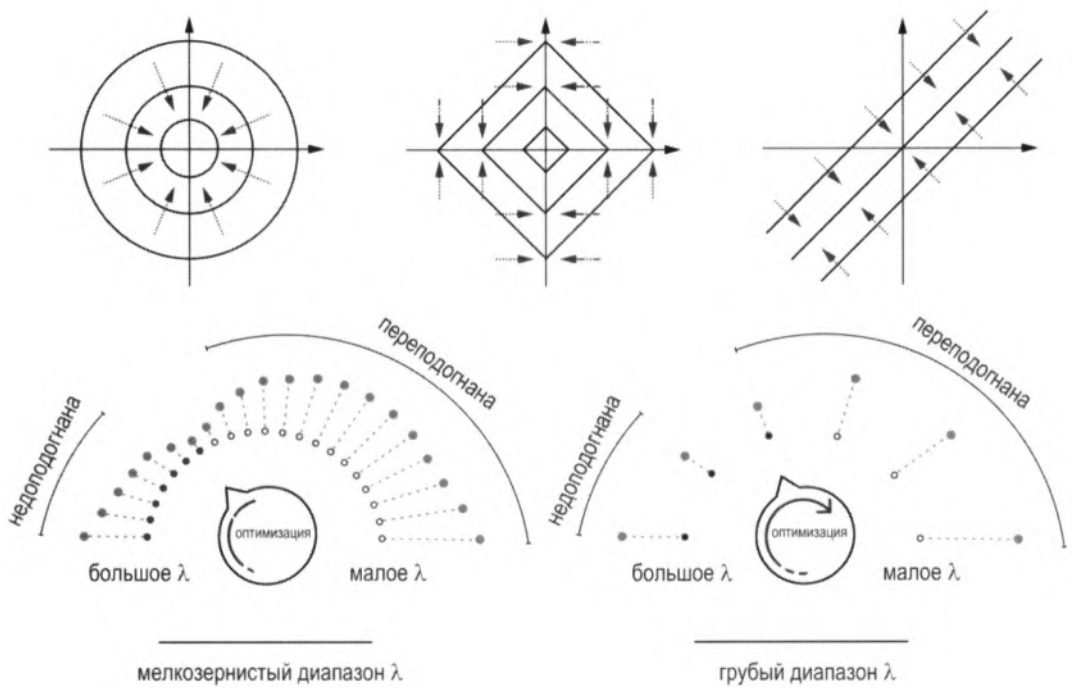


Рис. 11.39. Визуальное изображение того, где функции ℓ_2 (*слева сверху*), ℓ_1 (*в центре сверху*) и суммарной вариации (*справа сверху*) переносят глобальный минимум стоимостной функции — при использовании в качестве регуляризатора. Эти функции переносят глобальный минимум соответственно к началу координат, координатным осям и диагональным линиям, где последующие записи равны. Тестирование большого диапазона и числа значений регуляризационного параметра λ приводит к мелкозернистому поиску весов, минимизирующих валидационную ошибку (*слева снизу*). Меньшее число (или плохо выбранный диапазон) значений может привести к грубому поиску, который может проскочить через идеальные веса (*справа снизу*)

- **Выбор диапазона значений λ.** Аналогично тому, что мы уже видели в процедурах ранней остановки и бустинга, в случае с регуляризацией мы хотим выполнять наш поиск как можно тщательнее, поворачивая наш оптимизационный диск как можно плавнее слева направо в поисках нашей совершенной модели. Это желание непосредственно относится как к диапазону, так и к числу значений λ, которые мы тестируем. Например, чем больше значений мы апробируем в заданном диапазоне, тем плавнее мы поворачиваем наш оптимизационный диск (как показано визуально на рис. 11.39, *слева снизу*). Ограничение на количество опробируемых значений часто диктуется вычислительными и временными лимитами, т. к. для каждого апробированного значения λ должна быть выполнена полная минимизация соответствующей регуляризованной стоимостной функции. Это может сделать перекрестную валидацию на основе регуляризатора очень дорогостоящей с вычислительной точки зрения. С другой стороны, апробирование слишком малого числа значений может привести к грубому поиску весов, обеспечивающему минимальную валидационную ошибку, при этом увеличивая вероятность того, что такие веса будут целиком пропущены (как показано на рис. 11.39, *справа снизу*).

Пример 11.13. Настройка λ для задачи двухклассовой классификации

В этом примере мы используем квадратичный регуляризатор, чтобы найти надлежащий нелинейный классификатор для двухклассового классификационного набора данных, показанного рис. 11.40, *слева*, где периметр тренировочного набора окрашен в светло-синий цвет, а периметр валидационных точек — в оранжевый цвет. Здесь мы используем $B = 20$ нейросетевых единиц — высокоемкостную модель по отношению к этим данным — и пробуем 6 значений λ, равномерно распределенных между 0 и 0,5 (полностью минимизируя соответствующую регуляризованную стоимость в каждом случае). По мере изменения значения λ подгонки, обеспечиваемые весами, восстановленными из глобального минимума каждой регуляризованной стоимостной функции, показаны на рисунке слева, тогда как соответствующие тренировочная и валидационная ошибки продемонстрированы на рисунке справа и помечены соответственно синим и оранжевым цветами. В этом простом эксперименте значение где-то около $\lambda \approx 0{,}25$, судя по всему, обеспечивает наименьшую валидационную ошибку и соответствует наилучшей подгонке к набору данных в целом.

11.6.4. Сходство с регуляризацией для отбора признаков

Подобно процедуре бустинга, подробно рассмотренной в предыдущем разделе, здесь внимательный читатель заметит то, насколько описанный здесь каркас на основе регуляризатора похож на концепцию регуляризации, подробно описанную для отбора признаков в *разд. 9.7*. Эти два подхода очень близки по тематике, за исключением того, что здесь мы не выбираем из набора заданных входных признаков, а *создаем* их сами на основе универсального аппроксиматора. Вдобавок, вместо того, чтобы стремиться к *интерпретируемости* модели машинного обучения *чело-*

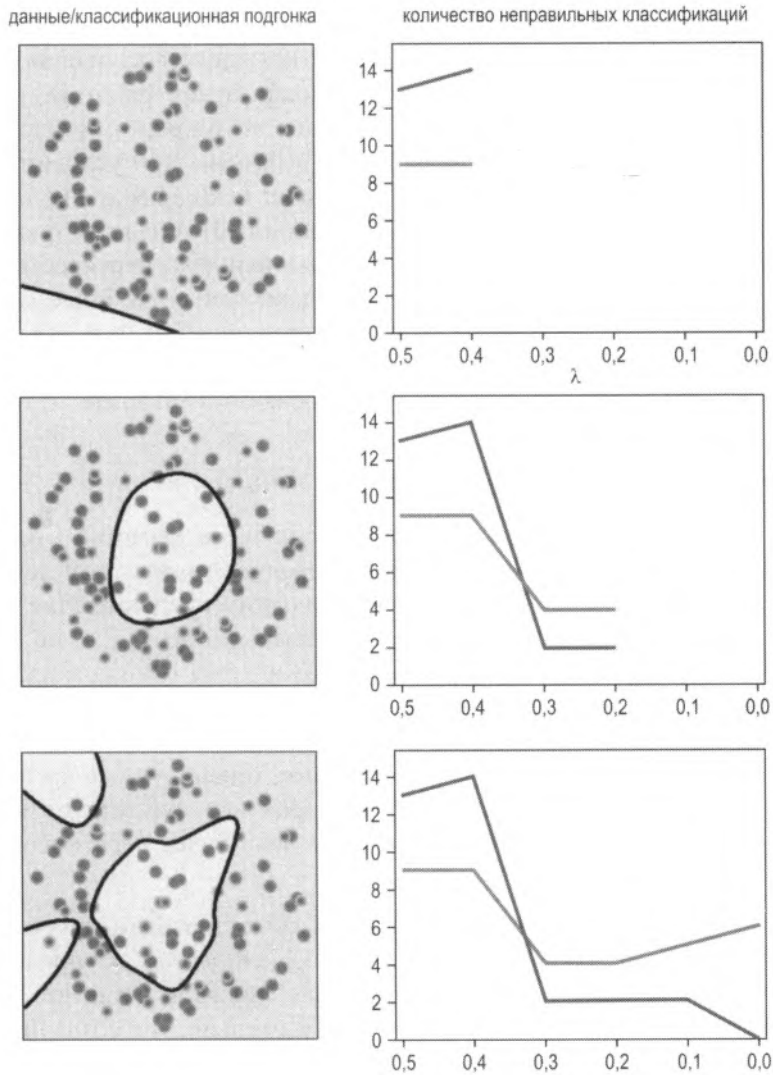


Рис. 11.40. Иллюстрация примера 11.13

веком в отношении регуляризации, как это было в *разд. 9.7*, здесь мы используем регуляризацию как инструмент для перекрестной валидации.

11.7. Тестирование данных

В *разд. 11.3.4* мы увидели, что вместо тренировочной ошибки *валидационная* ошибка является надлежащим измерительным инструментом, позволяющим нам точно идентифицировать надлежащую модельную/параметрическую настройку для обобщенных данных. Однако, как и тренировочная ошибка, отбор модели на основе минимальной валидационной ошибки может также потенциально приводить к моделям, которые являются *переобученными* к нашему исходному набору дан-

ных. Другими словами, по крайней мере в принципе, мы также можем получать переподгонку к валидационным данным. Это может сделать валидационную ошибку слабым индикатором того, насколько хорошо будет работать перекрестно-валидированная модель в целом. Как мы увидим в этом кратком разделе, потенциальные опасности этой реальности могут быть смягчены при условии, если набор данных достаточно велик, путем разделения наших исходных тренировочных данных не на два набора (тренировочный и валидационный), а на *три*: тренировочный, валидационный и тестовый. Измеряя результативность перекрестно-валидированной модели на *тестовом наборе*, мы не только получаем более качественную меру ее способности улавливать истинную природу явления, генерирующего данные, но и обзаводимся надежным измерительным инструментом для сравнения эффективности нескольких перекрестно-валидированных моделей.

11.7.1. Переподгонка к валидационным данным

В *разд. 11.3.4* мы узнали, как тренировочная ошибка в качестве измерительного инструмента не помогает нам надлежаще определять момент, когда настроенная модель имеет достаточную сложность для правильного представления конкретного набора данных. Там мы увидели, как переобученнная модель, т. е. та, которая достигает минимальной тренировочной ошибки, но является слишком сложной, невероятно хорошо представляет тренировочные данные, которые у нас имеются в настоящий момент, но одновременно очень плохо представляет явление, лежащее в основе данных (а также любые будущие данные, генерируемые им аналогичным образом). Хотя и не так сильно превалируя на практике, существует возможность, что надлежащая перекрестно-валидированная модель будет переобучена к *валидационным данным*.

Для того чтобы увидеть, как это может быть, давайте проанализируем экстремальный двухклассовый классификационный набор данных. Как представлено на рис. 11.41, *слева*, этот набор данных не показывает никакой значимой связи между входом и выходом (выходными метками). Действительно, мы создали его, выбрав координаты двумерных входных точек случайно над входным пространством, а затем назначив +1 (красный класс) половине точек (которые отбираются, опять же, случайным образом) и значение метки –1 (синий класс) другой половине.

Поскольку мы знаем, что базовое явление, генерирующее этот набор данных, является *совершенно случайным*, никакая модель, независимо от того, была ли она найдена с помощью перекрестной валидации или иным образом, никогда не позволит нам правильно предсказать метку будущих точек с точностью, существенно превышающей 50%. Другими словами, для таких случайных данных, как эти, ни одна модель не будет действительно обеспечивать точность, превосходящую случайность. Однако эта реальность не обязательно должна быть отражена в соответствующей перекрестно-валидированной модели (т. е. модели с минимальной валидационной ошибкой для некоторого раздела данных). Действительно, на рис. 11.41, *справа* показаны граница решения наивно перекрестно-валидированной модели для этого набора данных, где 1/5 исходных данных была использованы в качестве ва-

лидации, и участки, окрашенные в соответствии с модельными предсказаниями. Эта конкретная перекрестно-валидированная модель обеспечивает 70-процентную точность на валидационных данных, что, возможно, на первый взгляд кажется загадочным, учитывая наше понимание опорного явления. Однако это обусловлено тем, что, несмотря на то, что она была выбрана в качестве модели, которая минимизирует валидационную ошибку, эта модель по-прежнему является переобученной к исходным данным. Хотя она не так превалирует или имеет серьезное влияние, как переподгонка в моделях, минимизирующих тренировочные ошибки, переподгонка к валидационным данным, подобная этой, все-таки представляет опасность, которую на практике следует избегать, когда это возможно.

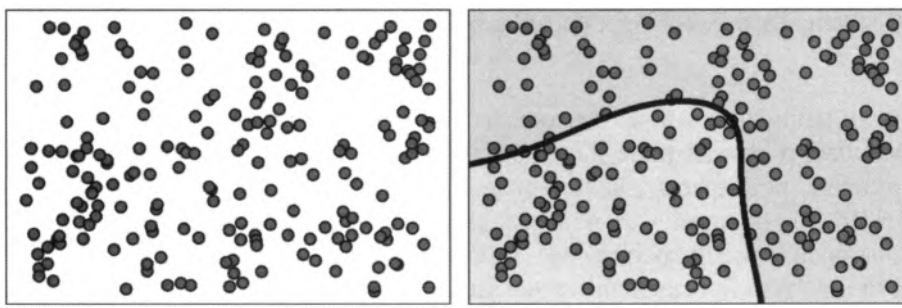


Рис. 11.41. Случайно сгенерированный двухклассовый классификационный набор данных (*слева*). Граница решения перекрестно-валидированной модели, обеспечивающей 70-процентную точность на валидационных данных, что значительно больше случайной вероятности (50%; *справа*)

11.7.2. Тестовые данные и тестовая ошибка

До сих пор мы использовали валидационные данные как для отбора наилучшей для наших данных модели (т. е. для перекрестной валидации), так и для измерения ее качества. Однако, как и в случае, когда понятие валидационных данных было впервые введено в *разд. 11.3.4*, мы видим, что использование одного и того же набора данных для выполнения обеих этих задач может приводить к отбору переобученной модели и убавлять полезность валидационной ошибки как меры качества модели. Решение этой проблемы (опять же, как и в начале, когда мы ввели понятие валидационных данных) состоит в том, чтобы разбить две задачи, которые мы сейчас назначаем валидационным данным, путем введения *второго* валидационного набора. Этот "второй валидационный набор" часто называется *тестовым набором* или *тестирущим набором* и используется исключительно для измерения качества нашей окончательной перекрестно-валидированной модели.

Разбив наши данные на три части (как показано на рис. 11.42), мы по-прежнему используем тренировочную и валидационную части точно так же, как они использовались до сих пор (т. е. для выполнения перекрестной валидации). Однако после того как перекрестно-валидированная модель построена, ее качество измеряется на отдельном *тестовом наборе данных*, на котором она не была ни натренирована, ни

валидирована. Эта *тестовая ошибка* дает "несмещенную" оценку результативности перекрестно-валидированной модели и, как правило, ближе к улавливанию истинной ошибки нашей модели на будущих данных, генерируемых тем же самым явлением.

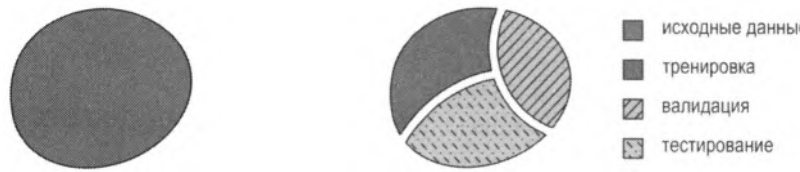


Рис. 11.42. Исходный набор данных (*слева*) случайным образом разбивается на три непересекающихся подмножества: тренировочный, валидационный и тестовый наборы (*справа*)

В ситуации наших случайных двухклассовых данных, представленных в *разд. 11.7.1*, тестовый набор такого рода дает гораздо более точную картину того, насколько хорошо наша перекрестно-валидированная модель будет работать в целом. На рис. 11.43 снова показан этот набор данных (причем точки валидационных данных выделены оранжевыми границами), теперь расширенный за счет добавления тестовой части (это точки, выделенные зелеными границами), которая генерируется точно так же, как был создан исходный набор данных на рис. 11.41. Важно отметить, что эта тестовая часть не использовалась во время тренировочной перекрестной валидации. В то время как наша перекрестно-валидированная модель достигла 70-процентной точности на валидационном наборе (как упоминалось ранее), она достигает лишь 52-процентной точности на тестовом наборе, что является более реалистичным индикатором истинной классификационной способности нашей модели, учитывая природу этих данных.

Какая часть исходного набора данных должна быть отнесена к нашему тестовому набору? Как и в случае с разбивкой на тренировочную и валидационную части

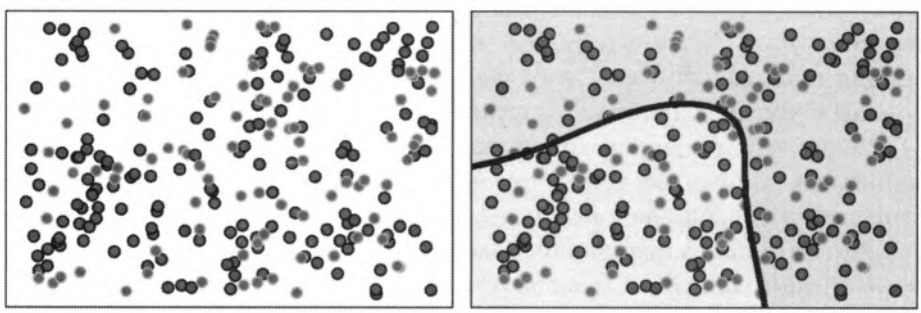


Рис. 11.43. Тот же набор данных, впервые показанный на рис.11.41, расширенный точками данных, выделенными зеленым цветом, которые были удалены из тренировочной и валидационной процедур и оставлены для тестирования (*слева*). Перекрестно-валидированная модель достигает лишь 52-процентной точности на тестовом наборе, т. е. гораздо лучшей оценки способности модели машинного обучения классифицировать случайные данные (*справа*)

(подробно описаным в *разд. 11.3.4*), здесь нет общего правила, за исключением, возможно, одного: использование тестовых данных — это роскошь, которой можно предаваться только тогда, когда у нас есть крупный объем данных. Когда же объем данных является весьма скудным, мы обязаны все это задействовать лишь для того, чтобы построить "наполовину разумную" перекрестно-валидированную модель. Однако, когда данных в изобилии, часто размер валидационного и тестового наборов выбирается схожим образом. Например, если 1/5 набора данных используется для валидации, то часто для простоты такая же часть используется и для тестирования.

11.8. Какой универсальный аппроксиматор лучше всего работает на практике?

За исключением особых обстоятельств, таких как те, которые кратко обсуждаются далее, обычно априори никогда нет никакой ясности, какой из универсальных аппроксиматоров будет работать лучше всего, если таковой имеется. Действительно, перекрестная валидация (как описано в предыдущих разделах) — это применяемый на практике инструментарий для решения, какая модель на основе какого типа универсального аппроксиматора лучше всего подходит для конкретной задачи. Используя эти методы, можно создать серию разных перекрестно-валидированных моделей, каждая из которых выстроена из отдельного типа универсального аппроксиматора, а затем сравнить их эффективность на тестовом наборе (описанном в *разд. 11.7*), чтобы увидеть, какой универсальный аппроксиматор работает лучше всего. В качестве альтернативы можно перекрестно валидировать ряд моделей, основанных на универсальных аппроксиматорах, и затем усреднить их, как описано далее в *разд. 11.9*, в итоге получив усредненную модель, состоящую из представителей нескольких универсальных аппроксиматоров.

В некоторых случаях широкое понимание набора данных может направлять выбор конкретного универсального аппроксиматора. Например, поскольку нередко бизнес-данные, данные переписи населения и (в более общем плане) *структурированные* наборы данных состоят из мешанины непрерывных и прерывистых категориальных входных признаков (*см. разд. 6.7.1*), древесные универсальные аппроксиматоры с их прерывистыми скачкообразными контурами часто дают в среднем более сильные результаты, чем другие типы универсальных аппроксиматоров. С другой стороны, данные, которые естественным образом являются непрерывными (например, данные, генерируемые естественными процессами, или сенсорные данные), часто лучше согласуются с непрерывным универсальным аппроксиматором: фиксированно-контурным или нейросетевым. Понимание того, нужно ли делать будущие предсказания *внутри* или *вне* входной области значений исходного набора данных, также помогает в выборе аппроксиматора. В таких случаях фиксированно-контурные или нейросетевые аппроксиматоры могут быть предпочтительнее древесных — последние по самой своей природе всегда создают совершенно плоские предсказания за пределами входной области значений исходных данных (дальнейшие подробности см. в упражнении 14.9).

Когда *интерпретируемость* работы модели *человеком* имеет первостепенное значение, это желание (при определенных обстоятельствах) может управлять выбором универсального аппроксиматора. Например, из-за их дискретной ветвящейся структуры (*см. разд. 14.2*) древесные универсальные аппроксиматоры интерпретируются гораздо легче, чем другие аппроксиматоры (в особенности нейросетевые). По аналогичным причинам фиксированно-контурные аппроксиматоры (например, многочлены) часто задействуются в естественных науках, как, например, гравитационное явление, лежащее в основе рампового набора данных Галилея, рассмотренного в примере 11.17.

11.9. Бутстрап-агрегирование перекрестно-валидированных моделей

Как мы подробно обсуждали в *разд. 11.3.4*, валидационные данные — это та часть нашего исходного набора данных, которую мы случайным образом исключаем из тренирочного процесса, чтобы определить надлежаще настроенную модель, которая будет точно представлять явление, генерирующее наши данные. Валидационная ошибка, генерируемая нашей настроенной моделью на этой "не встречавшейся ранее" части данных, является основополагающим измерительным инструментом, который мы используем для определения подходящей перекрестно-валидированной модели для всего нашего набора данных (кроме, возможно, тестового набора, *см. разд. 11.7*). Однако случайный характер разбивки данных на тренировочные и валидационные является очевидным недостатком нашего процесса перекрестной валидации: что произойдет, если случайная разбивка создает тренировочную и валидационную части, которые не являются желательными представителями опорного явления, которое их сгенерировало? Другими словами, что делать на практике с потенциально плохой тренировочно-валидационной разбивкой, которая может привести к перекрестно-валидированным моделям со слабой репрезентативностью?

Поскольку мы на самом деле нуждаемся в перекрестной валидации, чтобы выбирать подходящие модели в целом, и поскольку мы ничего не можем сделать с (случайной) *природой*, на основе которой мы разбиваем наши данные для перекрестной валидации (существует ли еще более качественный метод, который симулировал бы "будущее" нашего явления?), практическое решение этой фундаментальной проблемы состоит в том, чтобы просто создать несколько разных тренировочно-валидационных частей, определить соответствующую перекрестно-валидированную модель на каждой части, а затем *усреднить* полученные перекрестно-валидированные модели. Применив процедуру усреднения набора перекрестно-валидированных моделей, которая на жаргоне машинного обучения также называется бутстрап-агрегированием, или бэггингом[14], мы можем очень часто "усреднять" по-

[14] Бэггинг (bagging — от *англ.* bootstrap aggregating, т. е. бутстрап-агрегирование, или агрегирование бутстраповских выборок) — это общая методика формирования коллекции моделей путем бутстрапирования данных с их последующим агрегированием, при этом в качестве агрегирующей функции, как правило, используется среднее арифметическое, медиана или мода. — *Прим. пер.*

тенциально нежелательные характеристики каждой модели, одновременно синергезируя их положительные характеристики. Кроме того, с помощью *бутстрап-агрегирования* мы также можем эффективно совмещать перекрестно-валидированные модели, построенные из *разных* универсальных аппроксиматоров. И действительно, это наиболее разумный способ создания единой модели, построенной из разных типов универсальных аппроксиматоров на практике.

Здесь мы рассмотрим концепцию бутстрап-агрегирования или модельного усреднения для регрессии, а также для двухклассовой и многоклассовой классификации, разведав массив простых примеров. На этих простых примерах мы наглядно проиллюстрируем превосходную результативность бутстрап-агрегированных моделей, но в целом мы подтвердим это, используя понятие тестовой ошибки (*см. разд. 11.7*) или оценивания тестовой ошибки (часто задействуемой при бутстрап-агрегировании деревьев, *см. разд. 14.6*). Тем не менее принципы, описанные здесь, могут быть задействованы шире и для решения любой задачи машинного обучения. Как мы увидим, наилучший способ усреднить/агрегировать набор перекрестно-валидированных регрессионных моделей — это взять их *медиану*, а наилучший способ усреднить/агрегировать набор перекрестно-валидированных классификационных моделей — вычислить *моду* их предсказанных меток.

11.9.1. Бутстрап-агрегирование регрессионных моделей

Разведаем несколько способов агрегирования набора перекрестно-валидированных моделей для нелинейно-регрессионного набора данных, впервые описанного в примере 11.6. Как мы увидим, чаще всего лучший способ агрегировать (или усреднить) перекрестно-валидированные регрессионные модели состоит в том, чтобы взять их *медиану* (в отличие от их среднего значения).

Пример 11.14. Бутстрап-агрегирование перекрестно-валидированных регрессионных моделей

В наборе небольших графиков на рис. 11.44, *слева* мы показываем десять разных тренировочно-валидационных разбивок прототипического нелинейно-регрессионного набора данных, где 4/5 данных в каждом случае были использованы для тренировки (окрашены светло-синим цветом) и 1/5 — для валидации (окрашены желтым цветом). На графике при каждой разбивке исходных данных построена соответствующая перекрестно-валидированная модель, найденная посредством наивной перекрестной валидации (*см. разд. 11.4.2*) всего диапазона многочленных моделей степени от 1 до 20. Как мы видим, хотя *многие* из этих перекрестно-валидированных моделей работают довольно хорошо, некоторые из них (из-за особой тренировочно-валидационной разбивки, на которой они основаны) сильно *недообучены* или *переобучены* к исходному набору данных. В каждом случае слабая результативность полностью обусловлена конкретной опорной (случайной) тренировочно-валидационной разбивкой, которая приводит перекрестную валидацию к минимизирующей валидационной ошибке настроенной модели, которая по-прежнему не очень хорошо представляет истинное опорное явление. Взяв *среднее* значение (здесь *медиану*) из десяти перекрестно-валидированных моделей, показанных на

этих небольших панелях, мы можем усреднить слабую результативность этой горстки плохих моделей, придя к окончательной агрегированной модели, которая довольно хорошо обучена к данным, как показано на большом графике на рис. 11.44, *справа*.

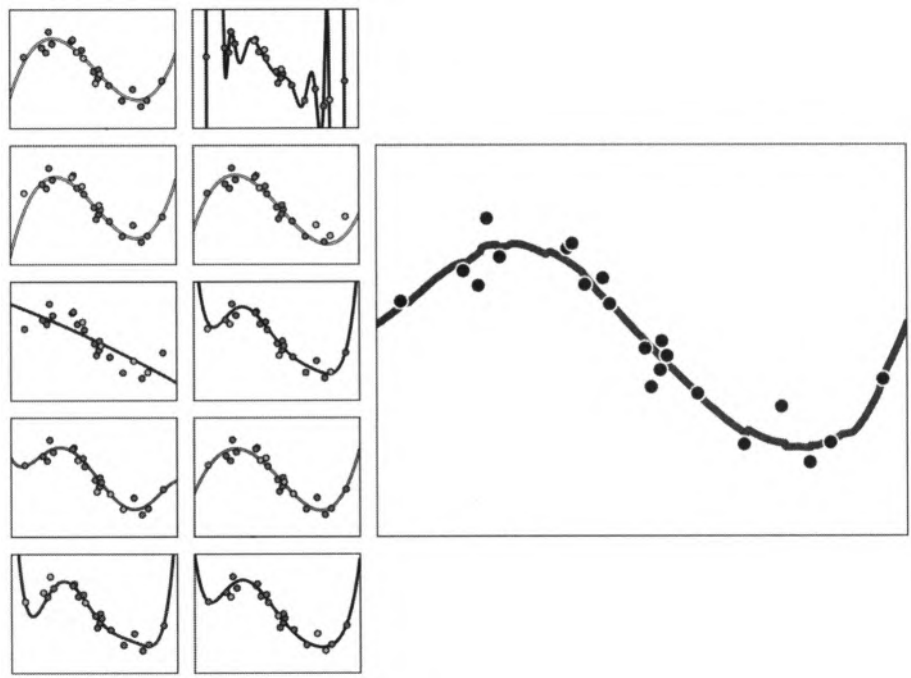


Рис. 11.44. Иллюстрация примера 11.14. Результаты применения перекрестной валидации к десяти случайным тренировочно-валидационным разбивкам нелинейно-регрессионного набора данных, причем каждая результирующая модель показана на одной из десяти небольших панелей. Здесь тренировочная и валидационная части в каждом случае окрашены соответственно в светло-синий и желтый цвета (*слева*). Подгонка, показанная красным цветом, полученная в результате бутстрап-агрегирования десяти моделей, чьи подгонки показаны слева (*справа*)

Зачем усреднять наши перекрестно-валидированные модели, используя *медиану*, а не *среднее* значение? Все дело в том, что среднее значение гораздо *чувствительнее* к выбросам, чем медиана. В верхнем ряду рис. 11.45 показаны регрессионный набор данных, приводившийся ранее, а также отдельные перекрестно-валидированные подгонки (*слева*), медианно-агрегированная модель (*в центре*) и среднезначно-агрегированная модель (*справа*). Здесь среднезначная модель сильно зависит от нескольких переобученных моделей в группе, и в итоге оказывается слишком осциллирующей, чтобы справедливо представлять явление, лежащее в основе данных. Медиана не подвержена такому влиянию и поэтому является гораздо более репрезентативной.

Когда мы агрегируем, мы просто усредняем различные перекрестно-валидированные модели с желанием избежать плохих аспектов низкорезультативных моделей, а также задействовать совместно сильные элементы высокорезультативных

моделей. Ничего в этом подходе не мешает нам агрегировать перекрестно-валидированные модели, построенные с использованием разных универсальных аппроксиматоров, и, действительно, это наиболее организованный способ совмещения разных типов универсальных аппроксиматоров на практике.

В нижнем ряду рис. 11.45 представлен результат перекрестно-валидированной многочленной модели (первый график), перекрестно-валидированной нейросетевой модели (второй график) и перекрестно-валидированной древесной модели (третий график), построенных посредством бустинга (*см. разд. 11.5*). Каждая перекрестно-валидированная модель использует разную тренировочно-валидационную разбивку исходного набора данных. Агрегированная медиана этих моделей показана на четвертом графике.

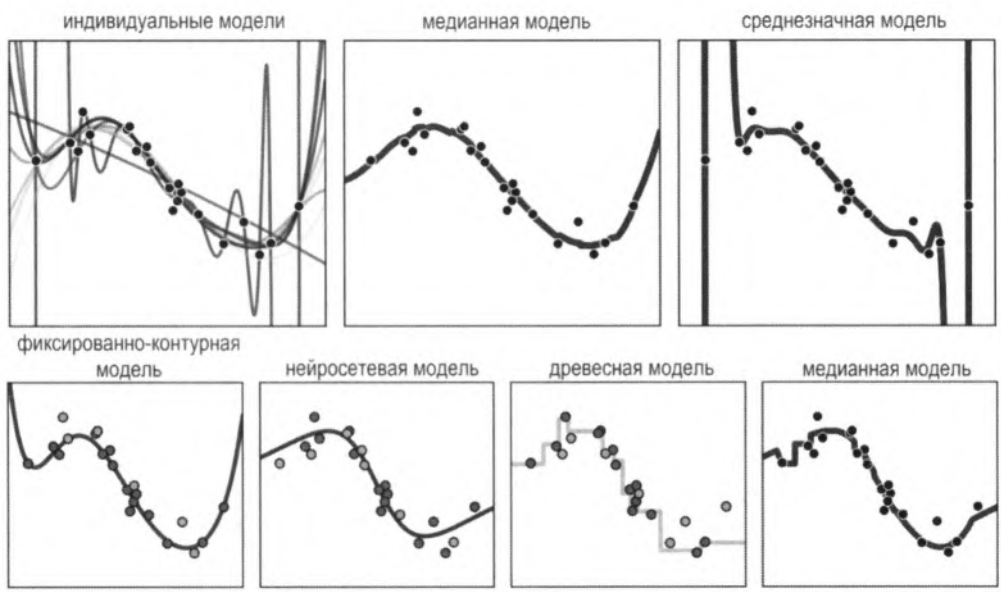


Рис. 11.45. Иллюстрация примера 11.14. Десять отдельных перекрестно-валидированных моделей, впервые показанных на рис. 11.44, графики которых построены вместе (*слева сверху*), медиана (*в центре сверху*) и среднее значение (*справа сверху*) этих моделей. В случае регрессии бутстрап-агрегирование посредством медианы дает более качественные результаты, поскольку она менее чувствительна к выбросам. В нижнем ряду: перекрестно-валидированные фиксированно-контурная многочленная (первый график), нейросетевая (второй график) и древесная (третий график) модели, а также медиана этих трех моделей (четвертый график)

11.9.2. Бутстрап-агрегирование классификационных моделей

Принцип, лежащий в основе агрегирования перекрестно-валидированных моделей, соблюдается аналогичным образом для классификационных задач так же, как и в случае регрессии. Поскольку мы не можем быть уверены в том, представляет ли конкретный (случайно выбранный) валидационный набор "будущие данные" из рассматриваемого явления точно, усреднение (или бутстрап-агрегирование) ряда

перекрестно-валидированных классификационных моделей обеспечивает способ усреднения слабо репрезентативных частей некоторых моделей при одновременном совмещении положительных характеристик различных моделей.

Поскольку предсказываемый результат классификационной (перекрестно-валидированной) модели представляет собой *дискретную* метку, усредненным значением, используемым для бутстрап-агрегирования перекрестно-валидированных классификационных моделей, является *мода* (т. е. наиболее часто предсказываемая метка).

Пример 11.15. Бутстрап-агрегирование перекрестно-валидированных двухклассовых классификационных моделей

В наборе небольших графиков на рис. 11.46, *слева* показаны пять разных тренировочно-валидационных разбивок прототипического двухклассового классифика-

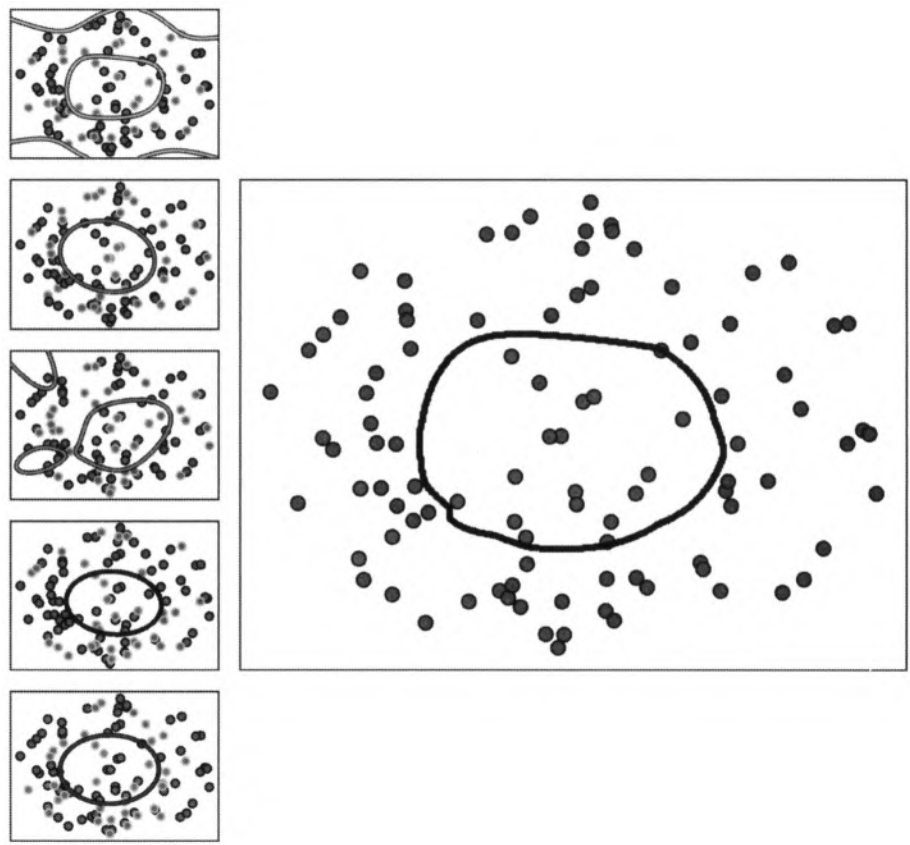


Рис. 11.46. Иллюстрация примера 11.15. Пять моделей перекрестно-валидированных на случайных тренировочно-валидационных разбивках данных, причем валидационные данные в каждом экземпляре выделены оранжевым контуром. Соответствующая нелинейная граница решения, обеспечиваемая каждой моделью, показана на каждой панели. Некоторые модели из-за разбивки данных, на которых они были построены, сильно переобучены (*слева*). Исходный набор данных с границей решения, обеспечиваемой агрегацией (т. е. модой) пяти перекрестно-валидированных моделей (*справа*)

ционного набора данных, впервые описанного в примере 11.7, где 2/3 данных в каждом экземпляре используются для тренировки и 1/3 — для валидации (границы этих точек окрашены оранжевым цветом). При каждой разбивке исходных данных на графике строится нелинейная граница решения, соответствующая каждой перекрестно-валидированной модели, найденной посредством наивной перекрестной валидации всего диапазона многочленных моделей степени с 1-й по 8-ю. Многие из этих перекрестно-валидированных моделей показывают довольно хорошую результативность, но некоторые из них (из-за конкретной тренировочно-валидационной разбивки, на которой они основаны) сильно *переобучены* к исходному набору данных. Путем бутстрап-агрегирования этих моделей с использованием наиболее популярных предсказаний для назначения меток (т. е. *моды* этих перекрестно-валидированных модельных предсказаний) мы порождаем соответствующую границу решения для данных, показанных на рис. 11.46, *справа*.

На рис. 11.47, *в центре сверху* изображены границы решения пяти перекрестно-валидированных моделей, каждая из которых построена с использованием $B = 20$ нейросетевых единиц, натренированных на разных тренировочно-валидационных разбивках набора данных, показанного на рисунке слева. В каждом случае 1/3 набора данных выбирается случайно в качестве валидации (выделено желтым цве-

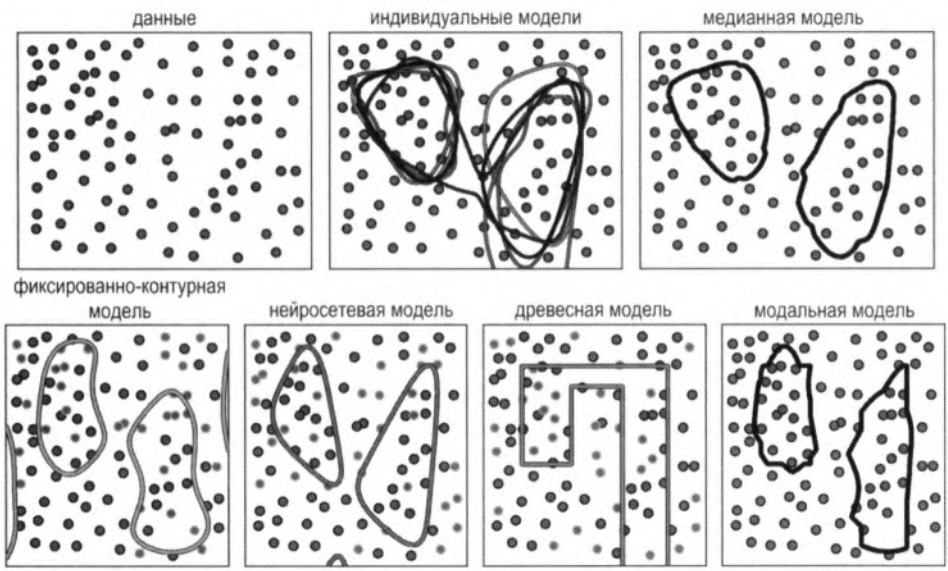


Рис. 11.47. Иллюстрация примера 11.15. Игрушечный двухклассовый классификационный набор данных, впервые описанный в примере 11.7 (*слева сверху*). Границы решения, каждая из которых показана другим цветом, получающиеся в результате перекрестной валидации пяти моделей на разных тренировочно-валидационных разбивках данных (*в центре сверху*). Граница решения, вытекающая из моды (*модальной модели*) пяти индивидуальных перекрестно-валидированных моделей (*справа сверху*). В нижнем ряду: границы решения, обеспечиваемые перекрестно-валидированными многочленной фиксированно-контурной моделью (первый график), нейросетевой моделью (второй график) и древесной моделью (третий граик). В каждом случае валидационная часть данных выделена желтым цветом. Граница решения, обеспечиваемая модой этих трех моделей (четвертый график)

том). В то время как некоторые из усвоенных границ решения (показанных сверху в центре) разделяют эти два класса довольно хорошо, другие справляются с этой работой хуже. На рис. 11.47, *справа сверху* показана граница решения агрегации, созданной путем взятия моды предсказаний из этих перекрестно-валидированных моделей, которая демонстрирует довольно хорошую результативность.

Как и в случае регрессии, при классификации мы также можем совмещать перекрестно-валидированные модели, построенные из разных универсальных аппроксиматоров. Этот подход проиллюстрирован в нижнем ряду на рис. 11.47, с помощью того же самого набора данных. В частности, показан результат перекрестно-валидированной многочленной модели (первый график), перекрестно-валидированной нейросетевой модели (второй график) и перекрестно-валидированной древесной модели (третий график). Каждая перекрестно-валидированная модель использует разную тренировочно-валидационную разбивку исходных данных, и агрегация (мода) этих моделей, показанная на четвертом графике, демонстрирует довольно хорошую результативность.

Пример 11.16. Бутстрап-агрегирование перекрестно-валидированных многоклассовых классификационных моделей

В этом примере проиллюстрировано бутстрап-агрегирование различных перекрестно-валидированных многоклассовых классификационных моделей на двух разных наборах данных, показанных на рис. 11.48, *слева*. В каждом случае мы наивно

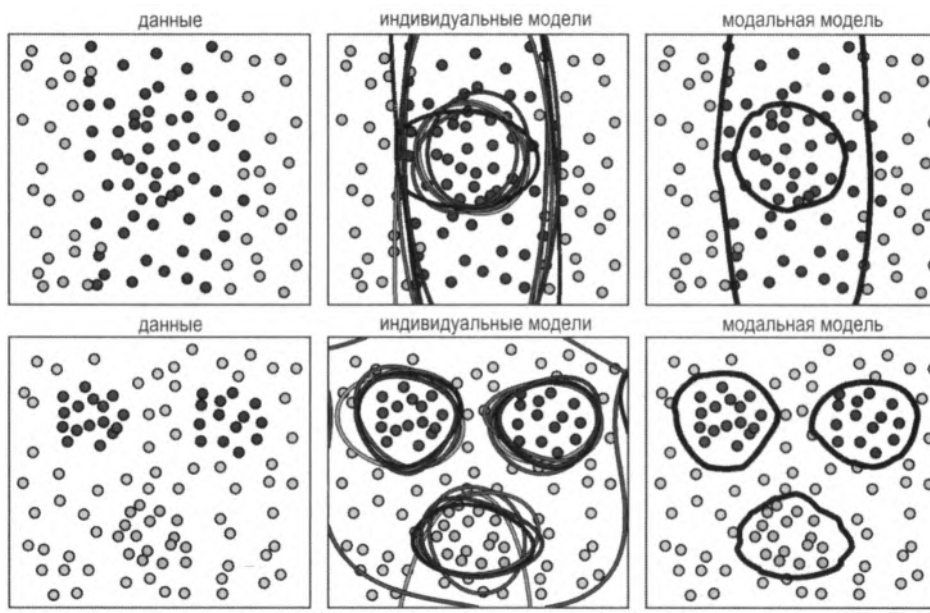


Рис. 11.48. Иллюстрация примера 11.16. Два многоклассовых классификационных набора данных (*слева*). Границы решения, получаемые в результате из пяти перекрестно-валидированных моделей, каждая из которых отображена отдельным цветом (*в центре*). Граница решения, обеспечиваемая бутстрап-агрегированием (*справа*)

перекрестно-валидируем многочленные модели степеней 1–5, в общей сложности усваивая пять перекрестно-валидированных моделей. В центре рисунка отображены границы решения, предоставляемые каждой перекрестно-валидированной моделью, отдельными цветами, тогда как граница решения окончательной *модальной модели* показана справа для каждого набора данных.

11.9.3. Сколько моделей мы должны агрегировать на практике?

Обратите внимание, что в примерах этого раздела точное число бутстрап-агрегированных перекрестно-валидированных моделей было установлено несколько произвольно. Как и в отношении других важных параметров, связанных с перекрестной валидацией (например, частью набора данных, резервируемой для валидации), не существует волшебного числа (перекрестно-валидированных моделей), обычно используемого на практике для бутстрап-агрегирования. В идеале, если бы мы знали, что любая случайная валидационная часть набора данных обычно хорошо представляет их, что нередко является верным для очень крупных наборов данных, было бы меньше необходимости в ансамблировании многочисленных перекрестно-валидированных моделей, где каждая из них тренировалась на разной тренировочно-валидационной разбивке исходных данных. Действительно, в таких случаях мы могли бы объединить ряд моделей, усвоенных на одной тренировочно-валидационной разбивке, чтобы достичь схожих улучшений по отношению к одной-единственной модели. С другой стороны, чем меньше мы могли бы доверять достоверности случайной валидационной части в представлении явления в целом, тем меньше мы могли бы доверять индивидуальной перекрестно-валидированной модели, и поэтому мы, возможно, захотим агрегировать их больше, чтобы помочь усреднить наши низкорезультативные модели, получаемые после плохих разбивок данных. Часто на практике такие факторы, как вычислительная мощность и размер набора данных, определяют ответ на вопрос: использовать бутстрап-агрегирование или нет? И если ответ "да", то сколько моделей задействовать в среднем?

11.9.4. Ансамблирование: бутстрап-агрегирование против бустинга

Описанная здесь техника бутстрап-агрегирования, в которой мы совмещаем несколько разных моделей, каждая из которых проходит перекрестную валидацию независимо от других, является первостепенным примером того, что на жаргоне машинного обучения называется ансамблированием. Метод *ансамблирования* (как следует из термина "ансамбль") обычно относится к любому методу совмещения разных моделей в контексте машинного обучения. Бутстрап-агрегирование (бэггинг), безусловно, относится к этой *общей* категории, как и общий подход к перекрестной валидации — бустинг, описанный в *разд. 11.5*. Однако эти два метода ансамблирования очень отличаются друг от друга.

В случае бустинга мы создаем *одну-единственную* перекрестно-валидированную модель путем постепенного сложения простых моделей, состоящих из одного универсального аппроксиматора (*см. разд. 11.5.4*). Каждая из составляющих моделей, участвующих в бустинге, тренируется таким образом, что каждая отдельная модель

зависит от своих предшественников (которые тренируются в первую очередь). С другой стороны, в случае бутстрап-агрегирования (как мы уже видели) мы усредняем *несколько перекрестно-валидированных* моделей, которые были натренированы *независимо* друг от друга. Действительно, любая из этих перекрестно-валидированных моделей в агрегированном ансамбле сама по себе может быть бустированной моделью.

11.10. *K*-блочная перекрестная валидация

В этом разделе мы подробно остановимся на понятии ансамблирования, именуемом *K*-блочной перекрестной валидацией, которая часто применяется, когда *интерпретируемость* конечной модели *человеком* имеет существенное значение. В то время как ансамблирование часто обеспечивает более подходящий усредненный предсказатель, который позволяет избежать потенциальных ловушек любой индивидуальной перекрестно-валидированной модели, интерпретируемость человеком обычно теряется, поскольку окончательная модель представляет собой среднее многих потенциально очень разных нелинейностей[15]. Вместо усреднения множества перекрестно-валидированных моделей над многочисленными разбивками данных, каждая из которых обеспечивает минимальную валидационную ошибку над соответствующей разбивкой, в случае *K*-блочной перекрестной валидации мы выбираем одну-единственную модель, имеющую минимальную *среднюю валидационную ошибку* над всеми разбивками данных. Это дает потенциально менее точную окончательную модель, но значительно более простую (чем ансамблевая модель) и более понятную людям. Как мы увидим, в специальных приложениях *K*-блочная перекрестная валидация используется и с *линейными* моделями.

11.10.1. Процедура *K*-блочной перекрестной валидации

K-блочная перекрестная валидация — это метод определения устойчивых перекрестно-валидированных моделей посредством процедуры, похожей на ансамблирование, которая ограничивает сложность окончательной модели, благодаря чему она становится более *интерпретируемой человеком*. Вместо усреднения группы перекрестно-валидированных моделей, каждая из которых достигает минимальной валидационной ошибки над случайной тренировочно-валидационной разбивкой данных, в случае *K*-блочной перекрестной валидации мы выбираем одну-единственную окончательную модель, которая достигает *наименьшей* средней валидационной ошибки над всеми разбивками, вместе взятыми. Отбирая одну-единственную модель для представления всего набора данных, в отличие от среднего значения разных моделей (как это делается при ансамблировании), мы облегчаем интерпретацию отобранной модели.

Конечно, желание, чтобы любая нелинейная модель была интерпретируемой, означает, что ее фундаментальные строительные блоки (универсальные аппроксимато-

[15] Пневые/древесные аппроксиматоры иногда являются исключением из этого общего правила, как подробно описано в *разд. 14.2*.

ры определенного типа) также должны быть интерпретируемыми. Нейросетевые аппроксиматоры, например, почти никогда не поддаются интерпретации человеком, в то время как фиксированно-контурные аппроксиматоры (чаще всего многочлены) и древесные аппроксиматоры (обычно пни) могут быть интерпретированы в зависимости от решаемой задачи. Отсюда, последние два типа универсальных аппроксиматоров чаще всего задействуются вместе с K-блочным методом.

Для того чтобы еще больше упростить окончательный результат этой процедуры, вместо использования совершенно случайных тренировочно-валидационных разбивок (как это делается с ансамблированием), мы разбиваем данные случайным образом на набор K непересекающихся фрагментов. Это наглядно показано на рис. 11.49, где исходные данные разбиты на $K = 3$ непересекающихся множества.

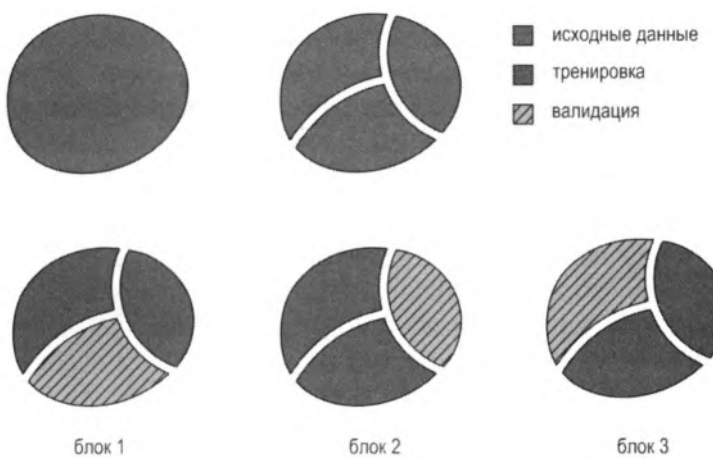


Рис. 11.49. Схематическая иллюстрация K-блочной перекрестной валидации для $K = 3$

Затем мы циклически перебираем K тренировочно-валидационных разбивок данных, состоящих из $K - 1$ тренировочных частей и финальной валидационной части, что позволяет каждой точке набора данных принадлежать к валидационному набору ровно один раз. Каждая такая разбивка называется *блоком*, которых в общей сложности существует K, отсюда и название "K-блочная" перекрестная валидация. На каждом блоке мы перекрестно валидируем один и тот же набор моделей и записываем валидационный балл каждой из них. После этого мы выбираем одну-единственную налучшую модель, которая дала наименьшую *среднюю валидационную ошибку*. Как только это будет сделано, выбранная модель будет перетренирована над всем набором данных, чтобы обеспечить окончательный настроенный предсказатель (предиктор) данных.

Поскольку этой процедурой никакие модели не совмещаются/усредняются, она может очень легко производить менее точные модели (с точки зрения *тестовой ошибки*, описанной в *разд. 11.7*) для общих задач обучения по сравнению с ансамблированием. Однако, когда интерпретируемость модели человеком затмевает по-

требности в исключительной результативности, K-блочная перекрестная валидация создает более эффективную модель, чем одна-единственная перекрестно-валидированная модель, которая все еще может пониматься людьми. Это в некоторой степени аналогично истории отбора признаков, подробно описанной в *разд. 9.6* и *9.7*, где интерпретируемость человеком является направляющим мотиватором (а не просто предсказательной силой).

Пример 11.17. Гравитационный эксперимент Галилея

В этом примере мы используем K-блочную перекрестную валидацию на наборе данных Галилея, подробно описанном в примере 10.2, чтобы восстановить квадратичное правило, которое было там сконструировано и которое сам Галилей угадал из аналогичного набора данных. Поскольку в этом наборе данных есть только $P = 6$ точек, интуиция подсказывает, что мы используем большое значение для K, как описано в *разд. 11.3.4*. В этом случае мы можем установить K как можно выше, т. е. $K = P$, имея в виду, что каждый блок будет содержать только одну точку данных для валидационных целей. Эта настройка K-блочной перекрестной валидации — иногда именуемая перекрестной валидацией с исключением по одному — обычно задействуется, когда размер данных чрезвычайно мал.

Здесь мы производим поиск над многочленными моделями степеней с 1-й по 6-ю, поскольку они не только легко интерпретируются, но и подходят для данных, почерпнутых из физических экспериментов (которые часто прослеживают гладкие правила). Как показано на рис. 11.50, хотя не все модели над шестью блоками хорошо вписываются в данные, модель, выбранная K-блочной перекрестной валидацией, действительно является квадратичной многочленной подгонкой, первоначально предложенной Галилеем.

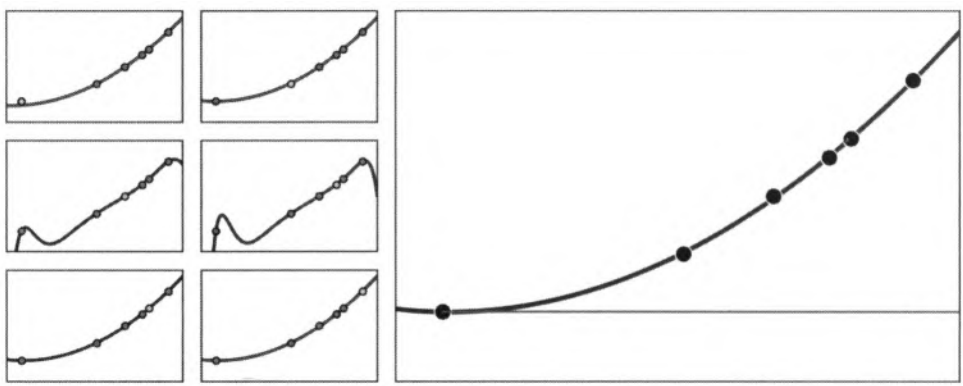


Рис. 11.50. Иллюстрация примера 11.17. Шесть перекрестно-валидированных моделей, каждая из которых натренирована над всеми точками набора данных, кроме одной. Здесь валидационная часть каждого блока (т. е. единственная точка данных) выделена желтым цветом (*слева*). Модель с наименьшей средней валидационной ошибкой является квадратичной (*справа*)

11.10.2. *K*-блочная перекрестная валидация и высокоразмерное линейное моделирование

На мгновение предположим, что у нас есть высокоемкостная модель (например, многочлен степени D, где число D очень велико), которая обеспечивает возможность нескольких видов переобученного поведения для нелинейно-регрессионного набора данных, причем каждый переобученный экземпляр модели обеспечивается разными настройками линейно-комбинационных весов модели. Этот сценарий проиллюстрирован на рис. 11.51, *слева*, где две настройки такой модели обеспечивают два отдельных переобученных предсказателя для обобщенного нелинейно-регрессионного набора данных. Как мы узнали в *разд. 10.2*, любая[16] *нелинейная* модель в исходном пространстве регрессионного набора данных соответствует *линейной* модели в пространстве преобразованных признаков (т. е. пространстве, где каждая отдельная входная ось задается одним из выбранных нелинейных признаков). Поскольку наша модель легко становится переобученной к исходным данным, наши данные лежат в пространстве преобразованных признаков вдоль *линейного подпространства*, которое может быть обучено совершенно с использованием многочисленных разных гиперплоскостей. Действительно, две нелинейные переобученные модели, показанные на рисунке слева, однозначно соответствуют двум линейным подгонкам в пространстве преобразованных признаков — проиллюстрированным символически[17] на рисунке справа.

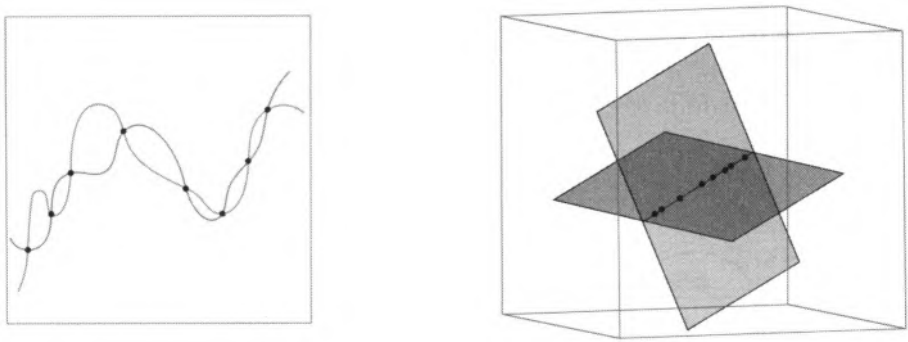


Рис. 11.51. Два примера переподгонки высокоемкостной модели к нелинейно-регрессионному набору данных (*слева*). Эти две модели, рассматриваемые в пространстве преобразованных признаков, являются линейными (*справа*)

Общий сценарий, изображенный на рис. 11.51, *справа* — это именно то, с чего мы начинаем, когда сталкиваемся с малыми наборами данных, имеющими очень высокую входную размерность: в таких сценариях даже линейная модель имеет чрезвычайно высокую емкость и может легко стать переобученной, практически исклю-

[16] Предположим, что любые параметры, внутренние для признаков (если они существуют), фиксированы.

[17] В действительности мы не смогли визуализировать это пространство, т. к. оно было бы слишком высокоразмерным.

чив использование более сложных нелинейных моделей. Поэтому в таких сценариях в целях правильной настройки параметров линейной (высокоемкостной) модели мы часто обращаемся к *регуляризации*, чтобы блокировать емкость в высокоемкостных моделях (как описано в *разд. 11.6.3*). Учитывая малый объем данных, используемых для определения наилучшей настройки регуляризационного параметра, K-блочная перекрестная валидация обычно задействуется для определения надлежащего значения регуляризационного параметра и в конечном счете параметров линейной модели.

Этот сценарий обеспечивает интересную точку пересечения с понятием *отбора признаков посредством регуляризации*, подробно описанным в *разд. 9.7*. Задействуя ℓ_1-регуляризатор, мы можем блокировать емкость нашей *высокоемкостной* линейной модели, одновременно отбирая при этом важные входные признаки, тем самым способствуя интерпретируемости усвоенной модели человеком.

Пример 11.18. Полногеномные исследования ассоциаций

Полногеномные исследования ассоциаций (Genome-wide association studies, GWAS) направлены на понимание связей между десятками тысяч генетических маркеров (входных признаков), берущихся из всего генома человека у нескольких испытуемых, с такими медицинскими диагнозами, как высокое кровяное давление, высокий уровень холестерина, заболевания сердца, диабет, различные формы рака и многие другие (рис. 11.52). Эти исследования обычно предусматривают относительно малое число пациентов с данным заболеванием (по сравнению с очень большим размером входных данных). Как результат, перекрестная валидация на основе регуляризации является полезным инструментом для усвоения значимых (линейных)

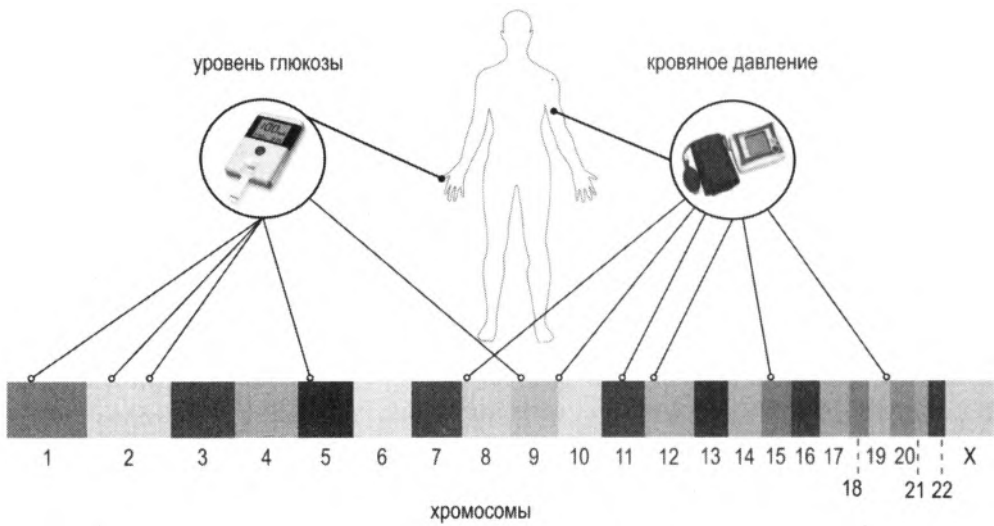


Рис. 11.52. Концептуальная иллюстрация общегеномного исследования ассоциаций, в котором количественный биологический признак (например, кровяное давление или уровень глюкозы) должен быть связан с конкретными геномными локациями

моделей для таких данных. Более того, использование регуляризатора (индуцирующего разреженность), такого как ℓ_1-норма, может помочь исследователям идентифицировать горстку генов, критически важных для изучаемого заболевания, что может улучшить его понимание, а также, возможно, дать толчок развитию генно-ориентированной терапии. Дальнейшие подробности см. в упражнении 11.10.

Пример 11.19. Исследования фМРТ

Нейробиологи считают, что только малое число активных участков головного мозга вовлечено в выполнение той или иной когнитивной задачи. Поэтому ограничение числа входных признаков, допустимых в классификационной модели, посредством *отбора ℓ_1-регуляризованных признаков* обычно делается для получения высокоэффективных и поддающихся интерпретации человеком результатов. На рис. 1.12 показан результат применения классификационной модели с отбором разреженных признаков к задаче диагностики пациентов с синдромом дефицита внимания и гиперактивности (СДВГ — attention deficit hyperactivity disorder, ADHD). Разреженно распределенные участки цвета представляют собой активационные области, обнаруженные обучающимся алгоритмом, которые проводят значительное различие между людьми с СДВГ и без него.

11.11. Когда усвоение признаков не срабатывает

Предположив, что мы имплементируем инструменты, описанные в этой главе, правильно, когда перекрестная валидация, ансамблирование и (в более широком смысле) усвоение признаков не срабатывают? Простой ответ таков — усвоение признаков не срабатывает, когда наши данные не в достаточной мере отражают базовое явление, которое их сгенерировало. Инженерия нелинейных признаков (описанная в *главе 10*) также не срабатывает в таких обстоятельствах. Это может произойти, когда случается одно или несколько из следующих событий.

- **Когда набор данных не имеет внутренне присущей структуры**: если в данных присутствует мало связей или вообще их нет (вследствие ненадлежащего измерения, экспериментирования или отбора входных данных), нелинейная модель, усвоенная посредством усвоения признаков, будет бесполезной. Например, на рис. 11.53, *слева* представлен небольшой двухклассовый набор данных, сформированный путем случайного выбора точек на единичном квадрате и случайного назначения каждой точке одной из двух классовых меток. Никакая классификационная граница, усвоенная из этого набора данных, никогда не сможет произвести значение, т. к. сами данные не содержат значимого шаблона.
- **Когда набор данных слишком мал**, чтобы представить истинное опорное явление, усвоение признаков может непреднамеренно определить неправильную нелинейность. Например, на рис. 11.53, *в центре* приведен простой пример этого события. Явление, лежащее в основе этого двухклассового набора данных, имеет нелинейную границу (показано пунктирно линией черного цвета). Одна-

ко, поскольку мы отобрали слишком мало точек, данные, которые у нас есть, линейно разделимы, и перекрестная валидация восстановит линейную границу (показанную сплошной линией черного цвета), которая не отражает истинную природу лежащего в основе явления. Из-за малого размера данных эта проблема неизбежна.

- **Когда набор данных плохо распределен**: даже если набор данных велик, он все равно может не отражать истинную природу опорного явления, которое его сгенерировало. Когда это происходит, усвоение признаков не сработает. Например, на рис. 11.53, *справа* представлен простой двухклассовый набор данных, два класса которого разделены совершенно округлой границей, показанной пунктиром черного цвета. Хотя набор данных является большим, все образцы данных были взяты из верхней части входного пространства. Рассматриваемый сам по себе этот достаточно крупный набор данных не очень хорошо отражает истинное опорное явление. В то время как перекрестная валидация дает модель, которая совершенно разделяет два класса, соответствующая параболическая граница решения (показанная сплошной линией черного цвета) не совпадает с истинной округлой границей. Такое обстоятельство неизбежно возникает, когда данные плохо распределены и не отражают того явления, которое их сгенерировало.

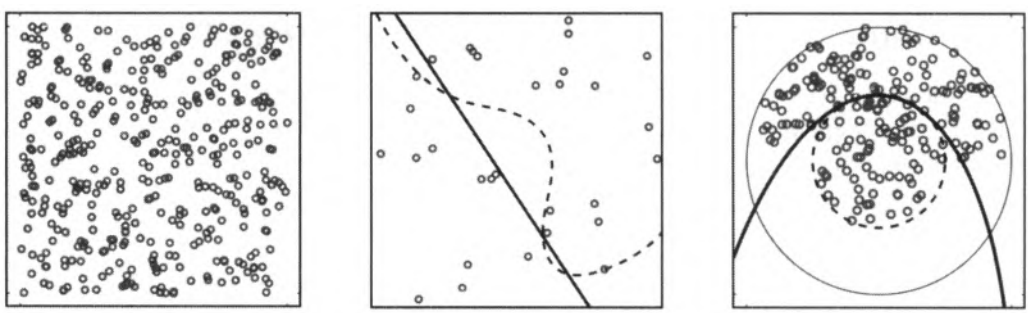


Рис. 11.53. Усвоение признаков не срабатывает, когда наши данные недостаточно отражают опорное явление, которое их сгенерировало. Это может произойти, когда набор данных плохо структурирован, слишком мал или плохо распределен, как данные, показанные соответственно на левой, средней и правой панелях

11.12. Заключение

В этой главе был описан ряд фундаментальных и чрезвычайно важных концепций из области *усвоения признаков*, или автоматического конструирования признаков, которые будут многократно повторяться до конца книги.

Самый главный из этих принципов — это понятие *универсальных аппроксиматоров*, введенное в *разд. 11.2*, которые являются аналогом охватывающих множеств (из векторной алгебры) для случая *совершенных данных*. Здесь мы познакомились с базовыми примерами трех фундаментальных семейств универсальных аппрокси-

маторов — *фиксированно-контурными*, *нейросетевыми* и *древесными*, каждое из которых обладает уникальными свойствами и техническими экстравагантностями, подробно рассматриваемыми в последующих главах. В отличие от случая совершенных данных, во время работы с *реальными данными* необходимо уделять большое внимание правильной настройке *емкости* модели, выстраиваемой на структурных единицах универсального аппроксиматора, и соответствующему *оптимизированию* ее параметров (*см. разд. 11.3*). Эти два "наборных диска" (как они описаны в *разд. 11.3.2*) представляют собой два главных элемента управления, которые мы имеем при правильном применении любой модели на основе универсального аппроксиматора к реальным данным. Действительно, усвоение признаков — это, по сути, надлежащая настройка емкостного и оптимизационного наборных дисков (автоматически) посредством методов *перекрестной валидации*, подробно описанных в *разд. 11.4–11.6*. Здесь мы увидели, что легче зафиксировать емкость на высоком уровне и тщательно оптимизировать, чем наоборот, что приводит к процедурам *бустинга* и *регуляризации*, описанным соответственно в *разд. 11.5* и *11.6*. Наконец, *бутстрап-агрегирование* — тщательное сочетание коллекции натренированных/перекрестно-валидированных моделей — было описано в *разд. 11.9* (наряду с аналогичной схемой *K*-блочной перекрестной валидации для более мелких и более выскоразмерных наборов данных в *разд. 11.10*), что обычно приводит к более высокорезультативным (бутстрап-агрегированным) моделям.

11.13. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:
github.com/jermwatt/machine_learning_refined.

11.1. Наивная перекрестная валидация I

Повторить эксперимент, описанный в примере 11.8, разбив исходный набор данных случайным образом на тренировочную и валидационную части. От вас не требуется воспроизводить рис. 11.27, но необходимо построить график, показывающий тренировочную и валидационную ошибки для вашего диапазона тестируемых моделей, и визуализировать найденную модель (вместе с данными), которая обеспечивает наименьшую валидационную ошибку. С учетом вашей конкретной тренировочно-валидационной разбивки ваши результаты могут отличаться от тех, что представлены в указанном примере.

11.2. Наивная перекрестная валидация II

Повторить эксперимент, описанный в примере 11.9, разбив исходный набор данных случайным образом на тренировочную и валидационную части. От вас не требуется воспроизводить рис. 11.28, но необходимо построить график, показывающий тренировочную и валидационную ошибки для вашего диапазона тестируемых моделей. С учетом вашей конкретной тренировочно-валидационной разбивки ваши результаты могут отличаться от тех, что представлены в указанном примере.

11.3. Перекрестная валидация на основе бустинга I

Повторить эксперимент, описанный в примере 11.11. От вас не требуется воспроизводить рис. 11.32, но необходимо построить график, показывающий тренировочную и валидационную ошибки для вашего диапазона тестируемых моделей.

11.4. Перекрестная валидация на основе бустинга II

Выполнить 20 раундов перекрестной валидации на основе бустинга с использованием нейросетевых единиц (определенных в уравнении (11.12)), задействовав набор данных по раку молочной железы, описанный в упражнении 9.5, и случайным образом разбив исходный набор данных на 80-процентную тренировочную и 20-процентную валидационную части.

11.5. Перекрестная валидация на основе регуляризации

Повторить эксперимент, описанный в примере 11.13. От вас не требуется воспроизводить рис. 11.40, но необходимо построить график, показывающий тренировочную и валидационную ошибки для вашего диапазона тестируемых моделей.

11.6. Бутстрап-агрегирование регрессионных моделей

Повторить первый эксперимент, описанный в примере 11.14, создав десять наивно перекрестно-валидированных многочленных моделей, обученных к разным тренировочно-валидационным разбивкам регрессионного набора данных, показанного на рис.11.44. Создать коллекцию графиков, подобных тем, что представлены на рис. 11.44, которые демонстрируют, как каждая отдельная модель обучена к данным, а также как обучена агрегированная медианная модель.

11.7. Бутстрап-агрегирование двухклассовых классификационных моделей

Повторить первый эксперимент, описанный в примере 11.15, создав пять наивно перекрестно-валидированных многочленных моделей, обученных к разным тренировочно-валидационным разбивкам двухклассового классификационного набора данных, показанного на рис. 11.46. Сравнить эффективность с точки зрения числа неправильных классифицирований над всем набором данных каждой отдельной модели и окончательной агрегированной модели.

11.8. Бутстрап-агрегирование многоклассовых классификационных моделей

Повторить второй эксперимент, описанный в примере 11.16, результаты которого показаны на рис. 11.48, *снизу*. Сравнить эффективность с точки зрения числа неправильных классифицирований над всем набором данных каждой отдельной модели и окончательной агрегированной модели.

11.9. *K*-блочная перекрестная валидация

Повторить эксперимент, описанный в примере 11.17, воспроизведя графики на рис. 11.50.

11.10. Классификация сахарного диабета

Выполнить K-блочную перекрестную валидацию с использованием линейной модели и ℓ_1-регуляризатора над популярным двухклассовым классификационным набором данных по геномике, состоящим из $P = 72$ точек данных, каждая из которых имеет входную размерность $N = 7128$. Это, как правило, будет создавать разреженную предсказательную линейную модель (как подробно описано в примере 11.18), которая полезна для определения малого числа генов, которые коррелируют с выходом этого двухклассового классификационного набора данных (т. е. независимо от того, имеет ли каждый индивидуум, представленный в наборе данных, диабет или нет).

12 Ядерные методы

12.1. Введение

В этой главе мы продолжим наше обсуждение фиксированно-контурных универсальных аппроксимаций, которое началось еще в *разд. 11.2.3*. Оно очень быстро приведет нас к понятию *ядра* как продуманного способа представления фиксированно-контурных признаков, благодаря которому они масштабируются изящнее при их применении к векторнозначным входным данным.

12.2. Фиксированно-контурные универсальные аппроксиматоры

Использовав классический многочлен в качестве образчика, в *разд. 11.2.3* мы ввели семейство универсальных фиксированно-контурных аппроксиматоров как коллекцию различных нелинейных функций, не имеющих внутренних (настраиваемых) параметров. В этом разделе мы продолжим обсуждение фиксированно-контурных аппроксиматоров, начав более глубокое погружение в технические аспекты, связанные с этими универсальными аппроксиматорами, и проблемы, которые мы должны решать при их задействовании на практике.

12.2.1. Тригонометрические универсальные аппроксиматоры

Что обычно характеризует фиксированно-контурные универсальные аппроксиматоры, так это отсутствие внутренних параметров и простая организация, причем очень часто структурные единицы определенного семейства фиксированно-контурных аппроксиматоров организуются в терминах степени или какого-либо другого естественного индекса (*см. разд. 11.2.3*). Эти простые характеристики сделали фиксированно-контурные аппроксиматоры, такие как многочлены (которые мы видели ранее), а также синусоидальные примеры и ряды Фурье (которые мы обсудим сейчас) чрезвычайно популярными в областях, смежных с машинным обучением, например в математике, физике и технике.

Пример 12.1. Синусоидальные аппроксиматоры

Коллекция синусоидальных волн возрастающей частоты является распространенным примером классического фиксированно-контурного аппроксиматора. При этом он имеет структурные единицы формы:

$$f_1(x) = \sin x, \quad f_2(x) = \sin(2x), \quad f_3(x) = \sin(3x) \text{ и т. д.,} \tag{12.1}$$

где m-й элемент в общем случае задается как $f_m(x) = \sin(mx)$. Первые четыре члена этого семейства функций показаны на рис. 12.1.

Обратите внимание, что, как и в случае с многочленами, каждый из этих элементов не имеет *настраиваемого* параметра внутри (и, следовательно, имеет *фиксированный* контур) и что элементы естественно упорядочены в терминах их сложности от низкой до высокой. Так же как и в случае с многочленами, мы можем легко обобщить этот каталог функций на более высокоразмерные входные данные. В общем случае для N-мерного входа синусоидальная единица принимает форму

$$f_m(\mathbf{x}) = \sin(m_1 x_1)\sin(m_2 x_2)\cdots\sin(m_N x_N), \tag{12.2}$$

где $m_1, m_2, ..., m_N$ — неотрицательные целые числа.

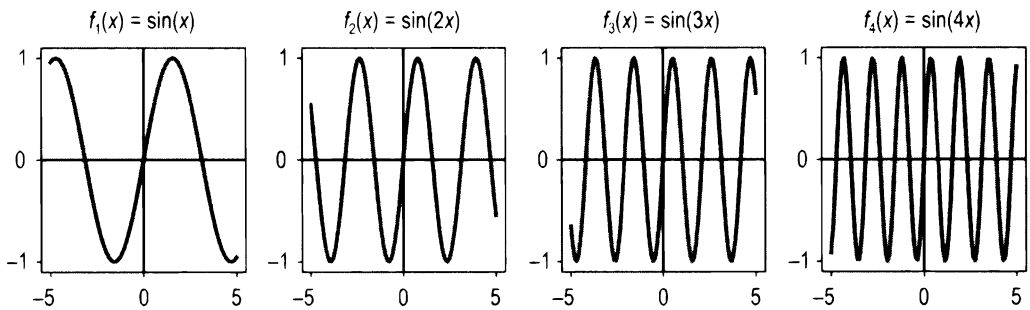


Рис. 12.1. Иллюстрация примера 12.1. Слева направо первые четыре единицы синусоидального универсального аппроксиматора

Пример 12.2. Базис Фурье

Аналогично синусоидальному семейству, описанному в примере 12.1, базис Фурье [56] — названный так в честь своего изобретателя Джозефа Фурье, который впервые использовал эти функции в начале 1800-х годов для изучения диффузии тепла, — состоит из набора парных синусоидальных и косинусоидальных волн (с постоянно возрастающей частотой) формы

$$f_{2m-1}(x) = \sin(2\pi m x) \text{ и } f_{2m}(x) = \cos(2\pi m x) \tag{12.3}$$

для всех $m \geq 1$. Также принято записывать единицы Фурье в компактной компле́ксной экспоненциальной форме (см. упражнение 12.1):

$$f_m(x) = e^{2\pi i m x}. \tag{12.4}$$

Для общего N-мерного входа каждая многомерная единица Фурье тогда принимает форму

$$f_m(\mathbf{x}) = e^{2\pi i m_1 x_1} e^{2\pi i m_2 x_2} \cdots e^{2\pi i m_N x_N}, \tag{12.5}$$

где $m_1, m_2, ..., m_N$ — целые числа.

12.2.2. Трудности масштабирования фиксированно-контурных аппроксиматоров с большой входной размерностью

Как мы видели в *главе 11*, при использовании многочленных единиц очень часто мы применяем полные пакеты одночленов в качестве многочлена некоторой степени. Например, многочленная модель степени D при двумерном входе состоит из всех одночленных единиц формы $f_m(\mathbf{x}) = x_1^p x_2^q$, где p и q — неотрицательные целые числа, такие что $0 < p + q \le D$.

В более общем случае при N-мерном входе многочленная единица принимает форму

$$f_m(\mathbf{x}) = x_1^{m_1} x_2^{m_2} \cdots x_N^{m_N}, \tag{12.6}$$

и для построения многочленной модели степени D мы собираем все такие члены, где $0 < m_1 + m_2 + ... + m_N \le D$ и $m_1, m_2, ..., m_N$ — неотрицательные целые числа. Если мы не используем многочленные единицы в тандеме с бустированием (*см. разд. 11.3*), то мы практически всегда применяем их как *полный пакет* единиц некоторой степени, а не по *отдельности*. Одна из причин тому в следующем: поскольку многочленные единицы упорядочены естественным образом (от низкой сложности к высокой), при включении в состав единицы конкретной сложности имеет смысл с организационной точки зрения включать все другие единицы в семействе, имеющие меньшую сложность. Например, обычно нет большого смысла определять квадратичные многочлены, свободные от линейных членов. Тем не менее упаковывание многочленных единиц таким образом — это не то, что мы *должны* делать при их задействовании, а разумная и общепринятая практика.

Подобно многочленам также принято при задействовании синусоидальных единиц и единиц Фурье использовать их в качестве полных пакетов, поскольку они тоже упорядочены в терминах их индивидуальных сложностей от низкой до высокой. Например, по аналогии с многочленом степени D мы можем упаковать модель Фурье степени D, состоящую из *всех единиц* формы, приведенной в уравнении (12.5), где $0 < \max(|m_1|, |m_2|, ..., |m_N|) \le D$. Этот выбор упаковывания в значительной степени является традиционным.

Однако очень серьезная практическая проблема возникает при задействовании фиксированно-контурных аппроксиматоров, таких как многочлены и тригонометрические базисы, когда они используются в полных пакетах единиц: даже при умеренном размере входной размерности N соответствующее число единиц в пакете M быстро растет вместе с N, вскоре становясь непомерно большим с точки зрения хранения и вычисления. Другими словами, число единиц типичного фиксированно-контурного аппроксиматора в модели, задействующей полный пакет таких единиц, *растет экспоненциально* вместе с размером входа.

Пример 12.3. Число единиц в многочленном аппроксиматоре степени D

Точное число M единиц в D-степенном многочлене входа с размерностью N можно вычислить точно:

$$M = \binom{N+D}{D} - 1 = \frac{(N+D)!}{N!D!} - 1. \quad (12.7)$$

Даже если входная размерность N имеет малый или умеренный размер, например $N = 100$ или $N = 1000$, то признаковая карта этих входных размерностей у ассоциированного многочлена степени $D = 5$ уже имеет соответственно $M = 96\,560\,645$ и $M = 8\,459\,043\,543\,950$ одночленов. В последнем случае невозможно удерживать признаковые векторы в памяти даже современного компьютера.

Пример 12.4. Число единиц в Фурье-аппроксиматоре степени *D*

Соответствующее число единиц M в пакете базисных Фурье-элементов степени D является еще более гигантским, чем у многочлена степени D: коллекция Фурье-признаков степени D произвольной входной размерности N имеет ровно

$$M = (2D+1)^N - 1 \quad (12.8)$$

единиц. При $D = 5$ и $N = 80$ оно составляет $11^{80} - 1$, число большее, чем текущее расчетное число атомов в видимой Вселенной!

Наш беглый анализ в примерах 12.3 и 12.4 показывает, что, поскольку суммарное число единиц фиксированно-контурного аппроксиматора растет *экспоненциально* во входной размерности, любой подход к отбору фиксированно-контурных единиц для нелинейной модели в целом является проблематичным. Например, в случае с многочленами, даже если бы мы выбрали меньшее множество только тех единиц, которые имеют точно такую же степень, т. е. все единицы, где $m_1 + m_2 + ... + m_N = D$, то в итоге мы все равно закончили бы комбинаторно большим числом задействуемых единиц.

Эта серьезная трудность масштабирования мотивирует существование так называемого *трюка с ядром*, описанного в следующем разделе, который расширяет использование классических фиксированно-контурных аппроксиматоров (при задействовании полных пакетов единиц) до задач с высокоразмерным входом.

12.3. Трюк с ядром

Указанная краеугольная проблема, связанная с невозможностью эффективного хранения и вычисления с помощью преобразований высокоразмерных фиксированно-контурных признаков, мотивирует искать более эффективные представления. В этом разделе мы введем понятие *ядрирования*[1], также обычно именуемое *трюком с ядром*, как продуманный способ построения фиксированно-контурных признаков практически для любой задачи машинного обучения. Ядрирование не

[1] Ядрирование (kernelization), или параметрическая редукция, — это метод разработки эффективных алгоритмов, которые достигают своей эффективности на шаге предобработки, на котором входные данные алгоритма заменяются меньшими входными данными, именуемыми "ядром". — *Прим. пер.*

только позволяет нам избежать проблемы комбинаторного взрыва, подробно описанной в конце предыдущего раздела, но и обеспечивает способ генерирования новых фиксированно-контурных признаков, определяемых исключительно посредством такого ядрированного представления.

12.3.1. Полезный факт из фундаментальной теоремы линейной алгебры

Прежде чем обсуждать понятие ядрирования, не лишним будет сначала вспомнить полезное положение из фундаментальной теоремы линейной алгебры о разложении любого M-мерного вектора $\boldsymbol{\omega}$ по столбцам заданной $M \times P$-матрицы $\mathbf{F}$. Обозначив p-й столбец матрицы $\mathbf{F}$ через $\mathbf{f}_p$, в том случае, когда $\boldsymbol{\omega}$ оказывается лежащим *внутри* столбцового пространства матрицы $\mathbf{F}$, мы можем выразить его через линейную комбинацию этих столбцов как

$$\boldsymbol{\omega} = \sum_{p=1}^{P} \mathbf{f}_p z_p , \tag{12.9}$$

где z_p — линейно-комбинационный вес или коэффициент, связанный с $\mathbf{f}_p$. Уложив эти веса в столбцовый $P \times 1$-вектор $\mathbf{z}$, мы можем записать это соотношение компактнее:

$$\boldsymbol{\omega} = \mathbf{F}\mathbf{z} . \tag{12.10}$$

С другой стороны, если $\boldsymbol{\omega}$ лежит *вне* столбцового пространства матрицы $\mathbf{F}$, как показано на рис. 12.2, то мы можем разложить его на две части — часть $\boldsymbol{\omega}$, принадлежащую подпространству, охватываемому столбцами матрицы $\mathbf{F}$, и ортогональную компоненту $\mathbf{r}$ — и записать его как

$$\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r} . \tag{12.11}$$

Обратите внимание, что $\mathbf{r}$, будучи ортогональным к интервалу столбцов в $\mathbf{F}$, алгебраически означает, что $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Более того, когда $\boldsymbol{\omega}$ находится в столбцовом пространстве $\mathbf{F}$, мы по-прежнему можем разложить его, используя более общую форму, приведенную в уравнении (12.11), установив $\mathbf{r} = \mathbf{0}_{M\times 1}$ без нарушения условия ортогональности $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$.

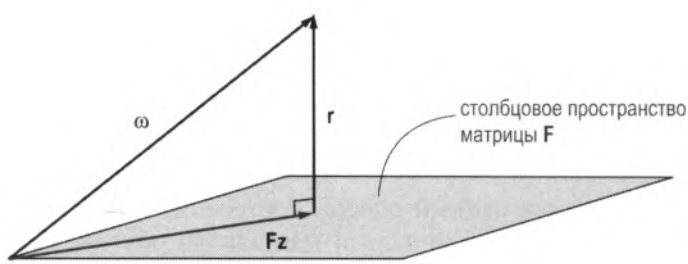


Рис. 12.2. Иллюстрация полезного факта из фундаментальной теоремы линейной алгебры, которая утверждает, что любой вектор $\boldsymbol{\omega}$ в M-мерном пространстве может быть разложен как $\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r}$, где вектор $\mathbf{F}\mathbf{z}$ принадлежит столбцовому пространству матрицы $\mathbf{F}$, а $\mathbf{r}$ ортогонален этому подпространству

В итоге, любой вектор $\boldsymbol{\omega}$ в M-мерном пространстве может быть разложен на столбцовое пространство заданной матрицы $\mathbf{F}$ как $\boldsymbol{\omega} = \mathbf{Fz} + \mathbf{r}$. Вектор $\mathbf{Fz}$ принадлежит подпространству, определяемому столбцами матрицы $\mathbf{F}$, а $\mathbf{r}$ ортогонален этому подпространству. Как мы теперь увидим, это простое разложение является ключом к более эффективному представлению фиксированно-контурных признаков.

12.3.2. Ядрирование стоимостных функций машинного обучения

Приведем несколько фундаментальных примеров того, как ядрируются стандартные задачи машинного обучения с учителем и их стоимостные функции, включая стоимость наименьших квадратов для регрессии и стоимость мягкого максимума для двухклассовой классификации. Практически все стоимостные функции машинного обучения можно ядрировать, следуя аргументам, подобным этим, включая многоклассовый мягкий максимум, анализ главных компонент и кластеризацию на основе K средних (см. упражнения главы).

Пример 12.5. Ядрирование регрессии посредством стоимости наименьших квадратов

Предположим, что мы хотим выполнить обобщенную нелинейную регрессию, используя наши M структурных единиц, принадлежащих фиксированно-контурному аппроксиматору степени D, причем наша соответствующая модель, вычисляемая в p-м входе $\mathbf{x}_p$, принимает форму

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = w_0 + f_1\left(\mathbf{x}_p\right)w_1 + f_2\left(\mathbf{x}_p\right)w_2 + \ldots + f_M\left(\mathbf{x}_p\right)w_M . \tag{12.12}$$

Для удобства запишем это компактнее, выставляя признаковые веса и смещение отдельно:

$$\text{model}\left(\mathbf{x}_p, b, \boldsymbol{\omega}\right) = b + \mathbf{f}_p^T\boldsymbol{\omega} , \tag{12.13}$$

где использована нотация смещения/признаковых весов (ранее введенная, например, в *разд. 6.4.5*):

$$b = w_0 \text{ и } \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix}, \tag{12.14}$$

а также сокращенное обозначение для всего нашего множества из M признаковых преобразований тренировочного входа $\mathbf{x}_p$:

$$\mathbf{f}_p = \begin{bmatrix} f_1\left(\mathbf{x}_p\right) \\ f_2\left(\mathbf{x}_p\right) \\ \vdots \\ f_M\left(\mathbf{x}_p\right) \end{bmatrix}. \tag{12.15}$$

В этой нотации наша стоимость наименьших квадратов для регрессии принимает форму

$$g(b, \boldsymbol{\omega}) = \frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\boldsymbol{\omega} - y_p\right)^2. \tag{12.16}$$

Теперь обозначим через $\mathbf{F}$ матрицу размера $M \times P$, образованную путем укладки векторов $\mathbf{f}_p$ по столбцам. Задействовав фундаментальную теорему линейной алгебры, рассмотренную в предыдущем разделе, мы можем записать $\boldsymbol{\omega}$ как

$$\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r}, \tag{12.17}$$

где $\mathbf{r}$ удовлетворяет $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Подставив это представление $\boldsymbol{\omega}$ обратно в стоимостную функцию в уравнении (12.16), получаем

$$\frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T(\mathbf{F}\mathbf{z} + \mathbf{r}) - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\mathbf{F}\mathbf{z} - y_p\right)^2. \tag{12.18}$$

Наконец, обозначив симметричную $P \times P$-матрицу $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ (и ее p-й столбец $\mathbf{h}_p = \mathbf{F}^T\mathbf{f}_p$), именуемую *ядерной матрицей*, или просто *ядром* для краткости, можно выразить нашу исходную стоимостную функцию эквивалентным образом

$$g(d, \mathbf{z}) = \frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{h}_p^T\mathbf{z} - y_p\right)^2. \tag{12.19}$$

При этом наша соответствующая модель, вычисляемая на p-м входе, теперь принимает эквивалентную форму

$$\text{model}\left(\mathbf{x}_p, b, \mathbf{z}\right) = b + \mathbf{h}_p^T\mathbf{z}. \tag{12.20}$$

Обратите внимание, что при ядрировании исходной регрессионной модели в уравнении (12.13) и связанной с ней стоимостной функции в уравнении (12.16) мы изменили их аргументы (ввиду подстановки $\boldsymbol{\omega}$), придя к полностью эквивалентной *ядрированной* модели в уравнении (12.20) и *ядрированной* стоимостной функции в уравнении (12.19).

Пример 12.6. Ядрирование двухклассовой классификации посредством стоимости мягкого максимума

Следуя шаблону, показанному в примере 12.5, здесь мы по существу повторяем тот же аргумент, задействовав стоимость двухклассового мягкого максимума.

Запишем нашу обобщенную стоимость двухклассового мягкого максимума с привлечением той же нотации, что и в примере 12.5:

$$g(b, \boldsymbol{\omega}) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\boldsymbol{\omega}\right)}\right). \tag{12.21}$$

Затем запишем представление $\boldsymbol{\omega}$ над $\mathbf{F}$ как $\boldsymbol{\omega} = \mathbf{F}\mathbf{z} + \mathbf{r}$, где $\mathbf{F}^T\mathbf{r} = \mathbf{0}_{P\times 1}$. Подставив это в уравнение (12.21) и упростив, получим

$$g(b, \mathbf{z}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T \mathbf{F}\mathbf{z}\right)}\right). \tag{12.22}$$

Обозначив ядерную $P \times P$-матрицу $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ (где $\mathbf{h}_p = \mathbf{F}^T\mathbf{f}_p$ — это p-й столбец матрицы $\mathbf{H}$), мы можем записать стоимостную функцию из уравнения (12.22) в *ядрированной* форме как:

$$g(b, \mathbf{z}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1 + e^{-y_p\left(b + \mathbf{h}_p^T \mathbf{z}\right)}\right). \tag{12.23}$$

Эта ядрированнаяя форма двухклассового мягкого максимума часто называется *ядрированной логистической регрессией.*

Используя аргумент того же рода, что и в примерах 12.5 и 12.6, мы можем решить практически любую задачу машинного обучения, обсуждаемую в этой книге, включая многоклассовую классификацию, анализ главных компонент, кластеризацию методом K средних, а также любую ℓ_2-регуляризованную версию этих моделей. Для удобства использования покажем как исходные, так и ядрированные формы популярных стоимостных функций обучения с учителем в табл. 12.1.

Таблица 12.1. Популярные стоимостные функции обучения с учителем и их ядрированные версии

Стоимостная функция	Оригинальная версия	Ядрированная версия
Наименьшие квадраты	$\frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{f}_p^T\boldsymbol{\omega} - y_p\right)^2$	$\frac{1}{P}\sum_{p=1}^{P}\left(b + \mathbf{h}_p^T\mathbf{z} - y_p\right)^2$
Двухклассовый мягкий максимум	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{f}_p^T\boldsymbol{\omega}\right)}\right)$	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)}\right)$
SVM с квадратичной маржой	$\frac{1}{P}\sum_{p=1}^{P}\max^2\left(0,\ 1 - y_p\left(b + \mathbf{f}_p^T\boldsymbol{\omega}\right)\right)$	$\frac{1}{P}\sum_{p=1}^{P}\max^2\left(0,\ 1 - y_p\left(b + \mathbf{h}_p^T\mathbf{z}\right)\right)$
Многоклассовый мягкий максимум	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1 + \sum_{\substack{j=0 \\ j \neq y_p}}^{C-1} e^{\left(b_j - b_{y_p}\right)\mathbf{f}_p^T\left(\boldsymbol{\omega}_j - \boldsymbol{\omega}_{y_p}\right)}\right)$	$\frac{1}{P}\sum_{p=1}^{P}\log\left(1 + \sum_{\substack{j=0 \\ j \neq y_p}}^{C-1} e^{\left(b_j - b_{y_p}\right)\mathbf{h}_p^T\left(\mathbf{z}_j - \mathbf{z}_{y_p}\right)}\right)$
ℓ_2-регуляризатор[2]	$\lambda\lVert\boldsymbol{\omega}\rVert_2^2$	$\lambda\mathbf{z}^T\mathbf{H}\mathbf{z}$

[2] ℓ_2-регуляризатор может быть добавлен в любую стоимостную функцию $g(b, \boldsymbol{\omega})$ в средней графе, и результирующая ядрированная форма суммы $g(b, \boldsymbol{\omega}) + \lambda\lVert\boldsymbol{\omega}\rVert_2^2$ будет суммой ядрированной стоимости и ядрированного регуляризатора, т. е. $g(b, \mathbf{z}) + \lambda\mathbf{z}^T\mathbf{H}\mathbf{z}$.

12.3.3. Популярные ядра в машинном обучении

Реальная ценность ядрирования любой стоимости машинного обучения состоит в том, что для многих фиксированно-контурных структурных единиц, включая многочленные признаки и Фурье-признаки, ядерная матрица $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ может быть построена *без* предварительного строительства матрицы $\mathbf{F}$ (которая часто имеет непомерно большой размер строк). Вместо этого, как мы увидим на ряде примеров, эта матрица может быть построена *поэлементно* с помощью простых формул. Более того, если думать о построении ядерных матриц таким образом, то это приводит к построению фиксированно-контурных универсальных аппроксиматоров, начиная с определения самой ядерной матрицы (а не с явного преобразования признаков). В любом случае, построив ядерную матрицу без предварительного вычисления $\mathbf{F}$, мы полностью избегаем проблемы экспоненциального масштабирования с фиксированно-контурными универсальными аппроксиматорами, обсуждавшейся в *разд. 12.2.2*.

Пример 12.7. Многочленное ядро

Рассмотрим следующее ниже многочленное отображение степени $D = 2$ из $N = 2$ в пятимерное ($M = 5$) пространство, заданное вектором признакового преобразования

$$\mathbf{f} = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_1x_2 \\ x_2^2 \end{bmatrix}. \tag{12.24}$$

Обратите внимание, что умножение некоторых или всех элементов в $\mathbf{f}$ на постоянное значение, например $\sqrt{2}$, как в

$$\mathbf{f} = \begin{bmatrix} \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}, \tag{12.25}$$

не изменяет это признаковое преобразование для наших целей моделирования, т. к. $\sqrt{2}$, присоединенные к нескольким членам, могут быть поглощены связанными с ними весами в $\boldsymbol{\omega}$ при формировании модели $\text{model}(\mathbf{x}, b, \boldsymbol{\omega}) = b + \mathbf{f}^T\boldsymbol{\omega}$. Обозначив кратко через $\mathbf{u} = \mathbf{x}_i$ и $\mathbf{v} = \mathbf{x}_j$ соответственно i-ю и j-ю входные точки данных, (i, j)-й элемент ядерной матрицы $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ для многочлена степени $D = 2$ запишем как

$$h_{i,j} = \mathbf{f}_i^T \mathbf{f}_j =$$

$$= \begin{bmatrix} \sqrt{2}u_1 & \sqrt{2}u_2 & u_1^2 & \sqrt{2}u_1u_2 & u_2^2 \end{bmatrix} \begin{bmatrix} \sqrt{2}v_1 \\ \sqrt{2}v_2 \\ v_1^2 \\ \sqrt{2}v_1v_2 \\ v_2^2 \end{bmatrix} =$$

$$= \left(1 + 2u_1v_1 + 2u_2v_2 + u_1^2v_1^2 + 2u_1u_2v_1v_2 + u_2^2v_2^2\right) - 1 =$$

$$= \left(1 + u_1v_1 + u_2v_2\right)^2 - 1 = \left(1 + \mathbf{u}^T\mathbf{v}\right)^2 - 1. \qquad (12.26)$$

Другими словами, ядерную матрицу **H** можно построить без предварительного конструирования явных признаков в уравнении (12.25), просто определив ее поэлементно как

$$h_{i,j} = \left(1 + \mathbf{x}_i^T\mathbf{x}_j\right)^2 - 1. \qquad (12.27)$$

Этот способ определения матрицы многочленного ядра очень полезен тем, что нам требуется доступ только к исходным входным данным, а не к самим явным многочленным признакам.

Хотя правило построения ядра в уравнении (12.27) было выведено специально для $N = 2$ и многочлена степени $D = 2$, можно показать, что многочленное ядро может быть определено поэлементно, аналогичным образом для общих N и D как

$$h_{i,j} = \left(1 + \mathbf{x}_i^T\mathbf{x}_j\right)^D - 1. \qquad (12.28)$$

Пример 12.8. Ядро Фурье

Признаковое Фурье-преобразование степени D из одномерного ($N = 1$) в двумерное (M = 2) пространство может быть записано как признаковый $2D \times 1$-вектор

$$\mathbf{f} = \begin{bmatrix} \sqrt{2}\cos(2\pi x) \\ \sqrt{2}\sin(2\pi x) \\ \vdots \\ \sqrt{2}\cos(2\pi Dx) \\ \sqrt{2}\sin(2\pi Dx) \end{bmatrix}, \qquad (12.29)$$

где, как было объяснено в предыдущем примере, умножение его элементов на $\sqrt{2}$ не изменяет исходное преобразование, определенное в уравнении (12.3) для наших целей моделирования. В этом случае соответствующий (i, j)-й элемент ядерной матрицы $\mathbf{H} = \mathbf{F}^T\mathbf{F}$ может быть записан как

$$h_{i,j} = \mathbf{f}_i^T\mathbf{f}_j = \sum_{m=1}^{D} 2\left[\cos\left(2\pi m x_i\right)\cos\left(2\pi m x_j\right) + \sin\left(2\pi m x_i\right)\sin\left(2\pi m x_j\right)\right]. \qquad (12.30)$$

Задействовав простое тригонометрическое тождество $\cos\alpha\cos\beta+\sin\alpha\sin\beta=$ $=\cos(\alpha-\beta)$, это можно записать эквивалентным образом как

$$h_{i,j}=\sum_{m=1}^{D}2\cos\left(2\pi m\left(x_i-x_j\right)\right). \tag{12.31}$$

Задействовав комплексное определение косинуса, т. е. $\cos\alpha=\left(e^{i\alpha}+e^{-i\alpha}\right)/2$, мы можем переписать это как

$$h_{i,j}=\sum_{m=1}^{D}\left[e^{2\pi im\left(x_i-x_j\right)}+e^{-2\pi im\left(x_i-x_j\right)}\right]=\left[\sum_{m=-D}^{D}e^{2\pi im\left(x_i-x_j\right)}\right]-1. \tag{12.32}$$

Если x_i-x_j — это целое число, то $e^{2\pi im\left(x_i-x_j\right)}=1$, а выражение внутри скобок в уравнении (12.32) составляет в сумме $2D+1$. Предположив, что это не так, и рассматривая только суммирование, мы можем написать

$$\sum_{m=-D}^{D}e^{2\pi im\left(x_i-x_j\right)}=e^{-2\pi iD\left(x_i-x_j\right)}\sum_{m=0}^{2D}e^{2\pi im\left(x_i-x_j\right)}. \tag{12.33}$$

Обратив внимание, что сумма в правой части является геометрической прогрессией, мы можем еще больше упростить вышеизложенное как

$$e^{2\pi iD\left(x_i-x_j\right)}\frac{1-e^{2\pi i\left(x_i-x_j\right)(2D+1)}}{1-e^{2\pi i\left(x_i-x_j\right)}}=\frac{\sin\left((2D+1)\pi\left(x_i-x_j\right)\right)}{\sin\left(\pi\left(x_i-x_j\right)\right)}, \tag{12.34}$$

где окончательное равенство следует из комплексного определения синуса, т. е. $\sin\alpha=\left(e^{i\alpha}-e^{-i\alpha}\right)/2$.

Поскольку в пределе, когда t приближается к любому целочисленному значению, мы имеем $\sin\left((2D+1)\pi t\right)/\sin\left(\pi t\right)=2D+1$ (что можно показать, используя правило Лопиталя из базового исчисления), то в заключение мы можем в целом написать, что

$$h_{i,j}=\frac{\sin\left((2D+1)\pi\left(x_i-x_j\right)\right)}{\sin\left(\pi\left(x_i-x_j\right)\right)}-1. \tag{12.35}$$

Похожую процедуру выведения ядра Фурье можно выполнить для общего N-мерного входа (см. упражнение 12.11).

Пример 12.9. Ядро радиальной базисной функции

Еще одним популярным вариантом ядра является ядро радиальной базисной функции (Radial Basis Function, RBF), определяемое поэлементно над входными данными как:

$$h_{i,j}=e^{-\beta\left\|\mathbf{x}_i-\mathbf{x}_j\right\|_2^2}, \tag{12.36}$$

где $\beta > 0$ — *гиперпараметр*, который должен быть настроен к данным. Хотя ядро RBF в типичной ситуации определяется непосредственно как ядерная матрица в уравнении (12.36), его можно проследить обратно до явного признакового преобразования, как и в случае многочленных ядер и ядер Фурье. То есть мы можем отыскать явную форму такого фиксированно-контурного признакового преобразования **f**, что

$$h_{i,j} = \mathbf{f}_i^T \mathbf{f}_j, \tag{12.37}$$

где $\mathbf{f}_i$ и $\mathbf{f}_j$ — признаковые преобразования соответственно входных точек $\mathbf{x}_i$ и $\mathbf{x}_j$.

Признаковое RBF-преобразование отличается от многочленных преобразований и Фурье-преобразований тем, что связанный с ним признаковый вектор **f** является *бесконечномерным*. Например, при $N = 1$ признаковый вектор **f** принимает форму

$$\mathbf{f} = \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \\ \vdots \end{bmatrix}, \tag{12.38}$$

где m-й элемент (или признак) определяется как

$$f_m(x) = e^{-\beta x^2} \sqrt{\frac{(2\beta)^{m-1}}{(m-1)!}} x^{m-1} \text{ для всех } m \geq 1. \tag{12.39}$$

При $N > 1$ соответствующий признаковый вектор принимает аналогичную форму, которая является также бесконечной по длине, что делает невозможным даже построение и хранение такого признакового вектора (независимо от входной размерности).

Обратите внимание, что контур (и, следовательно, подогнанное поведение) ядер RBF зависит от установки их гиперпараметра β. В общем случае, чем больше задан β, тем сложнее становится ассоциированная модель, задействующая ядро RBF. Для того чтобы проиллюстрировать это, на рис. 12.3 показаны три примера обучения с учителем: регрессия (верхний ряд), двухклассовая классификация (средний ряд) и многоклассовая классификация (нижний ряд), — с помощью ядра RBF с тремя отличающимися настройками β в каждом случае. Это создает недоподогнанное поведение (*слева*), разумное предсказательное поведение (*в центре*) и переподогнанное поведение (*справа*). В каждом случае метод Ньютона использовался для минимизирования каждой соответствующей стоимости и, следовательно, настройки параметров каждой модели. На практике β устанавливается посредством *перекрестной валидации* (см. пример 12.10).

Хотя мы представили здесь несколько ядер, наиболее часто используемых на практике, читатель может ознакомиться, например, с более исчерпывающим перечнем ядер и их свойств в [57, 58].

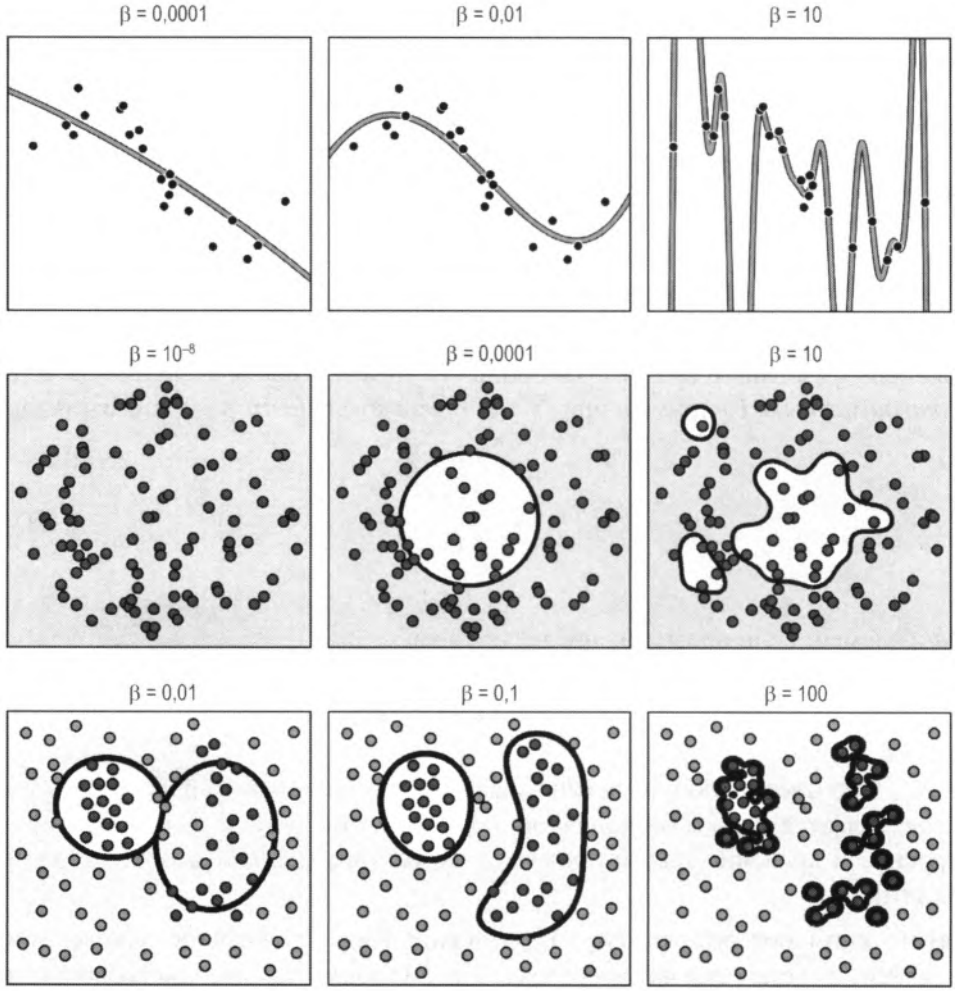


Рис. 12.3. Иллюстрация примера 12.9

12.3.4. Выполнение предсказаний с помощью ядрированных моделей

Как мы видели в примерах 12.5 и 12.6, ядрированная форма общей контролируемой модели, вычисляемой в точке **x**, принимает форму

$$\text{model}(\mathbf{x}, b, \mathbf{z}) = b + \mathbf{h}^T \mathbf{z}, \tag{12.40}$$

где параметры b и **z** должны быть настроены путем минимизирования соответствующей ядрированной стоимости. В этом каркасе ядризация **h** обобщенного входа **x** предусматривает вычисление относительно *каждой* точки $\mathbf{x}_p$ в (тренировочном) наборе данных. Например, при ядре D-степенного многочлена **h** задается как P-мерный вектор:

$$\mathbf{h} = \begin{bmatrix} \left(1 + \mathbf{x}_1^T \mathbf{x}\right)^D - 1 \\ \left(1 + \mathbf{x}_2^T \mathbf{x}\right)^D - 1 \\ \vdots \\ \left(1 + \mathbf{x}_P^T \mathbf{x}\right)^D - 1 \end{bmatrix}. \tag{12.41}$$

Эта необходимость задействования каждой (тренировочной) точки данных при вычислении натренированной модели является практически уникальной[3] для ядрированных учеников, т. к. мы не увидим этого требования при задействовании других универсальных аппроксиматоров в последующих главах.

12.4. Ядра как меры сходства

Если мы оглянемся назад на форму многочленного ядра, ядер Фурье и RBF в примерах 12.7–12.9, то увидим, что в каждом случае (i, j)-й элемент ядерной матрицы является функцией, определенной на паре $(\mathbf{x}_i, \mathbf{x}_j)$ входных данных. Например, изучая ядро RBF

$$h_{i,j} = e^{-\beta \|\mathbf{x}_i - \mathbf{x}_j\|_2^2}, \tag{12.42}$$

мы видим, что в качестве функции от $\mathbf{x}_i$ и $\mathbf{x}_j$ оно измеряет сходство между этими двумя входами посредством ℓ_2-нормы их разности. Чем больше похожи $\mathbf{x}_i$ и $\mathbf{x}_j$ во входном пространстве, тем больше становится $h_{i,j}$, и наоборот. Другими словами, ядро RBF можно интерпретировать как *меру сходства*, которая описывает, насколько близко два входа похожи друг на друга. Эта интерпретация ядер как мер сходства применима и к другим ранее введенным ядрам, включая полиномиальные ядра и ядра Фурье, хотя эти ядра ясно кодируют сходство разными способами.

На рис. 12.4 визуализированы наши три образцовых ядра (полиномиальное, Фурье и RBF) как меры сходства с фиксированием $\mathbf{x}_i$ в $\mathbf{x}_j = [0{,}5 \quad 0{,}5]^T$ и построением на графике $h_{i,j}$ для мелкозернистого диапазона значений $\mathbf{x}_j$ над единичным квадратом $[0 \quad 1]^2$, что в результате дает цветокодированную поверхность, показывающую, как каждое ядро трактует точки вблизи $\mathbf{x}_i$. Проанализировав этот рисунок, мы можем получить общее представление о том, как эти три ядра определяют *сходство* между точками. Во-первых, мы видим, что многочленное ядро трактует точки данных $\mathbf{x}_i$ и $\mathbf{x}_j$ схожим образом, если их внутреннее произведение является высоким или, другими словами, если они сильно коррелируют друг с другом. Похожим образом точки трактуются как несходные, когда они являются ортогональными друг другу. С другой стороны, ядро Фурье трактует точки как сходные, если они

[3] Вычисление классификатора методом K ближайших соседей также предусматривает задействование всего тренировочного набора.

лежат близко друг к другу, но их сходство отличается, как функция sinc, по мере увеличения их расстояния друг от друга. Наконец, ядро RBF обеспечивает гладкое сходство между точками: если они находятся близко друг к другу в *евклидовом* смысле, то они считаются очень похожими, но как только расстояние между ними проходит определенный порог, они быстро становятся несхожими.

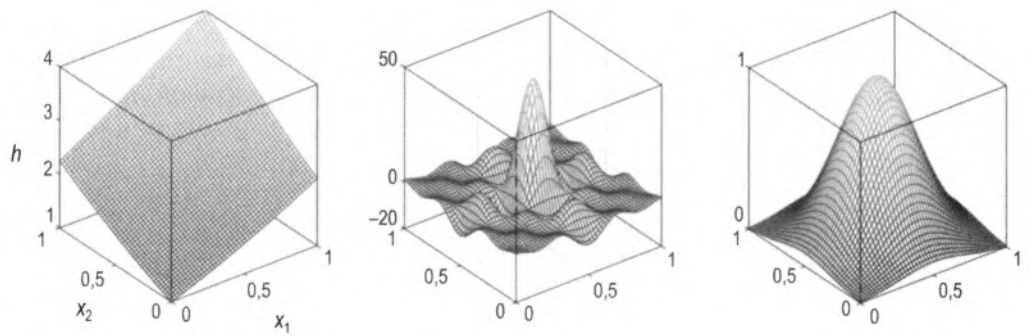


Рис. 12.4. Поверхности, сгенерированные полиномиальным ядром, ядрами Фурье и RBF, с центром в точке $\mathbf{x}_i = [0,5 \quad 0,5]^T$. Каждая точка поверхности имеет цветовую кодировку, основанную на ее магнитуде, которая может рассматриваться как мера сходства между $\mathbf{x}_i$ и соответствующим ей входом. Многочленное ядро степени $D = 2$ (*слева*), ядро Фурье степени $D = 3$ (*в центре*) и ядро RBF с $\beta = 10$ (*справа*)

12.5. Оптимизация ядрированных моделей

Как обсуждалось ранее, практически любая модель машинного обучения (с учителем или без учителя) может быть ядром. Реальная ценность в ядрировании состоит в том, что для большого диапазона типов ядер мы можем фактически построить ядерную матрицу **H** *без* явного определения связанных с ней признаковых преобразований. Как мы уже видели, это позволяет нам обойти проблему масштабирования, связанную с фиксированно-контурными аппроксиматорами с крупной входной размерностью (*см. разд. 12.2.2*). Более того, поскольку окончательная ядрированная модель остается линейной по своим параметрам, соответствующие ядрированные стоимостные функции вполне "хороши" с точки зрения их общей геометрии. Например, любая выпуклая стоимостная функция для регрессии и классификации *остается неизменной* при ядрировании, включая популярные стоимостные функции для регрессии, двухклассовой и многоклассовой классификации (подробно описанные в *главах 5–7*). Это позволяет использовать практически любой метод оптимизации для настройки ядрированного контролируемого ученика, с подходов нулевого порядка до подходов первого порядка и даже мощных подходов второго порядка, таких как метод Ньютона (подробно описанный в *главах 2–4*).

Однако, поскольку обобщенная ядерная матрица **H** является квадратной матрицей размера $P \times P$ (где P — это число точек данных в тренировочном наборе), ядрированные модели по своей сути масштабируются квадратично (и поэтому очень пло-

хо) по размеру тренировочных данных. Это не только чрезвычайно затрудняет тренировку ядрированных моделей на крупных наборах данных, но и предсказания с использованием таких моделей (которые, как мы видели в *разд. 12.3.4*, требуют вычисления *каждой* точки тренировочных данных) становятся все более трудными по мере увеличения размера тренировочных данных.

Большинство стандартных способов решения этой сложной проблемы масштабирования по размеру тренировочных данных вращаются вокруг того, чтобы избежать создания всей ядерной матрицы **H** сразу, в особенности во время тренировки. Например, можно использовать первопорядковые методы, такие как стохастический градиентный спуск, благодаря которому одновременно обрабатывается только малое число тренировочных точек данных, т. е. во время тренировки одновременно создается только малое подмножество столбцов матрицы **H**. Иногда может использоваться и структура некоторых задач с целью предотвращения явного построения ядра.

12.6. Перекрестная валидация ядрированных учеников

В общем случае существует большая разница между *емкостью* последующих степеней D и $D+1$ в моделях, задействующих многочленное ядро и ядро Фурье. Например, в случае многочленов разница между числом единиц, инкапсулированных в многочленном ядре степени D, и числом единиц в многочленном ядре степени $(D+1)$ может быть вычислена с помощью уравнения (12.7) как

$$\left(\begin{bmatrix} N+D+1 \\ D+1 \end{bmatrix}-1\right)-\left(\begin{bmatrix} N+D \\ D \end{bmatrix}-1\right)=\begin{bmatrix} N+D \\ D+1 \end{bmatrix}. \tag{12.43}$$

Например, при $N=500$ в ядерной матрице степени $D=3$ содержится на 20 958 500 многочленных единиц больше, чем в ядерной матрице степени $D=2$. Из-за этого огромного комбинаторного скачка в емкости между ядрами последующей степени перекрестная валидация посредством *регуляризации* с ℓ_2-нормой (как подробно описано в *разд. 11.4*) является обычной практикой при задействовании многочленного ядра и ядра Фурье. Поскольку гиперпараметр β ядра RBF непрерывен, модели, задействующие ядро RBF, могут (в дополнение к регуляризационному подходу) быть перекрестно валидированы в принципе путем прямого сравнения различных значений β.

Пример 12.10. Классификация рака молочной железы с использованием ядра RBF

В этом примере мы используем наивную перекрестную валидацию (впервые подробно описанную в *разд. 11.4.2*) для определения идеального параметра β для ядра RBF над набором данных по раку молочной железы, впервые описанному в упражнении 6.13. В этом наборе экспериментов мы используем стоимость мягкого максимума и откладываем всторону 20% этого двухклассового набора данных (случайно) для целей валидации (причем одинаковая часть откладывается в сторону для

валидации по каждому используемому значению β). Мы испытываем диапазон из 50 равноотстоящих значений для β на интервале [0; 1], минимизируем соответствующую стоимость с помощью метода Ньютона и строим график числа неправильных классифицирований как для тренировочного (синим цветом), так и для валидационного (коричневым цветом) наборов на рис. 12.5. Минимальное число неправильных классифицирований в валидационном наборе произошло, когда β было установлено близко к значению 0,2, что привело к одной и пяти неправильным классифицированиям соответственно на тренировочном и валидационном наборах. Простой линейный классификатор, натренированный на одной и той же части данных, обеспечил 7 и 22 неправильных классифицирований соответственно на тренировочном и валидационном наборах.

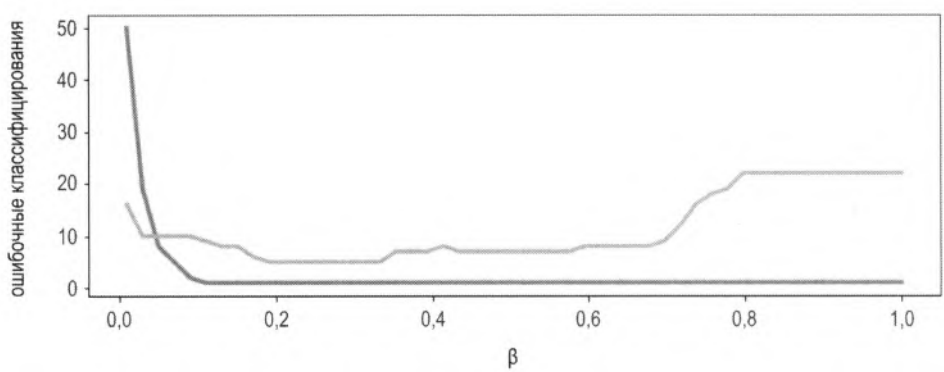


Рис. 12.5. Иллюстрация примера 12.10

12.7. Заключение

В этой главе мы продолжили описание фиксированно-контурных моделей, начиная с *разд. 11.2.3*, где они были впервые представлены. Мы начали с обзора нескольких популярных примеров фиксированно-контурных универсальных аппроксиматоров в *разд. 12.2*. Тот факт, что эти универсальные аппроксиматоры чрезвычайно плохо масштабируются вместе с входной размерностью набора данных, как описано в *разд. 12.2.2*, естественно, привел нас к обсуждению их расширения в виде *ядер* в *разд. 12.3*. Используя "трюк с ядром", мы можем не только расширить популярные фиксированно-контурные аппроксиматоры, чтобы легче справляться с высокоразмерным входом, но и создавать ряд новых аппроксиматоров непосредственно в виде ядер. Однако, в то время как ядрирование фиксированно-контурного аппроксиматора помогает ему преодолеть проблемы масштабирования по входной размерности набора данных, оно вводит проблему масштабирования по размеру *набора данных*. Эта проблема может быть несколько смягчена посредством продуманного построения и оптимизации ядерной матрицы (как описано в *разд. 12.5*). Наконец, в *разд. 12.6* мы кратко коснулись использования перекрестной валидации на основе регуляризатора, которая ранее подробно обсуждалась в *разд. 11.6*.

12.8. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:
github.com/jermwatt/machine_learning_refined.

12.1. Комплексное представление Фурье

Убедитесь, что, используя комплексные экспоненциальные определения функций косинуса и синуса, т. е. $\cos\alpha = \frac{1}{2}\left(e^{i\alpha} + e^{-i\alpha}\right)$ и $\sin\alpha = \frac{1}{2i}\left(e^{i\alpha} - e^{-i\alpha}\right)$, мы можем записать модель частичного Фурье-разложения

$$\text{model}(x, \mathbf{w}) = w_0 + \sum_{m=1}^{M}\left[\cos\left(2\pi m x\right) w_{2m-1} + \sin\left(2\pi m x\right) w_{2m}\right] \tag{12.44}$$

эквивалентным образом как

$$\text{model}(x, \mathbf{v}) = \sum_{m=-M}^{M} e^{2\pi i m x} v_m, \tag{12.45}$$

где компле́ксные веса $v_{-M}, ..., v_0, ..., v_M$ задаются в терминах действительных весов $w_0, w_1, ..., w_{2M}$ как

$$v_m = \begin{cases} \frac{1}{2}\left(w_{2m-1} - i w_{2m}\right), & \text{если } m > 0; \\ w_0, & \text{если } m = 0; \\ \frac{1}{2}\left(w_{1-2m} - i w_{-2m}\right), & \text{если } m < 0. \end{cases} \tag{12.46}$$

12.2. Комбинаторный взрыв в одночленах

Подтвердить, что число одночленных единиц в многочлене степени D комбинаторно растет во входной размерности, как указано в уравнении (12.7).

12.3. Регрессия с многочленным ядром

Воспроизвести подгонки многочленного ядра степеней $D = 1$, $D = 3$ и $D = 12$ к нелинейному набору данных, показанному на рис. 12.6.

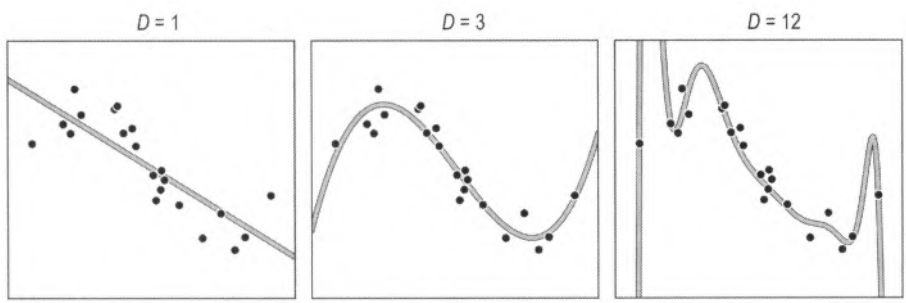


Рис. 12.6. Иллюстрация упражнения 12.3

12.4. Ядрировать ℓ_2-регуляризованную стоимость наименьших квадратов

Использовать аргумент ядризации, приведенный в примерах 12.5 и 12.6, чтобы ядрировать ℓ_2-регуляризованную стоимостную функцию наименьших квадратов.

12.5. Ядрировать стоимость многоклассового мягкого максимума

Использовать аргумент ядризации, приведенный в примерах 12.5 и 12.6, чтобы ядрировать стоимость многоклассового мягкого максимума.

12.6. Регрессия с ядром RBF

Имплементировать ядро RBF из примера 12.9 и выполнить нелинейную регрессию на наборе данных, показанном в верхнем ряду рис. 12.3, используя $\beta = 10^{-4}$, $\beta = 10^{-2}$ и $\beta = 10$ для воспроизведения соответствующих подгонок, показанных на рисунке.

12.7. Двухклассовая классификация с ядром RBF

Имплементировать ядро RBF из примера 12.9 и выполнить нелинейную двухклассовую классификацию на наборе данных, показанном в среднем ряду рис. 12.3, используя $\beta = 10^{-8}$, $\beta = 10^{-4}$ и $\beta = 10$. Для каждого случая построить график истории ошибок классифицирования, чтобы показать, что ваши результаты совпадают с тем, что показано на рисунке.

12.8. Многоклассовая классификация с ядром RBF

Имплементировать ядро RBF из примера 12.9 и выполнить нелинейную многоклассовую классификацию на наборе данных, показанном в нижнем ряду рис. 12.3, используя $\beta = 10^{-2}$, $\beta = 10^{-1}$ и $\beta = 100$. Для каждого случая построить график истории ошибок классифицирования, чтобы показать, что ваши результаты совпадают соответственно с теми, которые представлены на рисунке.

12.9. Многочленные ядра для произвольной степени и входной размерности

Показать, что многочленное ядро может быть определено поэлементно, как указано в уравнении (12.28), для общей степени D и входной размерности N.

12.10. Преобразование бесконечномерных признаков

Убедиться, что преобразование бесконечномерных признаков, определенное в уравнении (12.39), действительно производит поэлементную форму ядра RBF из уравнения (12.36).

12.11. Ядро Фурье для векторнозначного входа

Для общего N-мерного входа каждая Фурье-единица принимает форму

$$f_{\mathbf{m}}(\mathbf{x}) = e^{2\pi i m_1 x_1} e^{2\pi i m_2 x_2} \cdots e^{2\pi i m_N x_N} = e^{2\pi i \mathbf{m}^T \mathbf{x}}, \tag{12.47}$$

где вектор $\mathbf{m}$

$$\mathbf{m} = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_N \end{bmatrix} \tag{12.48}$$

содержит целочисленные элементы. Кроме того, Фурье-разложение по степени D содержит все такие единицы, удовлетворяющие $0 < \|\mathbf{m}\|_\infty \leq D$ (*см. разд. П3.5*, если вы не знакомы с нормой бесконечности). Вычислить соответствующий (i, j)-й элемент ядерной матрицы $\mathbf{H}$, т. е. $h_{i,j} = \mathbf{f}_i^T \overline{\mathbf{f}}_j$, где $\overline{\mathbf{f}}_j$ обозначает комплексный конъюгат $\mathbf{f}_j$.

12.12. Ядра и набор данных о раке

Повторить эксперимент, описанный в примере 12.10, и построить график, подобный показанному на рис. 12.5. Вы, возможно, достигните других результатов, основываясь на вашей случайной тренировочно-валидационной разбивке исходных данных.

13 Полносвязные нейронные сети

13.1. Введение

Как мы впервые увидели в *разд. 11.2.3*, искусственные нейронные сети, в отличие от многочленных и других фиксированно-контурных аппроксиматоров, имеют внутренние параметры, которые позволяют каждой из их структурных единиц принимать различные контуры. В этой главе мы подробно обсудим это введение в тему, рассмотрев общие многослойные нейронные сети, также именуемые *полносвязными сетями, многослойными персептронами* и *глубокими нейронными сетями прямого распространения*.

13.2. Полносвязные нейронные сети

В этом разделе мы опишем общие полносвязные нейронные сети, которые представляют собой рекурсивно выстраиваемые обобщения того рода структурных единиц, которые мы впервые увидели в *главе 11*. Поскольку эта тема нередко излагается запутанно, мы опишем полносвязные сети поступательно (и с некоторой избыточностью, которая, как мы надеемся, принесет пользу читателю), слой за слоем, начиная с *однослойных скрытых структурных единиц*, впервые упомянутых в *разд. 11.2.3*, предоставив алгебраическую, графовую и вычислительную точки зрения на их строительство. Затем мы кратко коснемся биологической правдоподобности полносвязных сетей и закончим этот раздел подробным описанием того, как эффективно имплементировать их на языке Python.

13.2.1. Структурные единицы с одним скрытым слоем

Общее алгебраическое представление (т. е. формула) структурной единицы с одним скрытым слоем, также именуемой однослойной единицей для краткости, впервые было приведено в уравнении (11.12) и является довольно простым: линейная комбинация входа, пропущенная через нелинейную функцию активации, обычно элементарную математическую функцию (например, tanh). Здесь мы будем обозначать такие структурные единицы в общем случае как

$$f^{(1)}(\mathbf{x}) = a\left(w_0^{(1)} + \sum_{n=1}^{N} w_n^{(1)} x_n\right), \tag{13.1}$$

где $a(\cdot)$ обозначает функцию активации, а верхние индексы у f и у всех $w_0, ..., w_N$ указывают на то, что они представляют соответственно однослойную единицу и ее внутренние веса.

Поскольку мы хотим расширить однослойную идею до создания многослойных сетей, полезно разделить последовательность двух операций, используемых для построения однослойной единицы: линейную комбинацию входа и пропускание через нелинейную функцию активации. Мы называем этот способ записи структурной единицы *рекурсивным рецептом* по созданию однослойных нейросетевых структурных единиц и подытоживаем его ниже.

Рекурсивный алгоритм для однослойных структурных единиц:

1. Выбрать функцию активации $a(\cdot)$.
2. Вычислить линейную комбинацию $v = w_0^{(1)} + \sum_{n=1}^{N} w_n^{(1)} x_n$.
3. Пропустить результат через активацию и сформировать $a(v)$.

Пример 13.1. Иллюстрирование емкости однослойных структурных единиц

В верхнем ряду на рис. 13.1 построены графики четырех экземпляров однослойной единицы с использованием tanh в качестве нашей нелинейной функции активации. Эти четыре нелинейные структурные единицы принимают форму

$$f^{(1)}(x) = \tanh\left(w_0^{(1)} + w_1^{(1)} x\right), \tag{13.2}$$

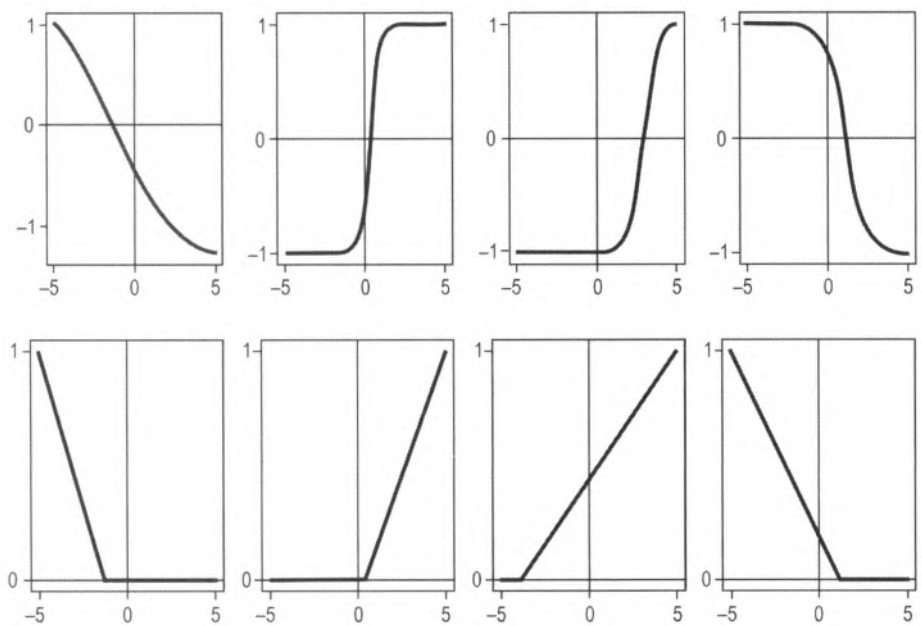


Рис. 13.1. Иллюстрация примера 13.1. Четыре экземпляра однослойной нейросетевой структурной единицы с активацией tanh (верхний ряд) и ReLU (нижний ряд)

где в каждом случае внутренние параметры структурной единицы (например, $w_0^{(1)}$ и $w_1^{(1)}$) были заданы случайно, придавая каждому экземпляру определенный контур. Это примерно иллюстрирует *емкость* каждой однослойной единицы. Емкость (понятие, впервые введенное в *разд. 11.2.2*) относится к диапазону контуров, которые может принимать такая функция с учетом всех различных настроек ее внутренних параметров.

В нижнем ряду на рис. 13.1 функция tanh заменена функцией активации ReLU[1], образуя однослойную единицу формы

$$f^{(1)}(x) = \max\left(0,\ w_0^{(1)} + w_1^{(1)} x\right). \tag{13.3}$$

В очередной раз внутренние параметры этой структурной единицы позволяют ей принимать разнообразие форм (отличных от тех, которые создаются активацией tanh), четыре экземпляра которых проиллюстрированы в нижнем ряду рисунка.

Если мы сформируем общую нелинейную модель с использованием $B = U_1$ таких однослойных единиц, как

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1^{(1)}(\mathbf{x}) w_1 + \ldots + f_{U_1}^{(1)}(\mathbf{x}) w_{U_1}, \tag{13.4}$$

j-я единица которой принимает форму

$$f_j^{(1)}(\mathbf{x}) = a\left(w_{0,j}^{(1)} + \sum_{n=1}^{N} w_{n,j}^{(1)} x_n\right), \tag{13.5}$$

тогда параметрическое множество Θ содержит не только веса окончательной линейной комбинации с w_0 по w_{U_1}, но и все параметры, внутренние для каждого $f_j^{(1)}$. Именно такого рода модель мы использовали в нейросетевых примерах в *главе 11*.

На рис. 13.2, *слева* показано общее графовое представление однослойной модели из уравнения (13.4), которое визуально распутывает индивидуальные алгебраические операции, выполняемые такой моделью. Визуальное представление, подобное этому, часто именуется *нейросетевой архитектурой*, или просто *архитектурой*. Здесь смещение и вход каждой однослойной структурной единицы, составляющей модель, показаны в виде последовательности точек на всем протяжении слева от диаграммы. Этот слой является для нас "видимым", поскольку именно здесь мы впрыскиваем входные данные в нашу сеть, которые мы сами можем "видеть", а также часто упоминается как *первый* или *входной* слой сети.

Линейная комбинация входных данных, ведущая к каждой стурктурной единице, затем визуально показана ребрами, соединяющими вход с полой окружностью (структурная единица суммирования), а нелинейная активация показана в виде большей голубой окружности (структурная единица активации). В середине этого визуального изображения (где синие окружности, представляющие все U_1 актива-

[1] Выпрямленная линейная единица или функция (Rectified Linear Unit, ReLU) была впервые введена в контексте двухклассовой классификации и стоимости персептрона в *разд. 6.4.2*.

ций, выравниваются) находится *скрытый* слой этой архитектуры. Этот слой называется "скрытым", потому что он содержит внутренне обработанные версии нашего входа, которые мы не "видим". Хотя название "скрытый" является не совсем точным (поскольку мы можем визуализировать внутреннее состояние этих единиц, если мы этого захотим), оно является общепринятой условностью, отсюда и название — структурная единица с *одним* скрытым слоем (single-hidden-layer unit). Выход из этих U_1 структурных единиц собирается в линейную комбинацию и снова визуализируется ребрами, соединяющими каждую такую единицу с окончательным суммированием, показанным в виде полой окружности. Она и является окончательным выходом из модели, который часто именуется *финальным* или *выходным* слоем сети и снова нам "виден" (а не скрыт).

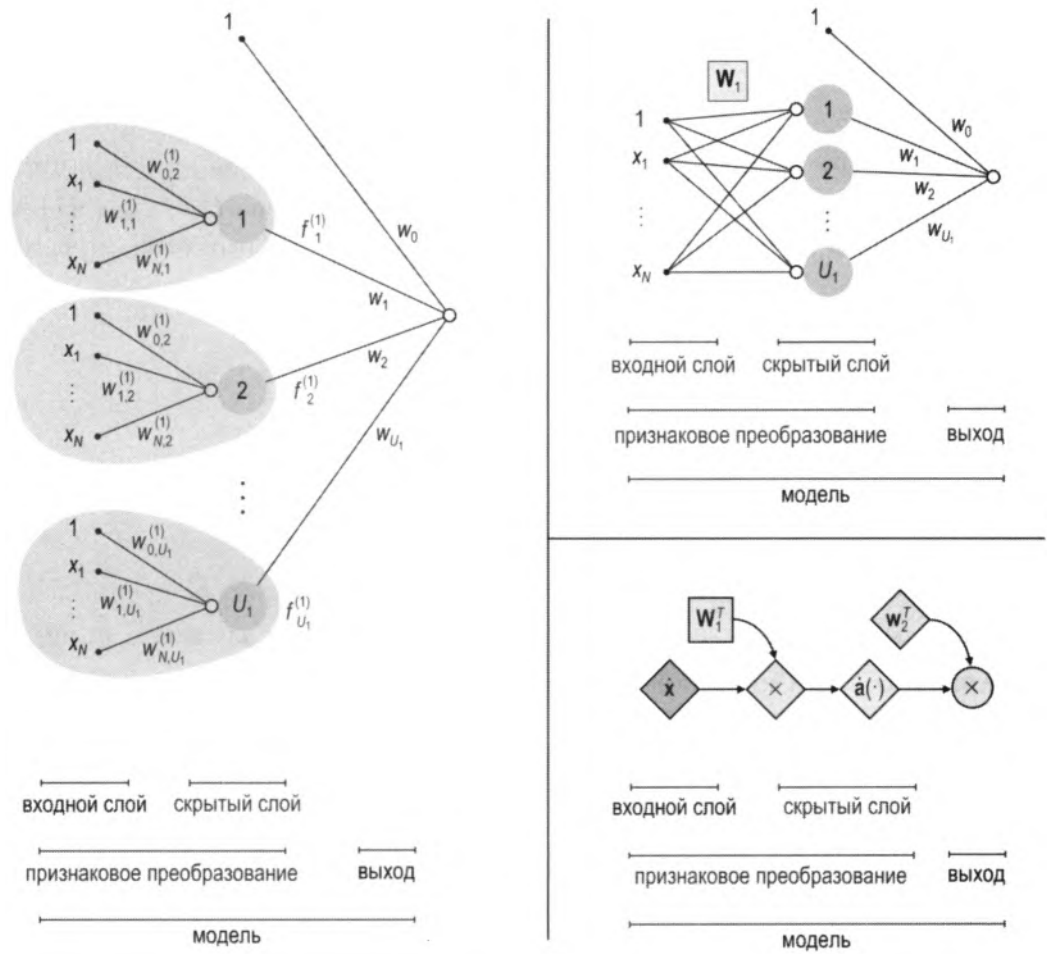


Рис. 13.2. Графовое представление однослойной нейросетевой модели, задаваемой в уравнении (13.4), которая состоит из U_1 однослойных структурных единиц. Сжатое графовое представление однослойной нейронной сети (*слева*). Эта сеть может быть представлена еще компактнее, иллюстрируя на простой диаграмме все вычисления, выполняемые однослойной нейросетевой моделью (*справа*)

Компактное представление однослойных нейронных сетей

Поскольку мы вскоре захотим добавлять больше скрытых слоев в нашу рудиментарную модель в процессе детализации многослойных сетей, такого рода визуальное изображение на рис. 13.2, *слева* быстро станет громоздким. Поэтому, чтобы сосредоточиться и лучше подготовиться к пониманию *более глубоких* нейросетевых структурных единиц, весьма полезно эту визуализацию представить более компактно. Мы можем сделать это, сначала использовав более лаконичную нотацию для представления нашей модели алгебраически, начиная с более сжатого представления нашего входа, поместив 1 в первый элемент нашего входного вектора $\mathbf{x}$, которое мы обозначим через $\dot{\mathbf{x}}$ как[2]

$$\dot{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_N \end{bmatrix}. \tag{13.6}$$

Далее соберем все *внутренние параметры* наших U_1 однослойных единиц. Рассматривая алгебраическую форму для j-й единицы в уравнении (13.5), мы видим, что она имеет $N+1$ таких внутренних параметров. Собрав эти параметры, сформируем вектор-столбец, начиная со смещения $w_{0,j}^{(1)}$, а затем входные веса $w_{1,j}^{(1)}$, ..., $w_{N,j}^{(1)}$ и поместим их в j-й столбец $(N+1)\times U_1$-матрицы:

$$\mathbf{W}_1 = \begin{bmatrix} w_{0,1}^{(1)} & w_{0,2}^{(1)} & \cdots & w_{0,U_1}^{(1)} \\ w_{1,1}^{(1)} & w_{1,2}^{(1)} & \cdots & w_{1,U_1}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N,1}^{(1)} & w_{N,2}^{(1)} & \cdots & w_{N,U_1}^{(1)} \end{bmatrix}. \tag{13.7}$$

С учетом этой нотации обратите внимание, как матрично-векторное произведение $\mathbf{W}_1^T\dot{\mathbf{x}}$ содержит каждую линейную комбинацию, *внутреннюю* для наших нелинейных единиц U_1. Другими словами, $\mathbf{W}_1^T\dot{\mathbf{x}}$ имеет размер $U_1\times 1$, и его j-я запись является именно линейной комбинацией входных данных, внутренних для j-й единицы:

$$\left[\mathbf{W}_1^T\dot{\mathbf{x}}\right]_j = w_{0,j}^{(1)} + \sum_{n=1}^{N} w_{n,j}^{(1)} x_n, \qquad j = 1, 2, ..., U_1. \tag{13.8}$$

Далее мы расширим нашу нотацию до произвольной функции активации $a(\cdot)$ для работы с таким вектором. Выражаясь конкретнее, мы определяем $\mathbf{a}(\cdot)$ как векторную функцию, которая принимает общий $(d\times 1)$-вектор $\mathbf{v}$ и возвращает — в качестве выхода — вектор того же самого размера, содержащий активацию каждой его входной записи:

---

[2] Эта нотация была введена и задействована ранее в *разд. 5.2*.

$$\mathbf{a}(\mathbf{v}) = \begin{bmatrix} a(v_1) \\ \vdots \\ a(v_d) \end{bmatrix}. \tag{13.9}$$

Обратите внимание, как с помощью этой нотации векторная активация $\mathbf{a}\left(\mathbf{W}_1^T \dot{\mathbf{x}}\right)$ становится $U_1 \times 1$-вектором, содержащим все однослойные единицы U_1, j-я из которых задается как

$$\left[\mathbf{a}\left(\mathbf{W}_1^T \dot{\mathbf{x}}\right)\right]_j = a\left(w_{0,j}^{(1)} + \sum_{n=1}^{N} w_{n,j}^{(1)} x_n\right), \qquad j = 1, 2, ..., U_1. \tag{13.10}$$

Использовав еще одну компактную нотацию для обозначения весов окончательной линейной комбинации

$$\mathbf{w}_2 = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_1} \end{bmatrix} \tag{13.11}$$

и расширив наш вектор $\mathbf{a}$, прикрепив к нему сверху 1 и обозначив результирующий $(U_1 + 1) \times 1$-вектор через $\dot{\mathbf{a}}$, мы можем, наконец, записать модель из уравнения (13.4) довольно компактно:

$$\text{model}(\mathbf{x}, \Theta) = \mathbf{w}_2^T \dot{\mathbf{a}}\left(\mathbf{W}_1^T \dot{\mathbf{x}}\right). \tag{13.12}$$

Эта более компактная алгебраическая формулировка поддается гораздо легче воспринимаемому визуальному изображению. На рис. 13.2, *справа* показана немного сокращенная версия нашего исходного графика с левой части рисунка, где линейные веса, прикрепленные к каждому входу, теперь заданы компактнее как множество пересекающихся отрезков прямой, соединяющих вход с каждой структурной единицей, где матрица $\mathbf{W}_1$ совместно представляет все U_1 взвешенных комбинаций. На рис. 13.2, *справа снизу* мы еще больше уплотнили наше первоначальное визуальное представление. В этом более компактном представлении мы можем проще визуализировать вычисления, выполняемые общей моделью однослойной нейронной сети из уравнения (13.12), скаляры, векторы и матрицы которого символически изображены соответственно в виде кругов, ромбов и квадратов.

13.2.2. Структурные единицы с двумя скрытыми слоями

Для того чтобы создать структурную нейросетевую единицу с *двумя скрытыми слоями*, или двухслойную единицу для краткости, мы применяем идею однослойной единицы, подробно описанную в предыдущем разделе, *рекурсивно*.

Мы делаем это, сначала построив множество из U_1 однослойных единиц и трактуя их как вход для еще одной нелинейной структурной единицы, т. е. мы берем их линейную комбинацию и пропускаем результат через нелинейную активацию.

Алгебраическая форма общей двухслойной единицы задается как

$$f^{(2)}(\mathbf{x}) = a\left(w_0^{(2)} + \sum_{i=1}^{U_1} w_i^{(2)} f_i^{(1)}(\mathbf{x}) \right), \tag{13.13}$$

что отражает рекурсивную природу построения двухслойных единиц с использованием однослойных. Эту рекурсивную природу можно также увидеть в приведенном ниже *рекурсивном алгоритме* построения двухслойных единиц.

Рекурсивный алгоритм для двухслойных структурных единиц:

1. Выбрать функцию активации $a(\cdot)$.
2. Построить U_1 однослойных единиц $f_i^{(1)}(\mathbf{x})$ для $i = 1, 2, ..., U_1$.
3. Вычислить линейную комбинацию $v = w_0^{(2)} + \sum_{i=1}^{U_1} w_i^{(2)} f_i^{(1)}(\mathbf{x})$.
4. Пропустить результат через активацию и сформировать $a(v)$.

Пример 13.2. Иллюстрирование емкости двухслойных структурных единиц

В верхнем ряду на рис. 13.3 построены графики четырех экземпляров двухслойной нейросетевой единицы (с использованием активации tanh) формы

$$f^{(2)}(x) = \tanh\left(w_0^{(2)} + w_1^{(2)} f^{(1)}(x)\right), \tag{13.14}$$

где

$$f^{(1)}(x) = \tanh\left(w_0^{(1)} + w_1^{(1)}(x)\right). \tag{13.15}$$

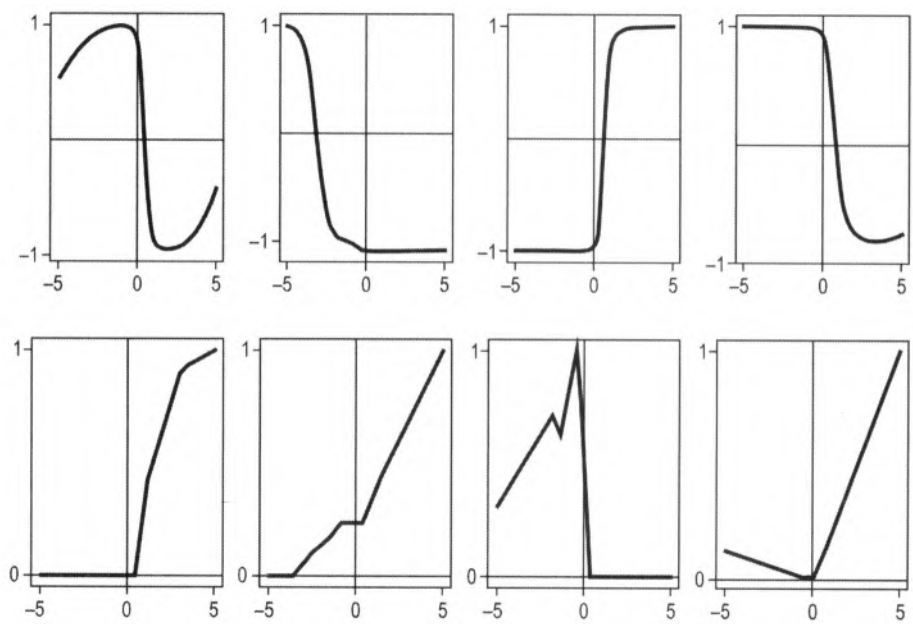


Рис. 13.3. Иллюстрация примера 13.2. Четыре экземпляра двухслойной нейросетевой единицы с активацией tanh (*сверху*) и ReLU (*снизу*)

Более широкое разнообразие контуров, принимаемых экземплярами этой структурной единицы, как показано на рисунке, отражает увеличение емкости двухслойных единиц по сравнению с их однослойными аналогами, представленными на рис. 13.1.

В нижнем ряду рис. 13.3 приведены четыре образчика одной и той же структурной единицы, только теперь в каждом слое мы используем активацию ReLU вместо tanh.

В общем случае если мы хотим создать модель с использованием $D = U_2$ двухслойных нейросетевых единиц, то пишем

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1^{(2)}(\mathbf{x})w_1 + ... + f_{U_2}^{(2)}(\mathbf{x})w_{U_2}, \tag{13.16}$$

где

$$f_j^{(2)}(\mathbf{x}) = a\left(w_{0,j}^{(2)} + \sum_{i=1}^{U_1} w_{i,j}^{(2)} f_i^{(1)}(\mathbf{x})\right), \qquad j = 1, 2, ..., U_2, \tag{13.17}$$

параметрическое множество Θ, как всегда, содержит эти (верхнеиндексные) веса, внутренние для нейросетевых единиц, а также окончательные линейно-комбинационные веса. Важно отметить, что в то время как каждая двухслойная структурная единица $f_j^{(2)}$ в уравнении (13.17) имеет уникальные внутренние параметры, обозначаемые через $w_{i,j}^{(2)}$, где i варьирует от 0 до U_1, внутренние веса каждой однослойной структурной единицы $f_j^{(1)}$ одинаковы во всех двухслойных структурных единицах.

На рис. 13.4 показано графовое представление (или архитектура) обобщенной двухслойной нейросетевой модели, алгебраическая форма которой приведена в уравнении (13.16). Слева проиллюстрирована каждая входная однослойная единица точно так же, как показано ранее на рис. 13.2, *сверху*. Входной слой на всем протяжении влево сначала подается в каждую из U_1 однослойных единиц (которые по-прежнему составляют первый *скрытый* слой сети). Линейная комбинация этих однослойных единиц затем подается в каждую из U_2 двухслойных единиц, которые условно называются вторым *скрытым* слоем, поскольку их вычисление также не является напрямую для нас "видимым". Здесь мы также увидим, почему такого рода архитектура называется *полносвязной*: каждая размерность входа связана с каждой структурной единицей в первом скрытом слое, а каждая структурная единица первого скрытого слоя связана с каждой структурной единицей второго скрытого слоя. Наконец, на всем протяжении направо от этой панели мы видим линейную комбинацию U_2 двухслойных единиц, которая создает последний (видимый) слой сети: выход нашей двухслойной модели.

Компактное представление двухслойных нейронных сетей

Как и в случае с однослойными моделями, здесь также полезно уплотнить и нашу нотацию, и соответствующую визуализацию двухслойной модели, чтобы упростить

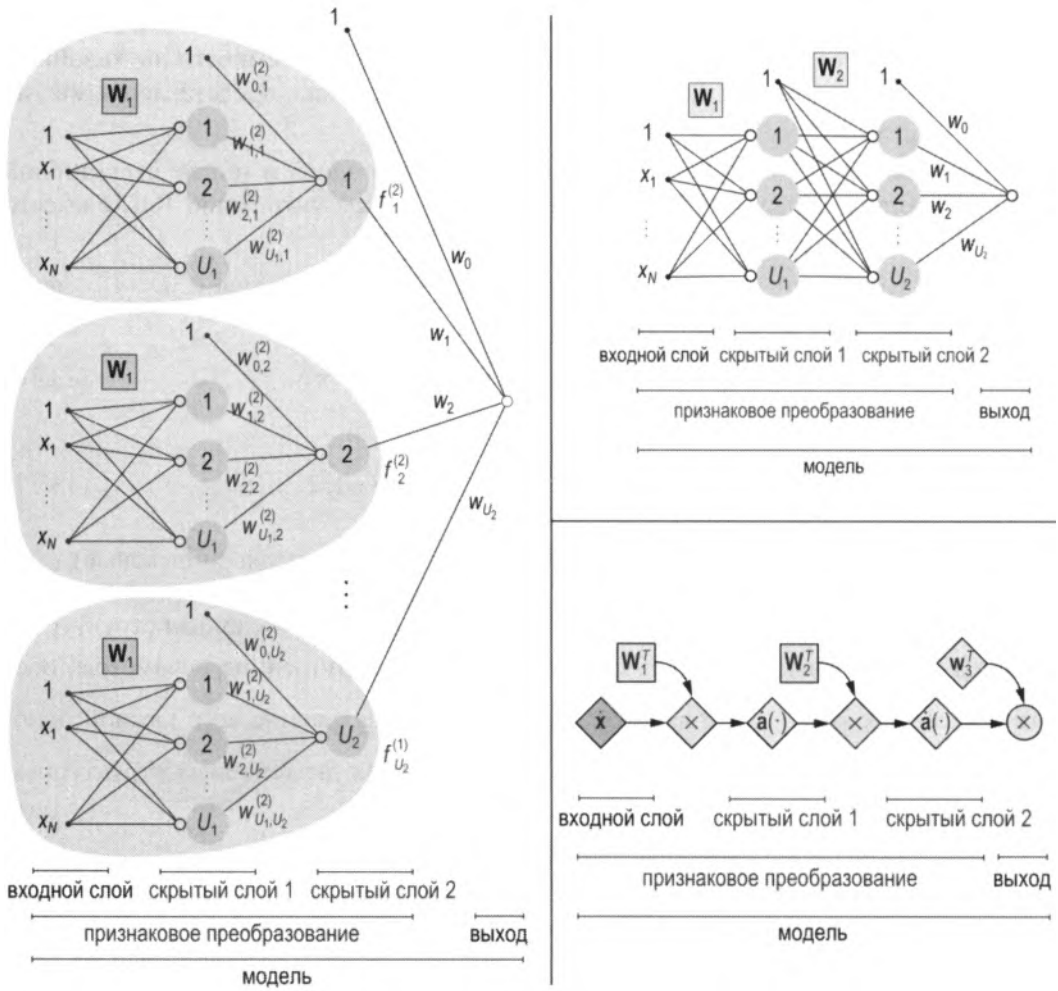


Рис. 13.4. Графовое представление двухслойной модели нейронной сети, приведенной в уравнении (13.16), которая состоит из U_2 двухслойных единиц (*слева*). Сжатое графовое представление двухслойной нейронной сети (*справа сверху*). Эта сеть может быть представлена компактнее, обеспечивая более простое изображение вычислений, выполняемых двухслойной нейросетевой моделью (*справа снизу*)

наше понимание и облегчить использование указанной концепции. Используя ту же самую нотацию, введенную в *разд. 13.2.1*, мы можем компактно обозначить выход из U_1 наших однослойных единиц как

$$\text{выход из первого скрытого слоя: } \dot{\mathbf{a}}\left(\mathbf{W}_1^T \dot{\mathbf{x}}\right). \tag{13.18}$$

Следуя тому же самому шаблону, что и раньше, мы можем сжать все внутренние веса U_2 структурных единиц во втором слое по столбцам в $(U_1+1)\times U_2$-матрицу формы:

$$\mathbf{W}_2 = \begin{bmatrix} w_{0,1}^{(2)} & w_{0,2}^{(2)} & \cdots & w_{0,U_2}^{(2)} \\ w_{1,1}^{(2)} & w_{1,2}^{(2)} & \cdots & w_{1,U_2}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{U_1,1}^{(2)} & w_{U_1,2}^{(2)} & \cdots & w_{U_1,U_2}^{(2)} \end{bmatrix}, \tag{13.19}$$

что точно отражает то, как мы определили внутреннюю весовую $(N+1)\times U_1$-матрицу $\mathbf{W}_1$ для наших однослойных структурных единиц в уравнении (13.7). Это позволяет нам также выразить выход из наших U_2 двухслойных структурных единиц компактно как

$$\text{выход из второго скрытого слоя: } \dot{\mathbf{a}}\left(\mathbf{W}_2^T\dot{\mathbf{a}}\left(\mathbf{W}_1^T\dot{\mathbf{x}}\right)\right). \tag{13.20}$$

Рекурсивная природа двухслойных структурных единиц здесь полностью обнажена. Напомним, что мы используем обозначение $\dot{\mathbf{a}}(\cdot)$ несколько свободно как векторнозначную функцию в том смысле, что она просто представляет собой взятие нелинейной активации $a(\cdot)$ *поэлементно* любого вектора, подаваемого в нее, как показано в уравнении (13.9), причем 1 добавляется вверх результата.

Конкатенирование окончательных линейно-комбинационных весов в один вектор как

$$\mathbf{w}_3 = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_2} \end{bmatrix} \tag{13.21}$$

позволяет записать полную двухслойную нейросетевую модель как

$$\text{model}(\mathbf{x}, \Theta) = \mathbf{w}_3^T\dot{\mathbf{a}}\left(\mathbf{W}_2^T\dot{\mathbf{a}}\left(\mathbf{W}_1^T\dot{\mathbf{x}}\right)\right). \tag{13.22}$$

Как и его однослойный аналог, эта компактная алгебраическая формулировка двухслойной нейронной сети поддается гораздо более удобоваримым визуальным изображениям. На рис. 13.4, *справа сверху* показана слегка сокращенная версия исходного графика из левой части рисунка, где избыточность изображения каждой однослойной структурной единицы была сведена к одному-единственному визуальному представлению. При этом мы удаляем все веса, назначенные пересекающимся ребрам, соединяющим первый и второй скрытые слои, и помещаем их в матрицу $\mathbf{W}_2$, определенную в уравнении (13.19), чтобы избежать загромождения визуализации. На рис. 13.4, *справа снизу* мы еще больше уплотняем это двухслойное изображение, где скаляры, векторы и матрицы символически изображены в виде кругов, ромбов и квадратов. Этот сильно уплотненный график обеспечивает упрощенное визуальное представление суммарного объема вычислений, выполняемых общей двухслойной нейросетевой моделью.

13.2.3. Общие структурные единицы с многочисленными скрытыми слоями

Следуя тому же шаблону, который мы видели до сих пор при описании одно- и двухслойных структурных единиц, мы можем построить общие полносвязные нейронные сети с произвольным числом скрытых слоев. Вместе с каждым добавочным скрытым слоем мы увеличиваем емкость нейросетевой структурной единицы, как мы видели ранее при переходе от однослойных единиц к двухслойным, а также в модели, построенной с использованием таких структурных единиц.

Для того чтобы построить общую нейросетевую структурную единицу с L скрытыми слоями, или сокращенно L-слойную единицу, мы просто применяем этот шаблон, который сформулировали ранее $L-1$ раз, рекурсивно, причем результирующая L-слойная единица принимает в качестве входа число U_{L-1} $(L-1)$-слойных единиц, как

$$f^{(L)}(\mathbf{x}) = a\left(w_0^{(L)} + \sum_{i=1}^{U_{L-1}} w_i^{(L)} f_i^{(L-1)}(\mathbf{x})\right). \tag{13.23}$$

Как и в случае с одно- и двухслойными структурными единицами, эта формула, возможно, будет легче восприниматься, если мы будем рассуждать о ней в терминах *рекурсивного алгоритма*, приведенного ниже.

Рекурсивный алгоритм для L-слойных структурных единиц:

1. Выбрать функцию активации $a(\cdot)$.
2. Построить число U_{L-1} $(L-1)$-слойных единиц $f_i^{(L-1)}(\mathbf{x})$ для $i = 1, 2, ..., U_{L-1}$.
3. Вычислить линейную комбинацию $v = w_0^{(L)} + \sum_{i=1}^{U_{L-1}} w_i^{(L)} f_i^{(L-1)}(\mathbf{x})$.
4. Пропустить результат через активацию и сформировать $a(v)$.

Обратите внимание, что хотя, в принципе, одна и та же функция активации не должна обязательно использоваться для всех скрытых слоев L-слойной единицы, ради простоты почти всегда используется один тип активации.

Пример 13.3. Иллюстрирование емкости трехслойных структурных единиц

В верхнем ряду на рис. 13.5 показаны четыре экземпляра трехслойной структурной единицы с активацией tanh. Большее разнообразие контуров по сравнению с одно- и двухслойными аналогами из примеров 13.1 и 13.2 отражает повышенную емкость этих единиц. В нижнем ряду повторен тот же самый эксперимент, только вместо tanh используется функция активации ReLU.

В общем случае мы можем произвести модель, состоящую из $B = U_L$ таких L-слойных единиц, как

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1^{(L)}(\mathbf{x}) w_1 + ... + f_{U_L}^{(L)}(\mathbf{x}) w_{U_L}, \tag{13.24}$$

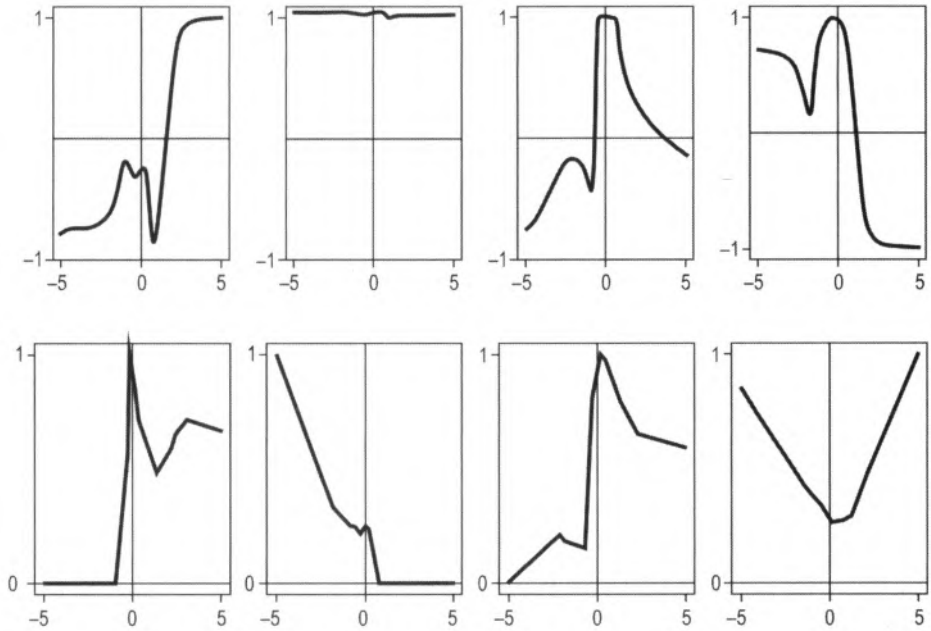


Рис. 13.5. Иллюстрация примера 13.3. Четыре экземпляра структурной единицы трехслойной сети с активацией tanh (*сверху*) и ReLU (*снизу*)

где

$$f_j^{(L)}(\mathbf{x}) = a\left(w_{0,j}^{(L)} + \sum_{i=1}^{U_{L-1}} w_{i,j}^{(L)} f_i^{(L-1)}(\mathbf{x}) \right), \qquad j = 1, 2, ..., U_L, \tag{13.25}$$

параметрическое множество Θ содержит эти веса, внутренние для нейросетевых единиц, и окончательные линейно-комбинационные веса.

На рис. 13.6 показано развернутое графовое представление этой модели, представляющее собой прямое обобщение тех видов визуализаций, которые мы видели ранее с одно- и двухслойными сетями. Слева направо мы видим входной слой сети, его L скрытых слоев и выходной слой. На жаргоне машинного обучения нередко модели, построенные с использованием трех или более скрытых слоев, называются *глубокими нейронными сетями*.

Компактное представление многослойных нейронных сетей

Для того чтобы упростить наше понимание общей многослойной нейросетевой архитектуры, мы можем использовать точно такие же компактные обозначения и визуализации, которые мы ввели в более простых контекстах одно- и двухслойных нейронных сетей. В полной аналогии с тем, как мы компактно представили двухслойные нейронные сети, обозначим выход из L-го скрытого слоя компактно:

$$\text{выход из } L\text{-го скрытого слоя: } \dot{\mathbf{a}}\left(\mathbf{W}_L^T \dot{\mathbf{a}}\left(\mathbf{W}_{L-1}^T \dot{\mathbf{a}}\left(\cdots \dot{\mathbf{a}}\left(\mathbf{W}_1^T \dot{\mathbf{x}}\right)\right)\right)\right). \tag{13.26}$$

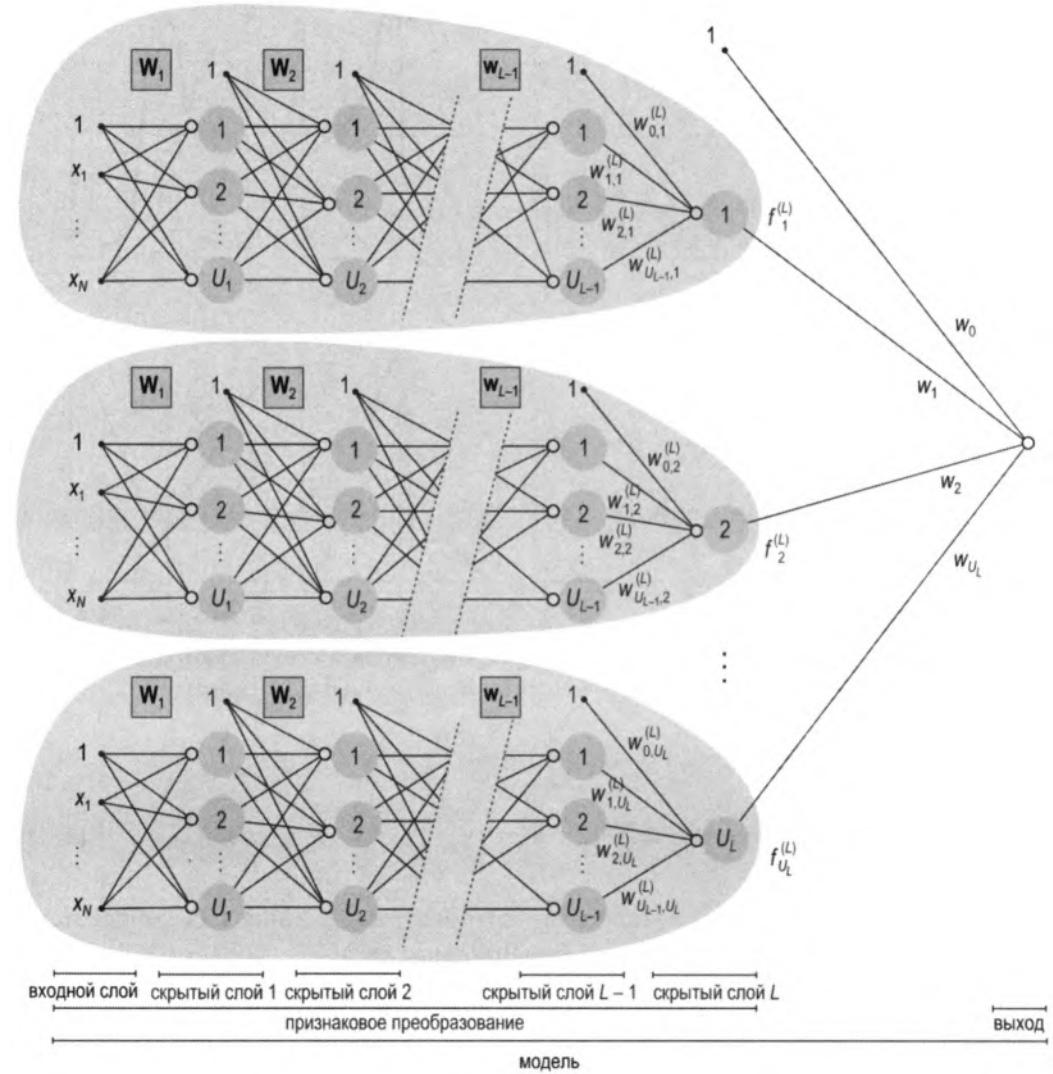


Рис. 13.6. Графовое представление модели L-слойной нейронной сети, приведенной в уравнении (13.24), состоящей из U_L L-слойных структурных единиц

Обозначив веса окончательной линейной комбинации через

$$\mathbf{w}_{L+1} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{U_L} \end{bmatrix}, \tag{13.27}$$

мы можем компактно выразить модель L-слойной нейронной сети как

$$\text{model}(\mathbf{x}, \Theta) = \mathbf{w}_{L+1}^T \dot{\mathbf{a}}\left(\mathbf{W}_L^T \dot{\mathbf{a}}\left(\mathbf{W}_{L-1}^T \dot{\mathbf{a}}\left(\cdots \dot{\mathbf{a}}\left(\mathbf{W}_1^T \dot{\mathbf{x}}\right)\right)\right)\right). \tag{13.28}$$

Опять-таки эта компактная алгебраическая формулировка L-слойной нейронной сети поддается гораздо более легко воспринимаемому визуальному изображению. Верхняя часть на рис. 13.7 показывает сокращенную версию исходного графика из рис. 13.6, где избыточность изображения каждой $(L-1)$-слойной структурной единицы была сведена к одной-единственной визуализации. Краткое визуальное изображение, показанное на рис. 13.7, *снизу*, представляет эту сетевую архитектуру еще компактнее, причем скаляры, векторы и матрицы показаны символически соответственно в виде кругов, ромбов и квадратов.

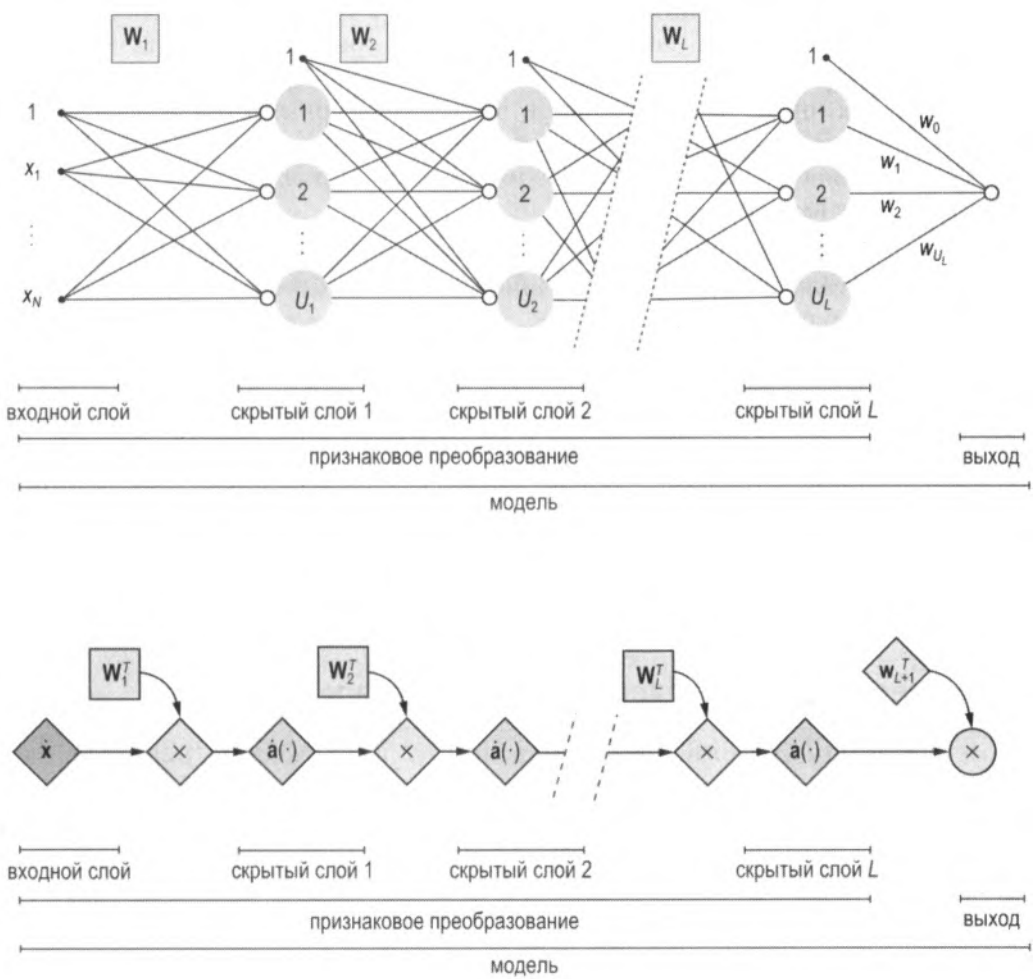


Рис. 13.7. Сжатое графовое представление модели L-слойной нейронной сети, показанной на рис.13.6 (*сверху*). Более компактная версия, кратко описывающая вычисления, выполняемые общей моделью L-слойной нейронной сети (*снизу*)

13.2.4. Отбор правильной сетевой архитектуры

Теперь мы увидели общий и рекурсивный метод построения произвольно "глубоких" нейронных сетей, но многие диковинки и технические вопросы, которые являются предметом последующих разделов в этой главе, остаются нерешенными. К ним относятся выбор функции активации, популярные методы перекрестной валидации для моделей, задействующих нейросетевые структурные единицы, а также различные вопросы, связанные с оптимизацией, такие как понятия обратного *распространения* и *пакетной нормализации*.

Однако сейчас можно обратиться к одному фундаментальному вопросу (по крайней мере в целом), а именно, как выбрать "правильное" число структурных единиц и слоев для нейросетевой архитектуры. Как и в случае выбора надлежащего универсального аппроксиматора в целом (*см. разд. 11.8*), обычно мы не знаем априори, какая архитектура будет работать лучше всего для конкретного набора данных (с точки зрения числа скрытых слоев и числа структурных единиц в расчете на скрытый слой). В принципе, для того чтобы определить наилучшую архитектуру для использования с конкретным набором данных, мы должны перекрестно валидировать массив вариантов.

При этом мы отмечаем, что, вообще говоря, емкость, получаемая при добавлении новых индивидуальных структурных единиц в модель нейронной сети, обычно намного меньше по сравнению с емкостью, получаемой при добавлении новых скрытых слоев. Это обусловлено тем, что добавление в архитектуру дополнительного *слоя* добавляет дополнительную рекурсию к вычислениям, участвующим в каждой структурной единице, что значительно увеличивает их емкость и емкость любой соответствующей модели, как мы видели в примерах 13.1–13.3. На практике выполнение модельного поиска в различных нейросетевых архитектурах обходится дорого, и поэтому необходимо идти на компромиссы, направленные на определение высококачественной модели с использованием минимальных вычислений. С этой целью в нейросетевых моделях обычно задействуется регуляризация на основе ранней остановки (*см. разд. 11.6.2* и *13.7*).

13.2.5. Нейронные сети: биологическая точка зрения

Человеческий мозг содержит приблизительно 10^{11} биологических нейронов, которые работают совместно, когда мы выполняем когнитивные задачи. Даже при выполнении относительно малых задач мы задействуем ощутимый ряд взаимосвязанных нейронов — именуемых *биологическими нейронными сетями* — для правильных действий. Например, где-то порядка 10^5–10^6 нейронов требуется для того, чтобы визуализировать реалистичное изображение нашего виртуального окружения. Большинство базовых терминологических жаргонизмов и принципов моделирования нейросетевых универсальных аппроксиматоров, которые мы видели до сих пор, возникли как (очень) грубая математическая модель таких биологических нейронных сетей.

Отдельный биологический нейрон (рис. 13.8, *слева вверху*) состоит из трех главных частей: *дендритов* (приемников нейрона), *сомы* (клеточного тела) и *аксона* (пере-

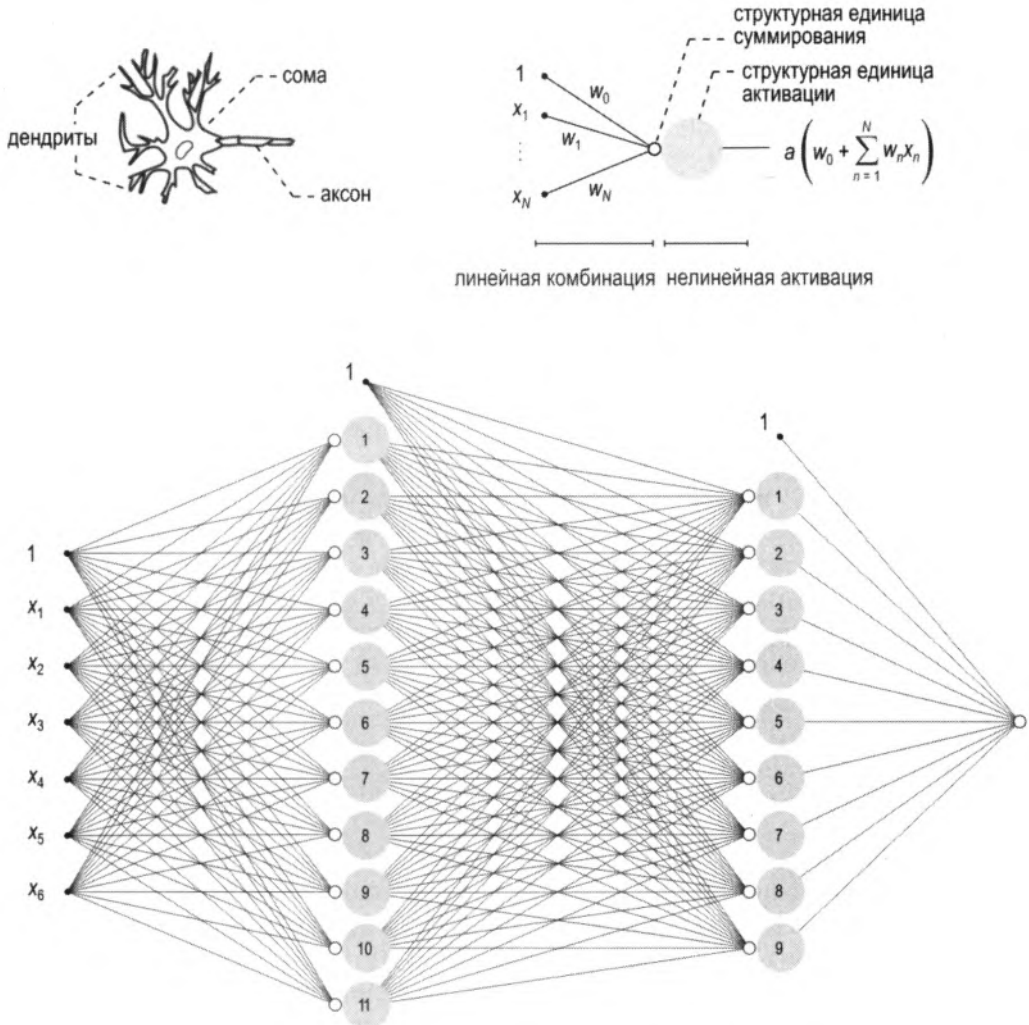


Рис. 13.8. Типичный биологический нейрон (*слева сверху*). Искусственный нейрон, т. е. упрощенная математическая модель биологического нейрона, состоящая, во-первых, из взвешенных ребер, представляющих индивидуальные умножения (1 на w_0, x_1 на w_1 и т. д.); во-вторых, из структурной единицы суммирования, показанной в виде малого кружка, представляющего собой сумму $w_0 + w_1 x_1 + \ldots + w_N x_N$, и в-третьих, из структурной единицы активации, показанной в виде более крупного голубого круга, представляющего сумму, вычисленную нелинейной функцией активации $a(\cdot)$ (*справа сверху*). Пример полносвязной двухслойной нейронной сети, обычно иллюстрируемый при детализации нейронных сетей с биологической точки зрения (*снизу*)

датчика нейрона). Начиная примерно с 1940-х годов психологи и нейробиологи, разделяя общее желание лучше понять человеческий мозг, заинтересовались математическим моделированием нейронов. Эти ранние модели, позднее получившие название искусственных нейронов (базовый образчик которых показан на рис. 13.8, *справа вверху*), в 1957 году увенчались появлением модели *перцептрона* [59]. Точ-

но имитируя структуру биологического нейрона, искусственный нейрон содержит набор дендритоподобных ребер, соединяющих его с другими нейронами, каждый из которых принимает вход и умножает его на (синаптический) вес, связанный с этим ребром. Эти взвешенные входы суммируются после их пропускания через структурную *единицу суммирования* (показанную малыми кружками). Результат впоследствии подается в структурную *единицу активации* (показанную большими голубыми кружками), выход из которого затем передается наружу через аксоноподобную проекцию. С биологической точки зрения считается, что нейроны остаются неактивными до тех пор, пока чистый вход, поступающий в клеточное тело (сому), не достигнет определенного порога, и в этот момент нейрон *активируется* и запускает электрохимический сигнал, отсюда и название *функции активации*.

Нанизывание больших наборов таких искусственных нейронов слоями создает более математически сложную модель биологической нейронной сети, которая тем не менее представляет собой *очень* простую аппроксимацию того, что происходит в головном мозге. На рис. 13.8, *снизу* показано графовое представление такого рода (здесь двухслойная сеть), которое используется при рассмотрении нейронных сетей с этой биологической точки зрения. В этом довольно сложном визуальном изображении каждая мультипликативная операция архитектуры изображена как ребро, создающее сетку пересекающихся линий, соединяющих каждый слой. При описании полносвязных нейронных сетей в *разд. 13.2.1–13.2.3* мы предпочитали более простые и компактные визуализации, которые избегают такого рода сложных визуальных сеток.

13.2.6. Имплементация на Python

В этом разделе мы покажем, как реализовать универсальную модель, состоящую из L-слойных нейросетевых единиц, на Python с использованием библиотеки NumPy. Поскольку такие модели могут иметь большое число параметров, а соответствующие стоимостные функции, задействующие их, могут быть сильно невыпуклыми (как далее обсуждается в *разд. 13.5*), методы локальной оптимизации первого порядка (например, градиентный спуск и его варианты, подробно описанные в *главе 3* и *приложении 1*) являются наиболее распространенными схемами, используемыми для настройки параметров общих моделей нейронных сетей. Более того, поскольку вычислять производные модели, задействуя нейросетевые структурные единицы "вручную", чрезвычайно утомительно (*см. разд. 13.4*), использовать автоматическое дифференцирование не только полезно, но и в большинстве случаев необходимо. Таким образом, в описанной здесь Python-реализации мы настоятельно рекомендуем применять библиотеку autograd — автоматический дифференциатор, используемый во всей этой книге (*см. разд. П2.10.2*), — чтобы сделать вычисление производных чистым и простым.

Далее приведена реализация нейросетевого признакового преобразования `feature_transforms` (обозначение, которое мы впервые ввели в *главе 10*).

```
1| # нейросетевое признаковое преобразование
2| def feature_transforms(a, w):
3|
```

```
    # перебрать каждый слой
    for W in w:

    # вычислить внутреннее произведение с весами текущего слоя
    a = W[0] + np.dot(П1.T, W[1:])

    # пропустить через активацию
    a = activation(a).T

    return a
```

Эта Python-функция принимает вход $\mathbf{x}$, записанный как переменная `a`, и весь набор весовых матриц $\mathbf{W}_1$, ..., $\mathbf{W}_L$, внутренних для нашей нейросетевой архитектуры, записанной как переменная `w`, где `w = [W_1, W_2, ..., W_L]`. Выход из нашей функции признакового преобразования — это выход из конечного слоя сети, выраженный алгебраически в уравнении (13.26). Мы вычисляем этот выход рекурсивно, перебирая в цикле *вперед* по сетевой архитектуре, начиная с первого скрытого слоя с использованием матрицы $\mathbf{W}_1$ и заканчивая вычислением последнего скрытого слоя с использованием $\mathbf{W}_L$. В результате получаем Python-функцию, состоящую из простого цикла `for` над весовыми матрицами скрытых слоев. Обратите внимание, что в строке 11 приведенной выше реализации `activation` может относиться к любой элементарной функции, построенной с использованием операций NumPy. Например, активация tanh может быть записана как `np.tanh(a)`, а активация ReLU — как `np.maximum(0,a)`.

Теперь, когда наша функция признакового преобразования завершена, мы можем реализовать нашу модель, являющуюся простой вариацией реализаций, которые мы видели в предыдущих главах. Здесь входами для этой Python-функции служат `x` — наши входные данные — и Python-список `theta` длины два, первая запись которого содержит наш список внутренних весовых матриц, а вторая запись содержит веса окончательной линейной комбинации.

```
# нейросетевая модель
def model(x, theta):

    # вычислить признаковое преобразование
    f = feature_transforms(x, theta[0])

    # вычислить окончательную линейную комбинацию
    a = theta[1][0] + np.dot(f.T, theta[1][1:])

    return a.T
```

Эта имплементация модели полносвязной нейронной сети может быть легко сопряжена с Python-реализацией обобщенными стоимостными функциями машинного обучения, подробно описанными в предыдущих главах.

Наконец, мы предоставляем Python-функцию `network_initializer`, которая создает начальные веса для общей нейросетевой модели, а также предоставляет простой

интерфейс для создания общих архитектур: именно этот инициализатор определяет контур нашей реализованной сети. Для того чтобы создать желаемую сеть, мы просто вводим разделенный запятыми список размеров слоев `layer_sizes` следующей формы

```
layer_sizes = [N, U_1, ..., U_L, C]
```

где `N` — это входная размерность; `U_1`, ..., `U_L` — число желаемых структурных единиц в скрытых слоях соответственно с 1 по L; `C` — выходная размерность.

Затем инициализатор автоматически создаст начальные весовые матрицы (соответствующих размеров), а также окончательные веса для линейной комбинации, упакованные вместе как выход `theta_init`.

```
# создание начальных весов для нейросетевой модели
def network_initializer(layer_sizes, scale):

    # контейнер для всех настраиваемых весов
    weights = []

    # создать надлежаще размеренную первоначальную
    # весовую матрицу для каждого слоя сети
    for k in range(len(layer_sizes)-1):

        # получить размеры слоев для текущей весовой матрицы
        U_k = layer_sizes[k]
        U_k_plus_1 = layer_sizes[k+1]

        # изготовить весовую матрицу
        weight = scale*np.random.randn(U_k+1, U_k_plus_1)
        weights.append(weight)

    # упаковать веса так, чтобы theta_init[0] содержал все
    # весовые матрицы, внутренние для сети, а theta_init[1]
    # содержал окончательные линейно-комбинационные веса
    theta_init = [weights[:-1], weights[-1]]

    return theta_init
```

Далее мы приведем несколько примеров использования этой реализации.

Пример 13.4. Нелинейная классификация с использованием многослойных нейронных сетей

В этом примере мы используем многослойную архитектуру для выполнения нелинейной классификации, сначала на двухклассовом наборе данных, показанном ранее в примере 11.9. Здесь мы произвольно выбираем сеть, имеющую четыре скрытых слоя с десятью структурными единицами в каждом слое, и активацию tanh. Затем мы настраиваем параметры этой модели, минимизируя связанную с ней стоимость двухклассового мягкого максимума (см. уравнение (10.31)) посредством

градиентного спуска, визуализируя усвоенную нелинейную границу решения в верхнем ряду на рис. 13.9 вместе с самим набором данных.

Далее мы выполняем многоклассовую классификацию на многоклассовом наборе данных, впервые показанном в примере 10.6 ($C = 3$), используя модель, состоящую из двух скрытых слоев, выбирая число структурных единиц в каждом слое произвольно соответственно как $U_1 = 12$ и $U_2 = 5$, активацию tanh и используя общую схему (т. е. сетевая архитектура является общей для каждого классификатора, как подробно описано в *разд. 10.5* для общих признаковых преобразований). Затем мы настраиваем параметры этой модели, минимизируя соответствующую стоимость многоклассового мягкого максимума (как показано в уравнении (10.42)), и демонстрируем результирующую усвоенную границу решения в нижнем ряду на рис. 13.9.

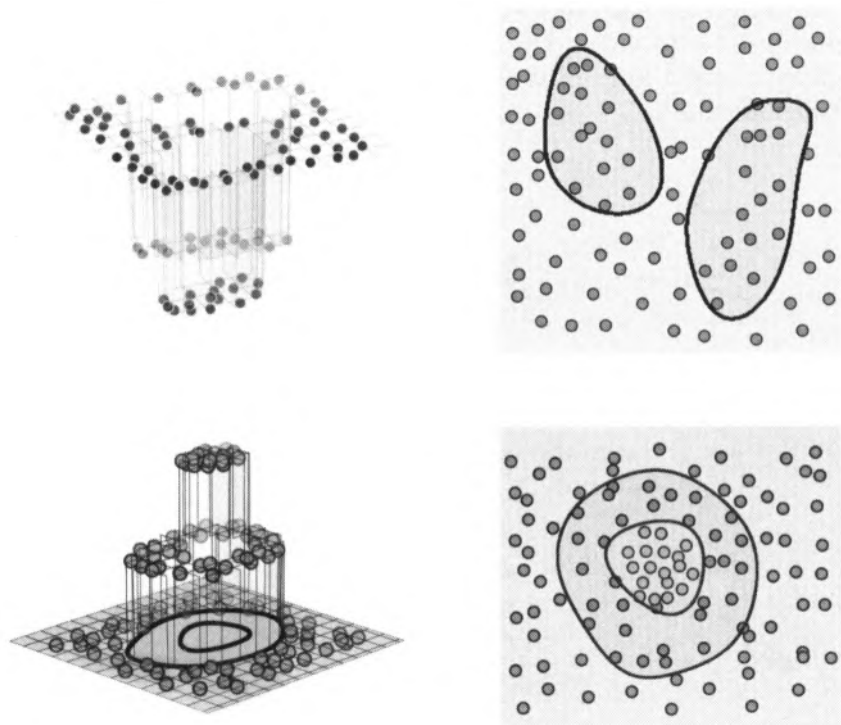


Рис. 13.9. Иллюстрация примера 13.4. Результирующая граница решения, усвоенная полносвязной нейронной сетью на двухклассовом наборе данных (*сверху*) и многоклассовом наборе данных (*снизу*) как с точки зрения регрессии (*слева*), так и с точки зрения персептрона (*справа*)

Пример 13.5. Случайные автокодировочные топологические многообразия

В *разд. 10.6.1* мы ввели общий нелинейный автокодировщик, который состоит из двух нелинейных функций: кодировщика $\mathbf{f}_e$ и декодировщика $\mathbf{f}_d$, параметры кото-

рых мы настраиваем так, чтобы (в идеале) композиция $\mathbf{f}_d(\mathbf{f}_e(\mathbf{x}))$ образовывала наилучшее *нелинейное топологическое многообразие*, на котором покоится входной набор данных. Другими словами, при заданном наборе входных точек $\{\mathbf{x}_p\}_{p=1}^{P}$ мы стремимся настроить параметры нашей пары "кодировщик/декодировщик" таким образом, чтобы

$$\mathbf{f}_d(\mathbf{f}_e(\mathbf{x})) \approx \mathbf{x}_p, \qquad p = 1, 2, ..., P. \tag{13.29}$$

На рис. 13.10 визуализированы девять экземпляров функции $\mathbf{f}_d(\mathbf{f}_e(\mathbf{x}))$, каждый из которых технически называется топологическим многообразием, где $\mathbf{f}_d$ и $\mathbf{f}_e$ являются пятислойными нейронными сетями с десятью структурными единицами в каждом слое и функцией sinc в качестве активации. Все веса в каждом случае задаются случайно, чтобы показать тип нелинейных топологических многообразий, которые мы потенциально могли бы обнаружить с помощью таких кодировочной/декодировочной функций. Из-за чрезвычайно высокой емкости этой нелинейной автокодировочной модели контуры экземпляров, показанных на рисунке, довольно разнообразны.



Рис. 13.10. Иллюстрация примера 13.5. Девять случайных топологических многообразий, генерируемых многослойным нейросетевым автокодировщиком

Пример 13.6. Нелинейный автокодировщик с использованием многослойных нейронных сетей

В этом примере мы иллюстрируем использование многослойного нейросетевого автокодировщика для усвоения нелинейного топологического многообразия над набором данных, показанное на рис. 13.11, *слева сверху*. Здесь и для кодировщика, и для декодировщика мы обычно используем трехслойную полносвязную сеть с десятью структурными единицами в каждом слое и активацией tanh. Затем мы

настраиваем параметры обеих функций вместе, минимизируя стоимость наименьших квадратов, приведенную в уравнении (10.53), и раскрываем надлежащее нелинейное топологическое многообразие, на котором покоится набор данных. На рис. 13.11 показаны: *справа сверху* — усвоенное топологическое многообразие; *слева снизу* — декодированная версия исходного набора данных, т. е. исходный набор данных, спроецированный на наше усвоенное топологическое многообразие; *справа снизу* — *проекционная карта*, визуализирующая то, как все данные в этом пространстве проецируются на усвоенное топологическое многообразие посредством графика векторного поля.

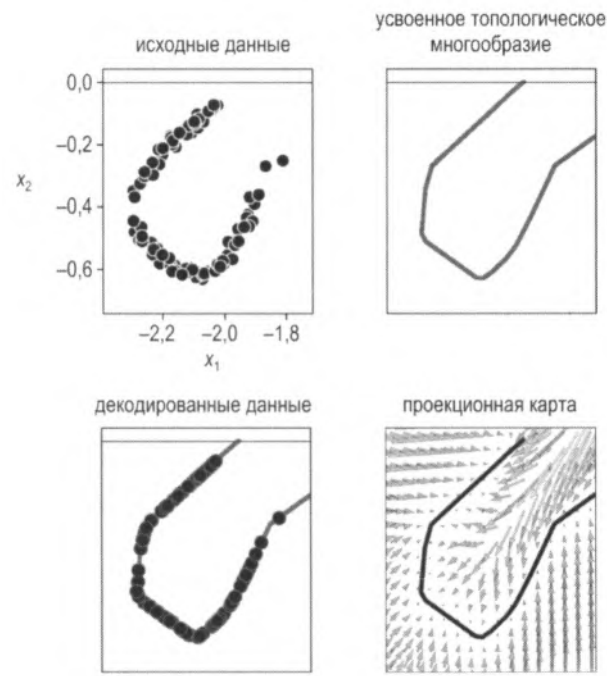


Рис. 13.11. Иллюстрация примера 13.6

13.3. Функции активации

В качестве активации, в принципе, можно использовать любую (нелинейную) функцию для полносвязной нейронной сети. В течение некоторого времени после их изобретения активации выбирались главным образом на основе их биологической правдоподобности, поскольку именно с этой точки зрения первоначально рассматривались сами нейронные сети (как подробно описано в *разд. 13.2.5*). Сегодня функции активации выбираются исходя из практических соображений, включая нашу способность правильно оптимизировать модели, которые их задействуют, а также (разумеется) общую результативность, которую они обеспечивают. В этом разделе мы кратко рассмотрим популярные классические и современные функции активации на нескольких примерах.

Пример 13.7. Шаговая и сигмоидная активации

Как широко обсуждалось в *разд. 13.2.5*, концепция нейронных сетей была впервые введена с биологической точки зрения, где каждая структурная единица архитектуры имитирует биологический нейрон в человеческом мозге. Считалось, что такие нейроны действуют как цифровые переключатели, являясь либо полностью "включенными", либо "выключенными" для передачи информации соединенным клеткам. Это убеждение естественным образом привело к использованию *шаговой функции*, принимающей только два значения: 0 (выкл.) и 1 (вкл.). Однако этот вид шаговой функции (которую мы обсуждали в контексте логистической регрессии в *разд. 6.2*) приводит к кусочным плоским стоимостным функциям (см., например, рис. 6.3, *слева*), которые чрезвычайно трудно оптимизировать с помощью любого метода локальной оптимизации. В контексте логистической регрессии как раз такая проблема и привела к возникновению *логистической сигмоиды*, и по той же практической причине сигмоидная функция была одной из первых широко используемых функций активации. В качестве гладкой аппроксимации шаговой функции логистическая сигмоида рассматривалась как разумный компромисс между желаемой нейронной моделью и практической необходимостью надлежащей настройки параметров.

Хотя логистическая сигмоида очень хорошо работает в сравнительно более простом контексте линейной классификации, при использовании в качестве функции активации она часто приводит к технической проблеме, известной как проблема *исчезающего градиента*. Обратите внимание, как логистическая сигмоидная функция (показанная на рис. 13.12, *слева сверху*) отображает почти все отрицательные входные значения (за исключением тех, которые находятся вблизи начала координат) на выходные значения, очень близкие к нулю, а ее производная (показанная на рис. 13.12, *слева снизу*) отображает входные значения вдали от начала координат на выходные значения, очень близкие к нулю. Эти характеристики могут привести к нежелательному сжатию градиента нейросетевой модели, задействующей сигмоидные активации, что препятствует надлежащей настройке параметров — проблема, которая возникает по мере добавления новых скрытых слоев.

На практике нейросетевые модели, задействующие функцию гиперболического тангенса (tanh), обычно работают лучше, чем та же сеть, задействующая логистические сигмоидные активации, поскольку сама функция центрирует свой выход вокруг начала координат. Однако, поскольку производная функции tanh схожим образом отображает входные значения вдали от начала координат на выходные значения, очень близкие к нулю, нейронные сети, задействующие активацию tanh, также могут страдать от проблемы исчезающего градиента.

Пример 13.8. Функция активации ReLU

В течение десятилетий после того, как впервые были введены полносвязные нейронные сети, исследователи задействовали почти исключительно логистические сигмоидные функции активации, основанные на их биологически вероятной природе (изложенной в *разд. 13.2.5*). Только в начале 2000-х годов некоторые исследо-

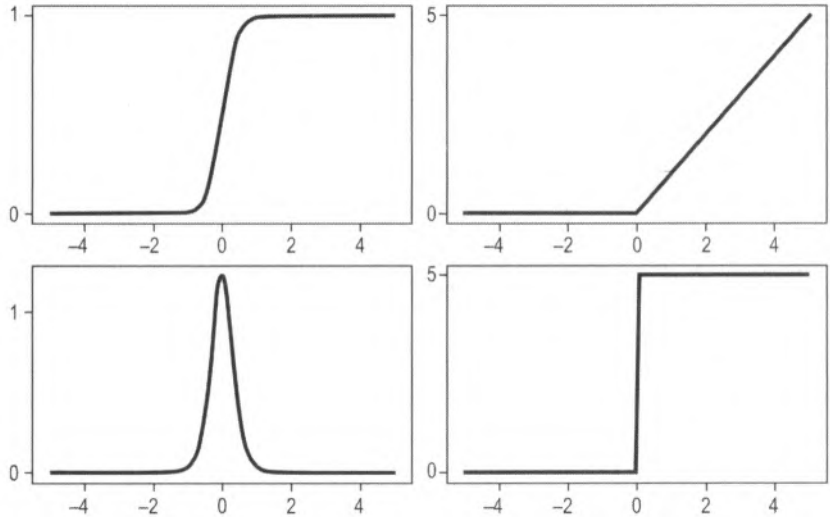


Рис. 13.12. Иллюстрация примеров 13.7 и 13.8. Логистическая сигмоидная функция (*слева сверху*) и ее производная (*слева снизу*). Производная здесь отображает большинство входных значений вдали от начала координат на выходные значения, очень близкие к нулю, что может привести к тому, что градиент сети, задействующей сигмоидную активацию, исчезнет во время оптимизации и, таким образом, не даст адекватно настроить параметры. Функция ReLU (*справа сверху*) и ее производная (*справа снизу*). Хотя нейронные сети, задействующие ReLU, не так чувствительны к проблемам исчезающего градиента, они должны инициализироваься вдали от нуля, чтобы предотвратить исчезновение структурных единиц

ватели начали отходить от этой традиции, занимаясь и тестируя альтернативные активации [60]. Простая функция ReLU (см. пример 13.1) была первой такой функцией активации, которая стала популяризироваться.

Вычислительно более простая функция по сравнению с логистической сигмоидой (которая предусматривает использование как логарифмической, так и экспоненциальной функции) ReLU быстро стала самой популярной применяемой сегодня функцией активации. Поскольку производная функции ReLU (изображенная на рис. 13.12, *справа снизу*) отображает на ноль только отрицательные входные значения, сети, задействующие эту активацию, как правило, не страдают (так сильно) от проблемы исчезающего градиента, обычно встречающейся в логистической сигмоидной активации (подробно описанной в предыдущем примере). Однако из-за контура самой ReLU (показанной на рис. 13,12, *справа сверху*) по-прежнему необходимо проявлять осторожность во время инициализирования и тренировки сети, задействующей активации ReLU, т. к. сами структурные единицы ReLU исчезают также при неположительных входах. Например, аналогично необходимости инициализировать стоимостную функцию ReLU, подробно описанную в *разд. 6.4*, вдали от начала координат, где процесс оптимизации будет останавливаться, полносвязная нейронная сеть, задействующая активации ReLU, должна инициализироваться вдали от начала координат во избежание исчезновения слишком болшого числа структурных единиц (и их градиентов).

Пример 13.9. Активация maxout

Активация maxout, определяемая как

$$a(x) = \max(v_0 + v_1 x,\ w_0 + w_1 x), \tag{13.30}$$

является родственницей ReLU, который принимает максимум две линейные комбинации входа (вместо одной линейной комбинации и нуля, как это имеет место с ReLU). Четыре экземпляра такой структурной единицы maxout показаны на рис. 13.13, где в каждом случае параметры v_0, v_1, w_0 и w_1 задаются случайно. Хотя это изменение кажется алгебраически довольно незначительным, многослойные нейросетевые архитектуры, задействующие активацию maxout, как правило, имеют определенные преимущества по сравнению с теми, которые задействуют активации tanh и ReLU, включая, во-первых, меньшее число трудностей с проблемными инициализациями, во-вторых, меньшее число проблем с исчезновением градиентов и, в-третьих, эмпирически более быстрое схождение с гораздо меньшим числом шагов градиентного спуска [61]. Эти преимущества имеют простую цену: активация maxout имеет в два раза больше внутренних параметров, чем ReLU или tanh, поэтому архитектуры, построенные с их использованием, имеют примерно в два раза больше настраиваемых параметров.

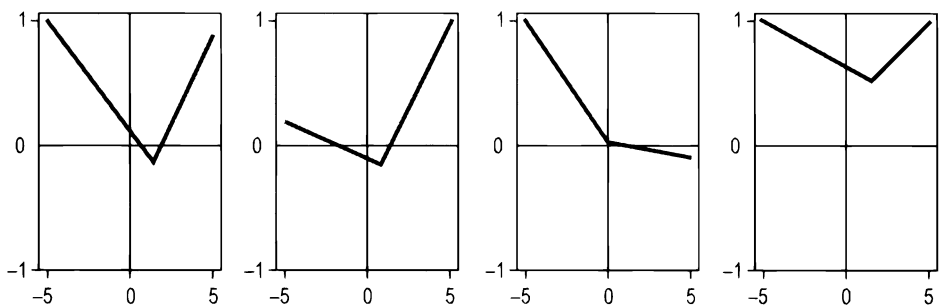


Рис. 13.13. Иллюстрация примера 13.9. Четыре экземпляра структурной единицы maxout из уравнения (13.30), причем параметры в каждом случае задаются случайно

13.4. Алгоритм обратного распространения

Алгоритм обратного распространения, часто именуемый просто обратным распространением (backprop), — это жаргонизм, используемый в машинном обучении для описания подхода к алгоритмическому вычислению градиентов с помощью компьютерной программы, которая особенно эффективна для моделей, задействующих многослойные нейронные сети. Обратное распространение является частным случаем более общей схемы, обычно именуемой *обратным режимом автоматического дифференцирования*. Заинтересованному читателю рекомендуется ознакомиться с *разд. П2.6* и *П2.7*, где изложены подробности об автоматическом дифференцировании.

Автоматическое дифференцирование позволяет нам переложить утомительное бремя вычисления градиентов вручную на неутомимого работника, которым является современный компьютер.

Другими словами, автоматический дифференциатор — это эффективный *вычислитель*, который превращает вычисление градиентов практически любой стоимостной функции просто в рутину. Как и многие технические достижения на протяжении всей истории, автоматическое дифференцирование было открыто и потом открывалось заново разными исследователями в различных областях науки и техники в разное время. Именно по этой причине это универсально применимое понятие (автоматическое дифференцирование) называется *обратным распространением* в сообществе машинного обучения, т. к. именно это название дали ему его первооткрыватели в этой области.

13.5. Оптимизация нейросетевых моделей

В типичной ситуации стоимостные функции, связанные с моделями полносвязных нейронных сетей, сильно невыпуклы. В этом разделе мы рассмотрим этот вопрос, выделив методы локальной оптимизации (*см. приложение 1*), которые особенно полезны при надлежащей настройке таких моделей.

13.5.1. Невыпуклость

Модели, задействующие многослойные нейронные сети, практически всегда невыпуклы. Однако они часто демонстрируют различную невыпуклость, с которой мы можем довольно легко справляться, используя передовые методы оптимизации, подобные описанным в *приложении 1*. Например, в *разд. 6.2.4* мы изучили контур стоимости наименьших квадратов для логистической регрессии, которая задействует сигмоидную модель. При одномерном входе эта модель, записанная

$$\text{model}(x, \Theta) = \sigma(w_0 + w_1 x), \tag{13.31}$$

может быть интерпретирована через призму нейронных сетей как одна единица однослойной нейронной сети со скалярным входом и логистической сигмоидной активацией, где веса окончательной линейной комбинации фиксированы (причем смещение равно нулю, а вес, касающийся $\sigma(\cdot)$, равен 1). На рис. 6.3, *в центре* была показана поверхность стоимости наименьших квадратов этой модели по сравнению с простым набором данных, представленным на рис. 6.2.

Рассматривая общий контур этой стоимостной функции, мы ясно видим, что она невыпуклая. Несколько участков поверхности стоимости, по обе стороны *длинной узкой долины*, содержащей глобальный минимум стоимости, являются почти полностью *плоскими*. В общем случае стоимостные функции, задействующие нейросетевые модели, имеют невыпуклые контуры, которые разделяют между собой общие базовые характеристики, представленные на этом рисунке: длинные узкие долины, плоские области и много седловых точек и локальных минимумов.

Такого рода невыпуклости представляют проблему для *базовых* схем оптимизации первого и второго порядков. Однако оба каркаса можно расширить, чтобы лучше

справляться с такими экстравагантностями. Это особенно верно для градиентных методов, поскольку мощные модификации метода стандартного градиентного спуска позволяют ему легко справляться с *длинными узкими долинами* и *плоскими областями* невыпуклых функций. Эти модификации включают импульсно обусловленные (*см. разд. П1.2*) и нормализованные градиентные методы (*см. разд. П1.3*). Объединение этих двух модификаций (*см. разд. П1.4*) в дополнение к минипакетной оптимизации (*см. разд. 7.8* и *П1.5*) может еще больше улучшить градиентные методы, благодаря чему они способны легче минимизировать стоимостные функции, как показано на рис. 6.3, *в центре*, и нейросетевые модели в целом. Одним словом, такого рода невыпуклости, встречающиеся у нейросетевых моделей, часто поддаются управлению с помощью передовых инструментов оптимизации.

Даже когда стоимостные функции нейронной сети имеют много *локальных минимумов*, они тяготеют к тому, чтобы лежать на глубине, близкой к их глобальным минимумам, и таким образом, как правило, обеспечивают схожую результативность, если их отыскивать посредством локальной оптимизации. Абстрактное изображение такого рода прототипической стоимостной функции, часто встречающейся в нейросетевых моделях, показано на рис. 13.14, сверху.

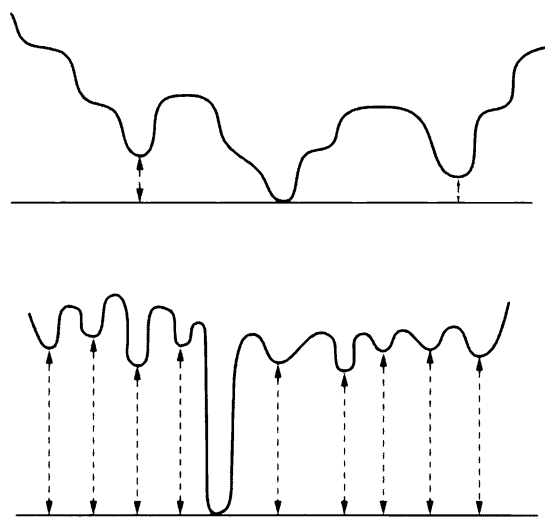


Рис. 13.14. Абстрактное изображение прототипической стоимостной функции, наблюдаемой при задействовании нейросетевых моделей. Такая стоимость может содержать много седловых точек, длинных узких долин и локальных минимумов, глубина которых близко совпадает с ее глобальными минимумами. С помощью передовых методов локальной оптимизации мы можем легко пересекать седловые точки и длинные узкие долины, определяя такие локальные минимумы (*сверху*). Абстрактное изображение невыпуклой функции, редко встречающейся при использовании нейросетевых моделей, для наихудшего сценария. Здесь разница в глубине между локальными и глобальными минимумами стоимости является существенной, и поэтому модели, задействующие параметры из каждой области, будут существенно различаться по качеству. Учитывая обширное число слабых локальных минимумов, такую стоимостную функцию очень трудно надлежаще минимизировать с помощью локальной оптимизации (*снизу*)

Все вышесказанное резко отличается от гипотетической невыпуклой стоимости (не часто встречающейся в нейронных сетях) для наихудшего сценария. Глобальные минимумы такой стоимости лежат значительно ниже ее локальных минимумов. Абстрактное изображение такой функции показано на рис. 13.14, *снизу*. Нет никаких улучшений, которые можно было бы внести в любой метод локальной оптимизации, чтобы обеспечить ему возможность эффективно минимизировать такую стоимостную функцию, а единственная практическая польза заключается в том, чтобы сделать много прогонов, начиная со случайных начальных точек, чтобы увидеть, может ли глобальный минимум быть достигнут каким-то отдельным прогоном.

Пример 13.10. Сравнение первопорядковых оптимизаторов на многослойной нейросетевой модели

В этом примере мы взяли $P = 50\,000$ случайно отобранных точек данных из набора данных MNIST (см. пример 7.10) для выполнения многоклассовой классификации ($C = 10$) с использованием нейронной сети с четырьмя скрытыми слоями, с десятью структурными единицами на слой и активацией tanh (эта конкретная архитектура была выбрана произвольно для целей указанного примера). Сравним эффективность трех первопорядковых оптимизаторов: схемы стандартного градиентного спуска (*см. разд. 3.5*), ее компонентно-нормализованной версии (*см. разд. П1.3.2*) и RMSProp (*см. разд. П1.4*).

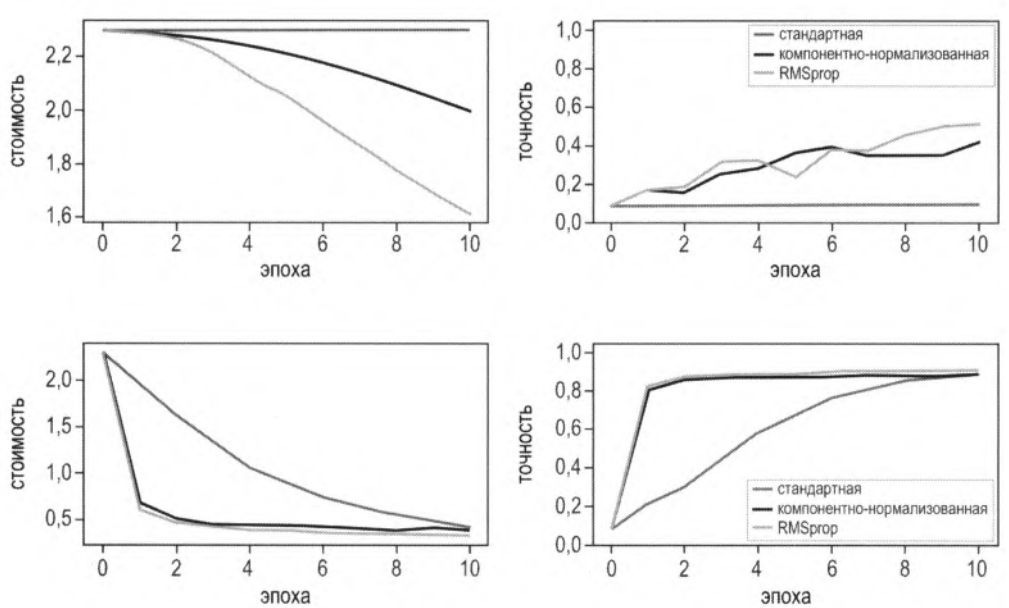


Рис. 13.15. Иллюстрация примера 13.10. Графики истории стоимостной функции (*слева*) и точности (*справа*), сравнивающие эффективность трех оптимизаторов первого порядка в настройке параметров модели четырехслойной нейронной сети над набором данных MNIST с использованием пакетной (*сверху*) и мини-пакетной (*снизу*) версий каждого оптимизатора

Каждый оптимизатор используется как в пакетном, так и в мини-пакетном режимах (пакет имеет размер 200; *см. разд. 7.8*), чтобы минимизировать стоимость многоклассового мягкого максимума над этими данными. Для всех прогонов мы инициализируем в одной и той же начальной точке и в каждом случае используем наибольшее фиксированное значение длины шага формы $\alpha = 10^{\gamma}$ (для целого числа γ), которое произвело схождение.

На рис. 13.15 представлены результаты пакетной (верхний ряд) и мини-пакетной (нижний ряд) версий, демонстрирующие как историю стоимости, так и историю точности, которые измеряют эффективность каждого оптимизатора в течение первых десяти эпох (или полных проходов по набору данных). И в пакетных, и в мини-пакетных прогонах мы видим, как компонентно-нормализованная версия градиентного спуска и RMSProp значительно превосходят стандартный алгоритм с точки зрения скорости схождения.

13.6. Пакетная нормализация

Ранее, в *разд. 9.3*, мы видели, как нормализирование каждого входного признака набора данных значительно помогает ускорить настройку параметров, в особенности с помощью методов оптимизации первого порядка, улучшая форму контуров стоимостной функции (делая их более "округлыми").

С нашей обобщенной линейной моделью

$$\text{model}(\mathbf{x}, \mathbf{w}) = w_0 + x_1 w_1 + ... + x_N w_N \tag{13.32}$$

стандартная нормализация предусматривает нормализирование распределения каждой входной размерности набора данных $\{\mathbf{x}_p\}_{p=1}^{P}$ путем выполнения подстановки

$$x_{p,n} \longleftarrow \frac{x_{p,n} - \mu_n}{\sigma_n} \tag{13.33}$$

для n-й входной размерности, где μ_n и σ_n — соответственно среднее значение и стандартное отклонение вдоль этой размерности.

В этом разделе мы узнаем, как прививка шага стандартной нормализации к *каждому* скрытому слою L-слойной нейросетевой модели

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1^{(L)}(\mathbf{x}) w_1 + ... + f_{U_L}^{(L)}(\mathbf{x}) w_{U_L}, \tag{13.34}$$

где $f_1^{(L)}$, ..., $f_{U_L}^{(L)}$ — L-слойные структурные единицы, описанные в *разд. 13.2*, схожим образом значительно упрощает настройку параметров такой модели.

С помощью этого расширенного приема стандартной нормализации, именуемого *пакетной нормализацией* [62], мы нормализуем не только вход нашей полносвязной сети, но и распределение каждой структурной единицы в каждом скрытом слое сети. Как мы увидим, это обеспечивает еще большее ускорение оптимизации (чем

то, которое предусмотрено стандартной нормализацией только входных данных) для моделей на основе полносвязных нейронных сетей.

13.6.1. Пакетная нормализация структурных единиц с одним скрытым слоем

Для простоты предположим, что мы имеем дело с моделью однослойной нейронной сети. Установив $L = 1$ в уравнении (13.34), мы получаем вот такую модель:

$$\text{model}(\mathbf{x}, \Theta) = w_0 + f_1^{(1)}(\mathbf{x})w_1 + ... + f_{U_L}^{(1)}(\mathbf{x})w_N . \tag{13.35}$$

Теперь мы расширяем базовую схему стандартной нормализации, применив одинаковую концепцию нормализации к каждому "весовому" распределению этой модели. Конечно, здесь *входные признаки* больше не касаются весов окончательной линейной комбинации (например, $w_1, w_2, ..., w_{U_1}$). Вместо этого они касаются весов, внутренних для самих однослойных структурных единиц. Мы сможем увидеть это легче, проанализировав j-ю однослойную единицу в этой сети

$$f_j^{(1)}(\mathbf{x}) = a\left(w_{0,j}^{(1)} + \sum_{n=1}^{N} w_{n,j}^{(1)} x_n \right), \tag{13.36}$$

в которой n-я размерность входа x_n касается только внутреннего веса $w_{n,j}^{(1)}$. Отсюда, в стандартном нормализировании входа мы непосредственно воздействуем на контуры стоимостной функции только вдоль весов, внутренних для однослойных единиц. Для того чтобы оказать воздействие на контуры стоимостной функции относительно весов, внешних для первого скрытого слоя (здесь весов окончательной линейной комбинации), мы, естественно, должны нормализовать *выход* из первого скрытого слоя.

Поместив эти выходные значения в множество и обозначая его через $\left\{ f_j^{(1)}\left(\mathbf{x}_p\right) \right\}_{p=1}^{P}$, мы, естественно, хотели бы стандартно нормализовать его распределение как

$$f_j^{(1)}(\mathbf{x}) \longleftarrow \frac{f_j^{(1)}(\mathbf{x}) - \mu_{f_j^{(1)}}}{\sigma_{f_j^{(1)}}}, \tag{13.37}$$

где среднее значение $\mu_{f_j^{(1)}}$ и стандартное отклонение $\sigma_{f_j^{(1)}}$ задаются как

$$\mu_{f_j^{(1)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(1)}\left(\mathbf{x}_p\right) \text{ и } \sigma_{f_j^{(1)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(f_j^{(1)}\left(\mathbf{x}_p\right) - \mu_{f_j^{(1)}} \right)^2} . \tag{13.38}$$

Важно обратить внимание на то, что, в отличие от наших входных признаков, выход из однослойных структурных единиц (и, следовательно, их распределения) изменяется каждый раз, когда изменяются внутренние параметры нашей модели, например на каждом шаге градиентного спуска. Постоянное изменение этих распределений на жаргоне машинного обучения именуется *внутренним ковариатным*

сдвигом[3], или просто *ковариатным сдвигом* для краткости, и подразумевает, что если мы хотим полностью перенести принцип стандартной нормализации, нам нужно будет нормализовывать выход из первого скрытого слоя на *каждом* шаге настройки параметров. Другими словами, мы должны привить стандартную нормализацию непосредственно в скрытый слой нашей архитектуры.

Далее приведен обобщенный алгоритм для выполнения как раз этого; он является простым расширением рекурсивного алгоритма для однослойных структурных единиц (*см. разд. 13.2.*1).

Рекурсивный алгоритм для пакетных нормализованных однослойных структурных единиц:

1. Выбрать функцию активации $a(\cdot)$.
2. Вычислить линейную комбинацию $v = w_0^{(1)} + \sum_{n=1}^{N} w_n^{(1)} x_n$.
3. Пропустить результат через активацию и сформировать $f^{(1)}(\mathbf{x}) = a(v)$.
4. Стандартно нормализовать $f^{(1)}$ как $f^{(1)}(\mathbf{x}) \longleftarrow \dfrac{f^{(1)}(\mathbf{x}) - \mu_{f^{(1)}}}{\sigma_{f^{(1)}}}$.

Пример 13.11. Визуализирование внутреннего ковариатного сдвига в однослойной сети

В этом примере проиллюстрирован внутренний ковариатный сдвиг в однослойной нейросетевой модели с использованием двух структурных единиц ReLU $f_1^{(1)}$ и $f_2^{(1)}$, применяемых для выполнения двухклассовой классификации игрушечного набора данных, представленного в примере 11.7. Выполним 5000 шагов градиентного спуска, чтобы минимизировать стоимость двухклассового мягкого максимума, используя эту однослойную сеть, где мы стандартно нормализуем входные данные.

На рис. 13.16 показана прогрессия этого градиентного спуска на графиках кортежей $\left\{\left(f_1^{(1)}\left(\mathbf{x}_p\right), f_2^{(1)}\left(\mathbf{x}_p\right)\right)\right\}_{p=1}^{P}$ в трех шагах, сделанных во время прогона. Верхний и нижний ряды соответственно демонстрируют ковариатный сдвиг и полную кривую истории стоимостной функции, где текущий шаг оптимизации отмечен на кривой красной графической точкой (нижний ряд).

Как видно из верхнего ряда на рис. 13.16, распределение этих кортежей резко меняется по мере прогрессии алгоритма градиентного спуска. Мы можем интуитивно понять (из наших предыдущих дискуссий о нормализации входных данных), что такого рода сдвигание распределения отрицательно влияет на скорость, с которой градиентный спуск может надлежаще минимизировать нашу стоимостную функцию.

---

[3] Внутренний ковариатный сдвиг (internal covariate shift) случается, когда происходит изменение во входном распределении в сети. Когда входное распределение изменяется, скрытые слои пытаются освоиться, чтобы приспособиться к новому распределению, что замедляет тренировочный процесс. Эта проблема возникает, когда статистическое распределение входных данных в сети резко отличается от входных данных, которые она видела раньше. — *Прим. пер.*

Далее мы повторяем этот эксперимент, используя те же самые настройки градиентного спуска, но теперь с пакетно-нормализованными однослойными структурными единицами, и строим график результатов схожим образом на рис. 13.17. Обратите внимание, как распределение выходов из активации остается значительно стабильнее по мере прогрессии градиентного спуска.

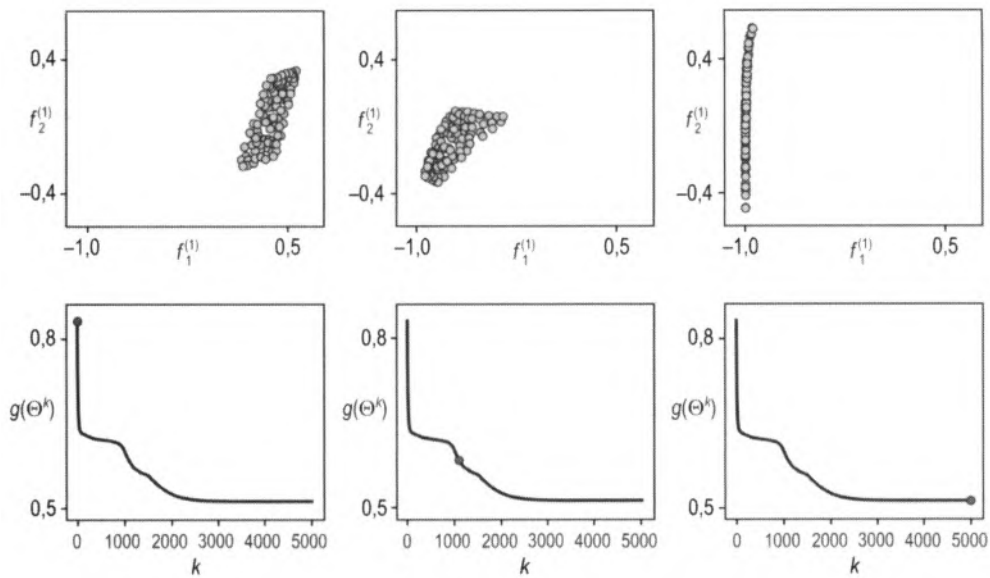


Рис. 13.16. Иллюстрация примера 13.11.
Градиентный спуск с использованием однослойной сети

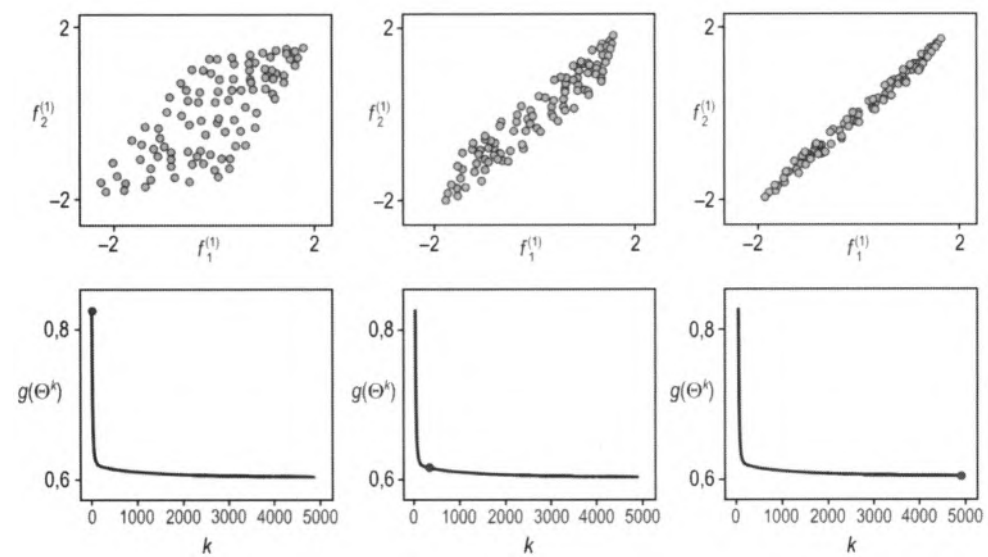


Рис. 13.17. Иллюстрация примера 13.11.
Градиентный спуск с пакетно-нормализованными однослойными структурными единицами

13.6.2. Пакетная нормализация структурных единиц с многочисленными скрытыми слоями

На мгновение предположим, что наша полносвязная нейронная сеть имеет только два скрытых слоя (например, $L = 2$ в уравнении (13.34)), и мы привили шаг стандартной нормализации к первому скрытому слою нашей сети, как описано в предыдущем разделе, благодаря чему теперь наши структурные единицы двух скрытых слоев касаются окончательных линейно-комбинационных весов (например, $w_1, w_2, ..., w_{U_2}$). Для того чтобы сгладить ассоциированную стоимостную функцию по отношению к w_j, мы нормализуем ассоциированное распределение нашей j-й структурной единицы как

$$f_j^{(2)}(\mathbf{x}) \longleftarrow \frac{f_j^{(2)}(\mathbf{x}) - \mu_{f_j^{(2)}}}{\sigma_{f_j^{(2)}}}, \tag{13.39}$$

где среднее значение $\mu_{f_j^{(2)}}$ и стандартное отклонение $\sigma_{f_j^{(2)}}$ определены как

$$\mu_{f_j^{(2)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(2)}(\mathbf{x}_p) \text{ и } \sigma_{f_j^{(2)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(f_j^{(2)}(\mathbf{x}_p) - \mu_{f_j^{(2)}}\right)^2}. \tag{13.40}$$

Как и при изучении однослойного случая в предыдущем разделе, здесь нам нужно будет привить этот шаг и ко второму скрытому слою нашей сети, чтобы всякий раз, когда ее параметры изменяются (например, во время каждого шага локального оптимизатора), распределение этой единицы оставалось нормализованным.

Распространяя эту концепцию на общую модель нейронной сети с L скрытыми слоями из уравнения (13.34), мы нормализуем выход из каждого скрытого слоя сети. Отсюда, в общем случае для L-слоевых единиц, как только мы стандартно нормализовали выход из каждого предшествующего слоя, мы стандартно нормализуем j-ю единицу L-го скрытого слоя:

$$f_j^{(L)}(\mathbf{x}) = a\left(w_{0,j}^{(L)} + \sum_{i=1}^{U_{L-1}} w_{i,j}^{(L)} f_i^{(L-1)}(\mathbf{x})\right) \tag{13.41}$$

посредством подстановки

$$f_j^{(L)}(\mathbf{x}) \longleftarrow \frac{f_j^{(L)}(\mathbf{x}_p) - \mu_{f_j^{(L)}}}{\sigma_{f_j^{(L)}}}, \tag{13.42}$$

где среднее значение $\mu_{f_j^{(L)}}$ и стандартное отклонение $\sigma_{f_j^{(L)}}$ определены как

$$\mu_{f_j^{(L)}} = \frac{1}{P}\sum_{p=1}^{P} f_j^{(L)}(\mathbf{x}_p) \text{ и } \sigma_{f_j^{(L)}} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(f_j^{(L)}(\mathbf{x}_p) - \mu_{f_j^{(L)}}\right)^2}. \tag{13.43}$$

Как и в однослойном случае, мы по-прежнему можем строить каждую пакетно-нормализованную структурную единицу рекурсивно, поскольку от нас требуется

лишь вставить шаг стандартной нормализации в конец каждого слоя, как описано в следующем ниже алгоритме (подобно алгоритму, приведенному для общей L-слоеной структурной единицы в *разд. 13.2.3*).

Рекурсивный алгоритм для пакетно-нормализованных L-слоевых структурных единиц:

1. Выбрать функцию активации $a(\cdot)$.
2. Построить U_{L-1} пакетно-нормализованных $(L-1)$-слоевых структурных единиц $f_i^{(L-1)}(\mathbf{x})$ для $i = 1, 2, ..., U_{L-1}$.
3. Вычислить линейную комбинацию $v = w_0^{(L)} + \sum_{i=1}^{U_{L-1}} w_i^{(L)} f_i^{(L-1)}(\mathbf{x})$.
4. Пропустить результат через активацию и сформировать $f^{(L)}(\mathbf{x}) = a(v)$.
5. Стандартно нормализовать $f^{(L)}$ посредством $f^{(L)}(\mathbf{x}) \longleftarrow \dfrac{f^{(L)}(\mathbf{x}) - \mu_{f^{(L)}}}{\sigma_{f^{(L)}}}$.

При задействовании стохастического или мини-пакетного первопорядкового метода для оптимизации (*см. разд. 7.8*) нормализация архитектуры выполняется точно так же, как подробно описано здесь, на каждом отдельном мини-пакете. Обратите внимание, что на практике формула пакетной нормализации в уравнении (13.42) часто параметризуется как

$$f^{(L)}(\mathbf{x}) \longleftarrow \alpha \frac{f^{(L)}(\mathbf{x}) - \mu_{f^{(L)}}}{\sigma_{f^{(L)}}} + \beta, \tag{13.44}$$

где включение настраиваемых параметров α и β (которые настраиваются вместе с другими параметрами пакетно-нормализуемой сети) обеспечивает более высокую гибкость. Однако даже без этих дополнительных параметров мы можем добиться значительного повышения скорости оптимизации во время настройки полносвязных нейросетевых моделей с задействованием (непараметризованной) пакетной нормализации.

Пример 13.12. Визуализирование внутреннего ковариатного сдвига в многослойной сети

Проиллюстрируем ковариатный сдвиг в четырехслойной сети с двумя структурными единицами на слой, используя активацию ReLU и набор данных, задействованный в примере 13.11. Сравнивним его с ковариатным сдвигом, присутствующим в пакетно-нормализованной версии той же сети. Будем использовать только две структурные единицы на слой, чтобы визуализировать распределение активационных выходов из каждого слоя.

Начиная с ненормализованной версии сети, мы видим, как и в однослойном случае в примере 13.11, значительный ковариатный сдвиг этой сети (см. верхний ряд на рис. 13.18). Здесь показано распределение единиц каждого скрытого слоя, при этом выходные кортежи ℓ-го скрытого слоя $\left\{\left(f_1^{(\ell)}(\mathbf{x}_p), f_2^{(\ell)}(\mathbf{x}_p)\right)\right\}_{p=1}^{P}$ окрашены в голу-

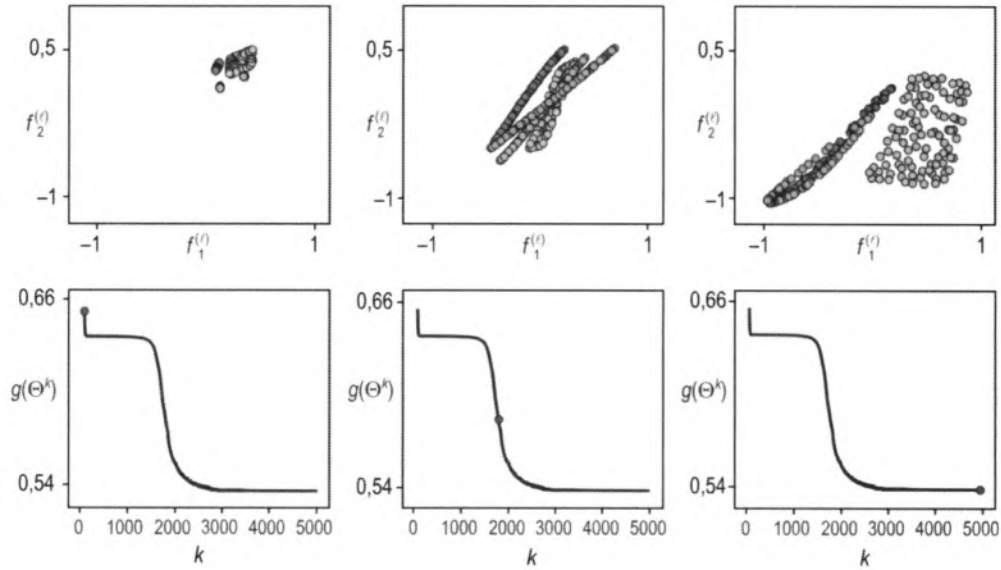


Рис. 13.18. Иллюстрация примера 13.12. Ненормализованная сеть

бой цвет для $\ell = 1$, в сиреневый для $\ell = 2$, в светло-зеленый для $\ell = 3$ и в оранжевый для $\ell = 4$.

Выполнение пакетной нормализации на каждом слое этой сети помогает значительно укротить этот ковариатный сдвиг. На рис. 13.19 показан результат выполнения того же эксперимента с использованием той же инициализации, активации и набора данных, но на этот раз с применением пакетно-нормализованной версии сети. Изучая рисунок слева направо по мере прогрессии градиентного спуска, мы

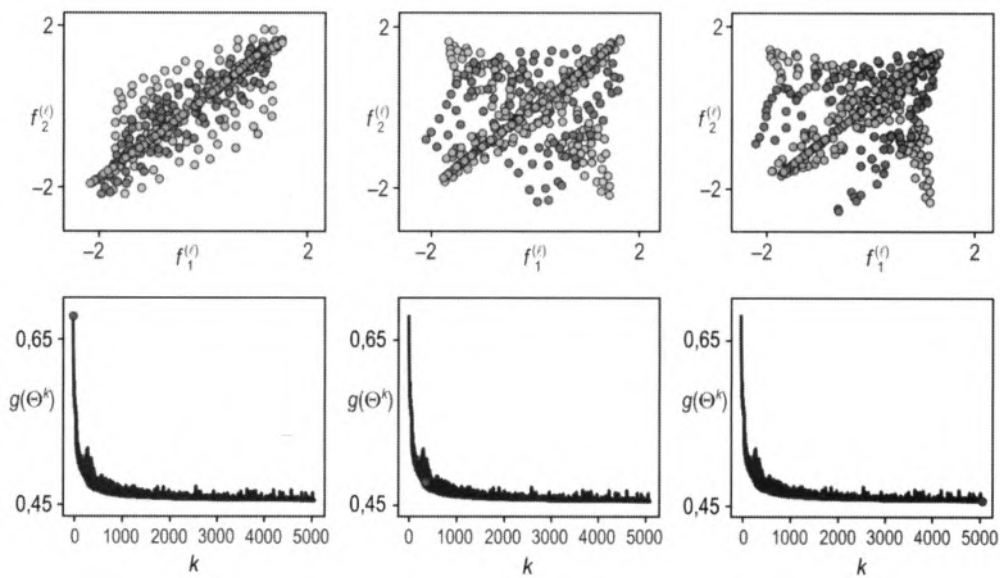


Рис. 13.19. Иллюстрация примера 13.12. Пакетно-нормализованная сеть

видим еще раз, что распределение активационных выходов каждого слоя остается гораздо стабильнее, чем раньше.

Пример 13.13. Стандартная нормализация против пакетной на наборе данных MNIST

Проиллюстрируем выгоду от пакетной нормализации с точки зрения ускорения оптимизации посредством градиентного спуска на наборе данных из $P = 50\,000$ случайно выбранных рукописных цифр из набора данных MNIST (представленного в примере 7.11). На рис. 13.20 показаны история стоимости (*слева*) и классификационная точность (*справа*) десяти эпох градиентного спуска с применением наибольшей длины шага 10^{γ} (для целого числа γ), которая, как мы обнаружили, обеспечивает адекватное схождение. Сравним стандартную и пакетно-нормализованную версии нейронной сети с четырьмя скрытыми слоями и с десятью структурными единицами на слой и активацией ReLU. Мы видим, что как с точки зрения значения стоимостной функции, так и с точки зрения числа ошибочных классифицирований (точности), пакетно-нормализованная версия позволяет минимизировать ее гораздо быстрее посредством градиентного спуска.

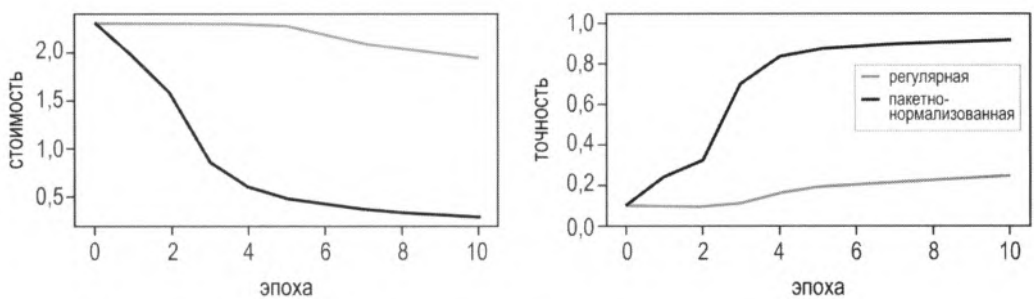


Рис. 13.20. Иллюстрация примера 13.13

13.6.3. Оценивание новых точек данных в пакетно-нормализованных сетях

При задействовании пакетно-нормализованной нейронной сети важно помнить, что мы должны трактовать новые точки данных (неиспользуемые во время тренировки) точно так же, как мы трактуем тренировочные данные. Из этого следует, что окончательные константы нормализации, определенные во время тренировки (т. е. различные средние значения и стандартные отклонения входа, а также те же самые для выхода из каждого скрытого слоя), должны быть сохранены и повторно использованы для того, чтобы можно было надлежаще оценивать новые точки данных. Выражаясь конкретнее, все константы нормализации в пакетно-нормализованной сети должны быть *зафиксированы* на значениях, вычисленных на заключительном шаге тренировки (например, на наилучшем шаге градиентного спуска), во время оценивания новых точек данных.

13.7. Перекрестная валидация посредством ранней остановки

Будучи сильно параметризованной, оптимизация стоимостных функций, ассоциированных с полносвязными нейронными сетями, в особенности задействующими много скрытых слоев, может потребовать значительных вычислений. По этой причине *регуляризация на основе ранней остановки* (как описано в *разд. 11.6.2*), которая предусматривает усвоение параметров для минимизирования валидационной ошибки во время одного прогона оптимизации, является популярным методом перекрестной валидации при задействовании полносвязных многослойных сетей. Понятие ранней остановки также лежит в основе специализированных методов ансамблирования, ориентированных на создание набора нейросетевых моделей для бутстрап-агрегирования (введенного в *разд. 11.9*) с минимальными вычислительными затратами (см., например, [63, 64]).

Пример 13.14. Ранняя остановка и регрессия

Проиллюстрируем процедуру ранней остановки с использованием простого нелинейно-регрессионного набора данных (разделенного на 2/3 для тренировки и 1/3 для валидации) трехслойной нейронной сети с десятью структурными единицами на слой и активацией tanh. На рис. 13.21 показаны три разных шага из одного прогона градиентного спуска (в общей сложности 10 000 шагов), по одному на каждый график, причем результирующая подгонка на каждом шаге показана поверх исходных (первый ряд), тренировочных (второй ряд) и валидационных (третий ряд) данных. Остановка градиентного спуска после выполнения (около) 2000 шагов обеспечивает для этой тренировочно-валидационной разбивки исходных данных мелкогранулированную нелинейную модель для всего набора данных.

Пример 13.15. Ранняя остановка и классификация рукописных цифр

В этом примере мы используем регуляризацию на основе ранней остановки для определения оптимальных настроек двухслойной нейронной сети со 100 структурными единицами на слой и активацией ReLU над набором данных MNIST рукописных цифр, впервые описанном в примере 7.10. Этот многоклассовый ($C = 10$) набор данных состоит из $P = 50\,000$ точек в тренировочном и 10 000 точек в валидационном наборах. С пакетом в размере 500 мы выполняем 100 эпох схемы стандартного мини-пакетного градиентного спуска, в результате чего получаем кривые истории тренировочной (синяя) и валидационной (оранжевая) стоимостных функций (*слева*) и точности (*справа*), показанные на рис. 13.22. Задействовав стоимость многоклассового мягкого максимума, мы нашли, что оптимальная эпоха в такой окружающей обстановке достигла примерно 99-процентной точности на тренировочном наборе и около 96-процентной точности на валидационном наборе. Существует возможность ввести усовершенствования, подобные тем, которые обсуждались в предыдущих разделах этой главы, чтобы повысить точность еще больше. Для сравнения: линейный классификатор — натренированный/валидированный на тех же самых дан-

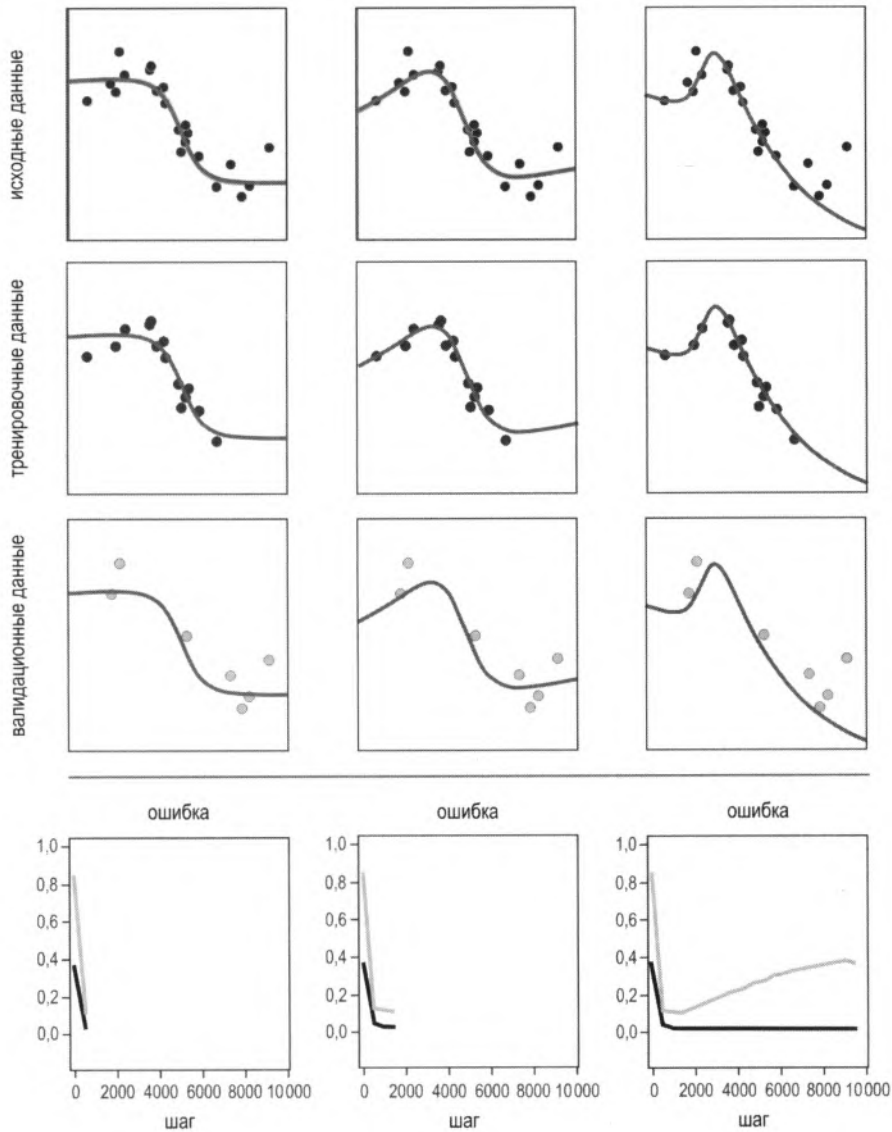


Рис. 13.21. Иллюстрация примера 13.14

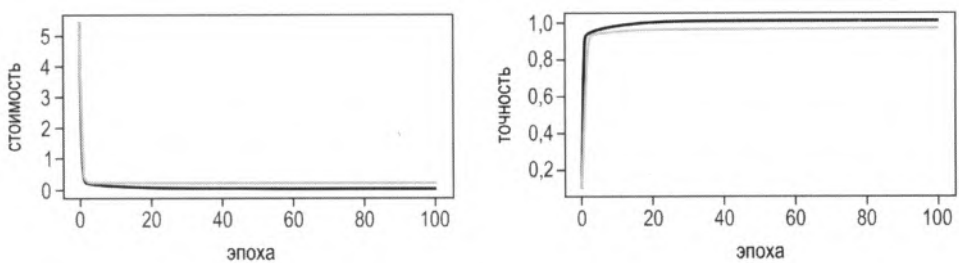


Рис. 13.22. Иллюстрация примера 13.15

ных — достиг соответственно 94- и 92-процентной тренировочной и валидационной точностей.

13.8. Заключение

В этой главе мы описали ряд технических вопросов, касающихся полносвязных нейронных сетей, которые впервые были представлены в *разд. 11.2.3*.

Мы начали с тщательного описания одно- и многослойных нейросетевых архитектур в *разд. 13.2*, затем последовало обсуждение функций активации в *разд. 13.3*, обратного распространения в *разд. 13.4* и невыпуклости стоимостных функций над нейросетевыми моделями в *разд. 13.5*. Затем в *разд. 13.6* была рассмотрена пакетная нормализация — естественное продолжение процедуры стандартной нормализации, описанной в *разд. 9.4*. Наконец, в *разд. 13.7* мы обсудили использование перекрестной валидации на основе регуляризации (и, в частности, ранней остановки) — впервые подробно описанной в *разд. 11.6* — с полносвязными нейросетевыми моделями.

13.9. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:

github.com/jermwatt/machine_learning_refined.

13.1. Двухклассовая классификация с использованием нейронных сетей

Повторить эксперимент по двухклассовой классификации, описанный в примере 13.4, начиная с реализации, приведенной в *разд. 13.2.6*. От вас не требуется воспроизводить результат, показанный в верхнем ряду на рис. 13.9, но вы можете удостовериться в своем результате, проверив, что можно достичь совершенной классификации данных.

13.2. Многоклассовая классификация с использованием нейронных сетей

Повторить эксперимент по многоклассовой классификации, описанный в примере 13.4, с реализацией, представленной в *разд. 13.2.6*. От вас не требуется воспроизводить результат, показанный в нижнем ряду рис. 13.9, но вы можете удостовериться в своем результате, проверив, что можно достичь совершенной классификации данных.

13.3. Число весов для усвоения в нейронной сети

А) Найти суммарное число Q настраиваемых параметров в общей нейронной сети с L скрытыми слоями с точки зрения переменных, выраженных в списке `layer_sizes` в *разд. 13.2.6*.

Б) Основываясь на ответе, полученном в части (*а*), объяснить, как входная размерность N и число точек данных P каждые в отдельности вносят свой вклад

в Q. Чем это отличается от того, что вы видели с ядерными методами в предыдущей главе?

13.4. Нелинейный автокодировщик с использованием нейронных сетей

Повторить эксперимент с автокодировщиком, описанный в примере 13.6, начиная с реализации, представленной в *разд. 13.2.6*. От вас не требуется воспроизводить проекционную карту, показанную на рис. 13.11, *справа снизу*.

13.5. Функция активации maxout

Повторить упражнение 13.4, используя активацию maxout (подробно описанную в примере 13.9).

13.6. Сравнение продвинутых первопорядковых оптимизаторов I

Повторить первый набор экспериментов, описанный в примере 13.10, и построить графики, подобные тем, что приведены в верхнем ряду на рис. 13.15. Ваши графики могут выглядеть не совсем так, как показано на этом рисунке (но должны быть похожи).

13.7. Сравнение продвинутых первопорядковых оптимизаторов II

Повторить второй набор экспериментов, описанный в примере 13.10, и построить графики, подобные тем, что приведены в нижнем ряду на рис. 13.15. Ваши графики могут выглядеть не совсем так, как показано на этом рисунке (должны быть похожи).

13.8. Пакетная нормализация

Повторить эксперимент, описанный в примере 13.13, и построить графики, подобные показанным на рис. 13.20. Ваши графики могут выглядеть не совсем так, как представлено на этом рисунке (но они должны выглядеть похоже).

13.9. Перекрестная валидация на основе ранней остановки

Повторить эксперимент, описанный в примере 13.14. От вас не требуется воспроизводить все графики, показанные на рис. 13.21. Однако вы должны построить график подгонки, обеспечиваемой весами, ассоциированными с минимальной валидационной ошибкой, поверх всего набора данных.

13.10. Распознавание рукописных цифр с использованием нейронных сетей

Повторить эксперимент, описанный в примере 13.15, и создать графики истории стоимости/точности, подобные тем, что представлены на рис. 13.22. Вы можете не воспроизводить в точности то, что сообщается на основе вашей конкретной реализации. Однако вы должны достичь похожих результатов, как показано в примере 13.15.

14 Деревья решений

14.1. Введение

В этой главе мы подробно остановимся на нашем обсуждении *деревьев решений*, впервые представленных в *разд. 11.2.3*, которые пользуются огромной популярностью благодаря своей высокой эффективности, в особенности при работе со структурированными данными (см., например, обсуждение в *разд. 11.8*). В этой главе мы разведаем технические экстравагантности, связанные с деревьями решений, опишем так называемые *регрессионные* и *классификационные деревья* и объясним их конкретное использование с перекрестной валидацией на основе бустирования и бутстрап-агрегированных ансамблей (впервые представленных соответственно в *разд. 11.5* и *11.9*), где на жаргоне машинного обучения они называются *бустированием градиента* и *случайными лесами*.

14.2. От пней до глубоких деревьев

В *разд. 11.2.3* мы представили простейший образчик дерева решений — пень. В этом разделе мы обсудим, как, используя простые пни, можно определять более общие и сложные древесные универсальные аппроксиматоры.

14.2.1. Пень

Самый простой древесный универсальный аппроксиматор — *пень* — представляет собой простую шаговую функцию формы

$$f(x)=\begin{cases} v_1, & \text{если } x \le s; \\ v_2, & \text{если } x > s \end{cases} \tag{14.1}$$

с тремя настраиваемыми параметрами: двухшаговыми уровнями или *листовыми* параметрами, обозначаемыми через v_1 и v_2 (значения которых задаются независимо друг от друга), и параметром точки ветвления s, определяющим границу между двумя уровнями. Этот простой пень изображен на рис. 14.1, *слева сверху*. *Справа сверху* на этом рисунке показано еще одно графовое представление обобщенного пня из уравнения (14.1), которое помогает объяснить конкретную номенклатуру (т. е. деревья, листья и т. д.), используемую в контексте древесных аппроксимато-

ров. Представленный таким образом пень можно рассматривать как бинарное дерево *глубины один*, при этом $f(x)$ — это корневой узел, а v_1 и v_2 — листовые узлы.

Пень, определенный в уравнении (14.1), принимает одномерные (т. е. скалярные) входные данные. Когда входные данные в целом являются N-мерными, пень разрезается вдоль одной размерности (или оси координат). Например, при определении вдоль некоторой n-й размерности пень, принимающий N-мерный вход **x**, определяется как

$$f(\mathbf{x}) = \begin{cases} v_1, & \text{если } x_n \le s; \\ v_2, & \text{если } x_n > s, \end{cases} \tag{14.2}$$

где x_n обозначает n-ю размерность **x**.

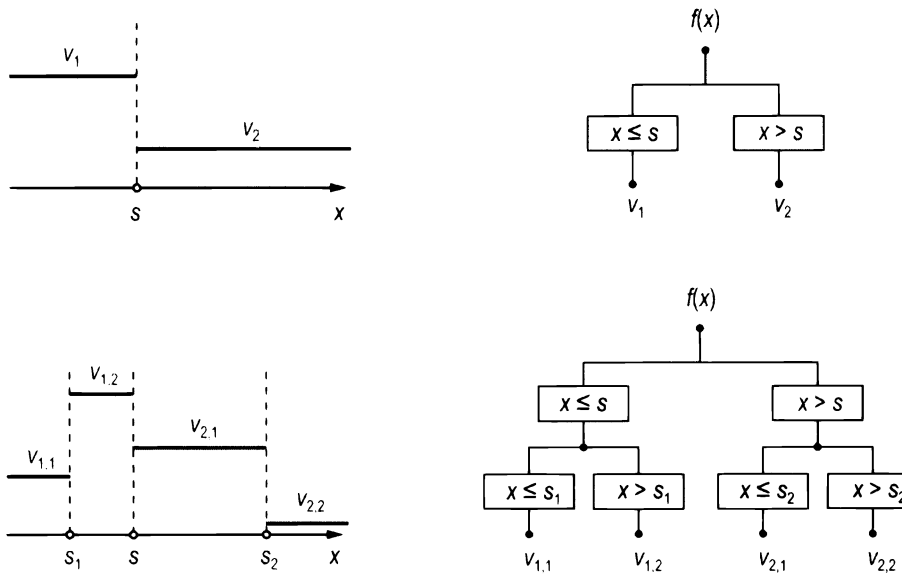


Рис. 14.1. Простой пень, определенный в уравнении (14.1) (*слева сверху*). Графическая иллюстрация пневой функции в виде двоичного дерева (*справа сверху*). Дерево глубины два, образованное рекурсией на каждом листе пня, заменяя его новым пнем (*слева снизу*). Графическая иллюстрация дерева глубины два (*справа снизу*)

14.2.2. Создание глубоких деревьев посредством рекурсии

Мы можем строить более глубокие деревья рекурсивно, применяя ту же самую концепцию, которая использовалась для пристройки пня к каждому из его листьев, т. е. путем разбивки каждого листа на два подлиста. Эта рекурсия приводит к дереву глубиной два с тремя точками ветвления и четырьмя отдельными листьями, как показано в нижнем ряду рис. 14.1. Дерево *глубиной два* имеет значительно большую емкость (*см. разд. 11.2*), чем пень, т. к. расположение точек ветвления и листовые значения могут быть установлены самыми разными способами, как видно из верхнего ряда на рис. 14.2.

Эта рекурсивная идея затем может постоянно применяться к каждому листу дерева глубиной два, чтобы создать дерево глубиной три и т. д. Чем глубже дерево становится, тем бóльшую емкость оно приобретает, причем каждая структурная единица способна принимать более широкое разнообразие разных контуров, как видно из нижнего ряда на рис. 14.2. Действительно, это отражается в том факте, что деревья становятся экспоненциально более параметризованными, чем глубже они делаются: можно легко показать, что дерево общей глубины D (со скалярным входом) будет иметь $2^D - 1$ точек ветвления и 2^D листьев и, таким образом, в общей сложности $2^{D+1} - 1$ настраиваемых параметров. На жаргоне машинного обучения эта рекурсивная процедура часто называется *выращиванием дерева*.

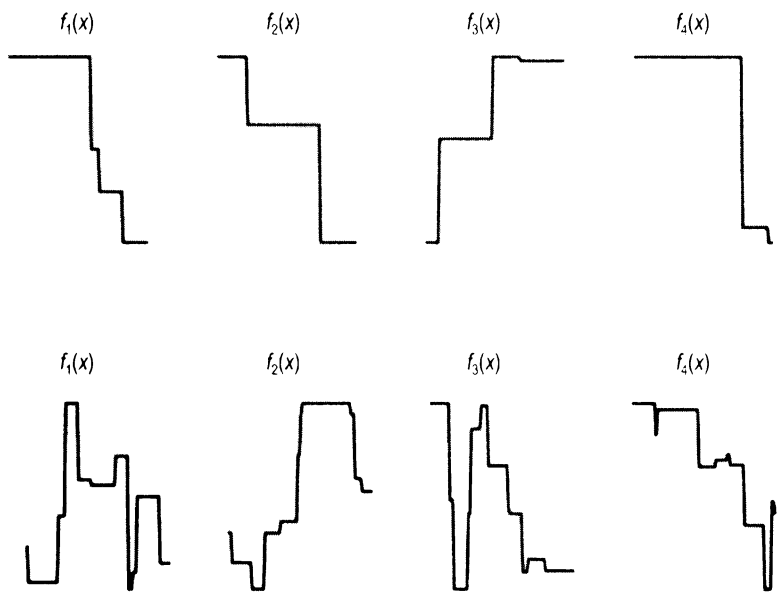


Рис. 14.2. Четыре экземпляра дерева глубиной $D = 2$ (*верхний ряд*) и дерева глубиной $D = 10$ (*нижний ряд*), где в каждом случае все параметры (т. е. точки ветвления и листовые значения) задаются случайным образом. Последнее явно способно генерировать более широкую полосу контуров при разных настройках своих параметров и, таким образом, обладает большей емкостью. Обратите внимание, что листья соединены вертикальными линиями, чтобы придать каждому экземпляру дерева непрерывный внешний вид, что делается только для целей визуализации

Важно отметить, что, *в отличие* от фиксированно-контурных и нейросетевых универсальных аппроксиматоров, древесные структурные единицы определяются *локально*. Из этого вытекает, что при настройке одного параметра многочленной или нейросетевой единицы он может глобально влиять на контур функции над всем входным пространством. Однако, когда мы разветвляем любой лист дерева, мы влияем на контур дерева в этом листе только локально. Вот почему древесные универсальные аппроксиматоры иногда называются *локально-функциональными аппроксиматорами*.

14.2.3. Создание глубоких деревьев путем сложения

Более глубокие, более гибкие деревья также можно построить путем *добавления* более мелких деревьев. Например, рис. 14.3 демонстрирует то, как можно создать дерево глубиной два, сложив три дерева глубиной один (т. е. три пня). Легко показать, что в целом сложение $2^D - 1$ пней (со скалярным входом) будет создавать дерево глубиной D (при условии, что пни не имеют общих точек ветвления).

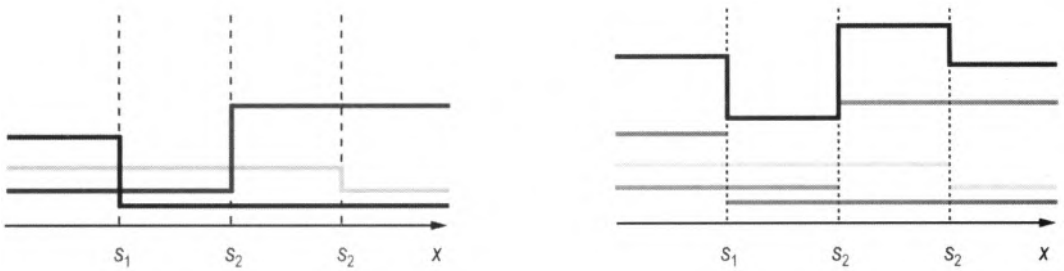


Рис. 14.3. Три пня, каждый из которых изображен отдельным цветом (*слева*). Для удобства визуализации мы соединили листья вертикальными линиями, чтобы придать каждому экземпляру пня непрерывный вид. Дерево глубины два (черным цветом), созданное путем сложения трех пней, показанных на левой панели (*справа*)

14.2.4. Интерпретируемость человеком

Учитывая их особенно простую структуру, мелкие древесные структурные единицы (такие как деревья глубиной один и глубиной два, показанные на рис. 14.1) часто легко интерпретируются людьми по сравнению с их фиксированно-контурными и нейросетевыми аналогами. Однако этот признак древесных структурных единиц быстро рассеивается по мере увеличения глубины дерева (см., например, деревья глубиной $D = 10$, изображенные в нижнем ряду рис. 14.2), а также когда деревья совмещаются или *ансамблируются*.

14.3. Регрессионные деревья

В этом разделе мы обсудим применение общих древесных универсальных аппроксиматоров для задачи регрессии, часто именуемых *регрессионными деревьями*. В отличие от фиксированно-контурных или нейросетевых универсальных аппроксиматоров, стоимостные функции, наполненные древесными структурными единицами, создают сильно невыпуклые, лестницеподобные функции, которые не получается легко минимизировать каким-либо методом локальной оптимизации. Для того чтобы понять, почему это так, давайте возьмем простой регрессионный набор данных, показанный на рис. 14.4, *слева сверху*, и попробуем вписать в него нелинейный регрессор, используя модель, состоящую из одного-единственного пня, путем минимизирования соответствующей стоимостной функции над этой моделью (например, стоимости наименьших квадратов).

14.3.1. Определение оптимальной точки ветвления при фиксированных листовых значениях

Подгонка модели на основе одного-единственного пня к нашему набору данных требует настройки трех ее параметров: местоположения точки ветвления пня, а также двух листовых значений. Для того чтобы упростить задачу, зафиксируем два листовых параметра, связанных с нашей моделью, в двух произвольных значениях, в результате чего остается оптимально настроить только параметр s точки ветвления, и, следовательно, теперь мы можем визуализировать *одномерную* стоимостную функцию $g(s)$ наименьших квадратов, включающую только параметр точки ветвления. На рис. 14.4, *справа сверху* показаны три экземпляра пней, окрашенных в красный, зеленый и синий цвета, с разными, но фиксированными листовыми значениями. Теперь мы берем каждый пень и перемещаем его по набору данных горизонтально, пробуя для каждого все возможные точки ветвления во входном пространстве набора данных. Три стоимостных функции наименьших квадратов, полученные в результате этого упражнения, представлены на среднем графике и окрашены в цвета, совпадающие с соответствующими им пнями, показанными на графике справа сверху. Каждая одномерная стоимость, как мы видим, выглядит как лестница, состоящая из большого числа совершенно плоских участков. Эти проблемные плоские участки являются прямым следствием контура нашей нелинейности (т. е. пня). Напомним, что подобное поведение мы наблюдали во время работы с шаговыми функциями в контексте логистической регрессии еще в *разд. 6.2* и *6.3*. Существование таких плоских участков крайне нежелательно, поскольку ни один алгоритм локальной оптимизации не может эффективно ими управлять.

Вместе с тем обратите внимание на то, что при зафиксированных значениях листьев пня соответствующая стоимость наименьших квадратов остается постоянной для всех значений точек ветвления *между* следующими друг за другом входными значениями. Другими словами, все три стоимостные функции на рис. 14.4, *в центре* принимают контур в виде лестницы, в котором их плоские ступенчатые области расположены в одних и тех же местах: участки между следующими друг за другом входными значениями.

Этот факт имеет практические последствия: хотя мы не можем правильно настроить параметр точки ветвления с помощью локальной оптимизации (из-за лестничной формы стоимостной функции над этим параметром), мы можем тем не менее найти его, просто протестировав *одно-единственное значение* (например, среднюю точку) во всех и каждой плоской области стоимости, т. к. все точки ветвления там производят одинаковое качество регрессии. Эта коллекция вычисленных средних точек для каждого из наших трех примеров пней показана на рис. 14.4, *снизу*.

14.3.2. Определение оптимальных листовых значений при фиксированной точке ветвления

В отличие от задачи определения оптимальной точки ветвления пня с фиксированными листовыми значениями, определить оптимальные листовые значения для пня с фиксированной точкой ветвления чрезвычайно просто. Поскольку листья пня

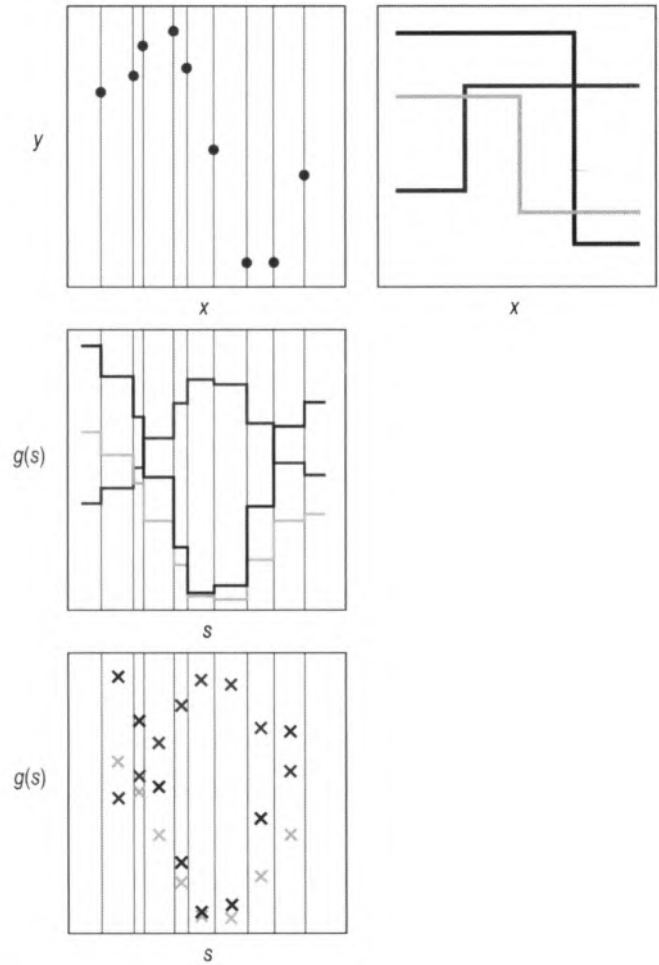


Рис. 14.4. Прототипический нелинейно-регрессионный данных набор (*слева сверху*). Три пня с фиксированными листовыми значениями (точка ветвления которых может варьироваться) (*справа сверху*). Каждый экземпляр пня скользит горизонтально по входным данным, изменяя свое значение точки ветвления, создавая три соответствующие лестницеподобные стоимости наименьших квадратов (*в центре*). Каждая стоимость в средней панели является постоянной между следующими один за другим входами, из чего вытекает, что нам нужно протестировать только одну точку ветвления в расчете на плоский участок, например среднюю точку, как показано на этой панели (*снизу*)

имеют *постоянное значение* и мы хотим, чтобы они оба были установлены так, чтобы вместе наш пень представлял данные как можно лучше, интуитивно имеет смысл установить значение каждого листа просто равным *среднему выходу* этих точек, которые он будет представлять. Этот выбор показан красным цветом на рис. 14.5, *справа* для нашего игрушечного регрессионного набора данных, где для заданной точки ветвления (проиллюстрированной вертикальной синей штриховой линией на левом графике) значение листа *слева* устанавливается равным среднему выходу тех точек, которые лежат слева от точки ветвления, а значение листа *справа*

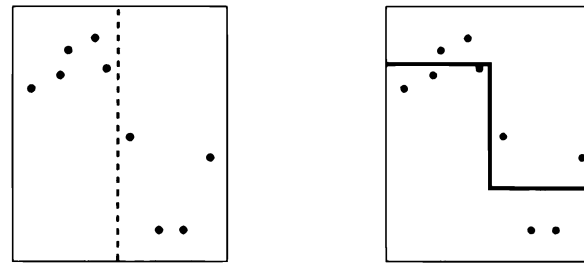


Рис. 14.5. Тот же самый регрессионный набор данных, показанный на рис. 14.4, вместе с фиксированной точкой ветвления, представленной вертикальной штриховой синей линией, которая делит входное пространство данных на два подпространства, лежащие слева и справа от этой линии (*слева*). Пень с оптимально заданными листовыми значениями, определяемыми как среднее значение выхода из всех точек данных слева и справа от точки ветвления (*справа*)

устанавливается равным среднему выходу тех точек, которые находятся *справа* от точки ветвления.

Этот интуитивный выбор листового значения может быть полностью оправдан посредством *условия оптимальности первого порядка*, введенного в *разд. 3.2*. Для того чтобы убедиться в этом, давайте сначала формализуем общий сценарий, который мы расследуем. В контексте регрессии наша фиксированная точка ветвления s определяется вдоль n-й входной размерности регрессионного набора данных — обозначенная через $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ — и разбивает данные на две секции. Мы можем отследить эти два подмножества наших данных с помощью индексных множеств Ω_L и Ω_R, которые обозначают входные/выходные пары нашего набора данных, лежащие по обе стороны от ветвления "слева" и "справа" от него, выраженные формально как

$$\Omega_L = \{p \mid x_{p,n} \le s\} \text{ и } \Omega_R = \{p \mid x_{p,n} > s\}. \tag{14.3}$$

Общий пень, использующий эту точку ветвления (отражая уравнение (14.2)), тогда можно записать как

$$f(\mathbf{x}) = \begin{cases} v_L, & \text{если } x_n \le s; \\ v_R, & \text{если } x_n > s, \end{cases} \tag{14.4}$$

где x_n — n-я размерность входа $\mathbf{x}$; v_L и v_R — листовые значения, которые мы определим.

Для получения оптимальных значений v_L и v_R можно минимизировать две *одномерные* стоимости наименьших квадратов, определенные над точками, принадлежащими индексному множеству соответственно Ω_L и Ω_R, как

$$g(v_L) = \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L}(v_L - y_p)^2 \text{ и } g(v_R) = \frac{1}{|\Omega_R|}\sum_{p\in\Omega_R}(v_R - y_p)^2, \tag{14.5}$$

где $|\Omega_L|$ и $|\Omega_R|$ — число точек, принадлежащих соответственно индексным множествам Ω_L и Ω_R.

Каждая из этих стоимостных функций исключительно проста. Приравняв производную каждой из них нулю (по отношению к соответствующему ей листовому значению) и вычислив, получим оптимальные листовые значения соответственно $v_L^\star$ и $v_R^\star$ в виде:

$$v_L^\star = \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L} y_p \quad \text{и} \quad v_R^\star = \frac{1}{|\Omega_R|}\sum_{p\in\Omega_R} y_p \,. \tag{14.6}$$

14.3.3. Оптимизация регрессионных пней

Совмещение двух рассмотренных ранее идей обеспечивает разумный обходной путь для настройки всех трех параметров пня для целей регрессии (в качестве альтернативы настройке всех трех вместе посредством локальной оптимизации, что мы не можем сделать). То есть сначала мы создаем набор кандидатных значений точек ветвления, записывая каждую среднюю точку между нашими входными данными, вдоль каждой ее входной размерности. Затем для каждой кандидатной точки ветвления мы оптимально определяем листовые значения пня, устанавливая их равными среднему значению выходных тренировочных данных слева и справа от точки ветвления, и вычисляем ее стоимостное значение (наименьших квадратов). Сделав это для всех кандидатных точек ветвления, мы отыскиваем лучший пень (с оптимальными значениями точки ветвления и листовыми значениями), как тот, который обеспечивает наименьшую стоимость.

Пример 14.1. Подгонка параметров простого регрессионного дерева

Впишем модель на основе одного пня в игрушечный регрессионный набор данных, показанный на рис. 14.4 и 14.5, иллюстрируя весь диапазон кандидатных пней, точки ветвления которых формируются путем взятия средней точки между каждой следующей друг за другом парой входов, и соответствующие листовые значения которых устанавливаются равными среднему значению выходного значения по обе стороны каждого ветвления. Приводя графики рис. 14.6 именно в таком порядке — *слева направо* и *сверху вниз*, мы иллюстрируем весь диапазон кандидатных пней для этого набора данных, анализируя слева направо все входы набора данных. В верхнем ряду показаны кандидатные пни со значениями стоимости наименьших квадратов, связанными с конкретным пнем, а также предшествовавшие ему, нанесенные на графике под ним. После того как все кандидаты были протестированы (как показано в нижнем ряду рис. 14.6), конкретный пень, обеспечивающий наименьшую возможную стоимость (здесь пятый пень в левой нижней панели), считается оптимальным.

В общем случае для набора данных из P точек, каждая из которых имеет входную размерность N, в общей сложности существует $N(P-1)$ точек ветвления для выбора из всего входного пространства задачи. Когда вычисление становится первостепен-

ной заботой (главным образом с очень большими P и/или N), можно использовать стратегии отбора, выводимые из этой базовой схемы, включая тестирование точек ветвления только вдоль случайной коллекции всех входных размерностей, тестирование более грубой коллекции точек ветвления и т. д.

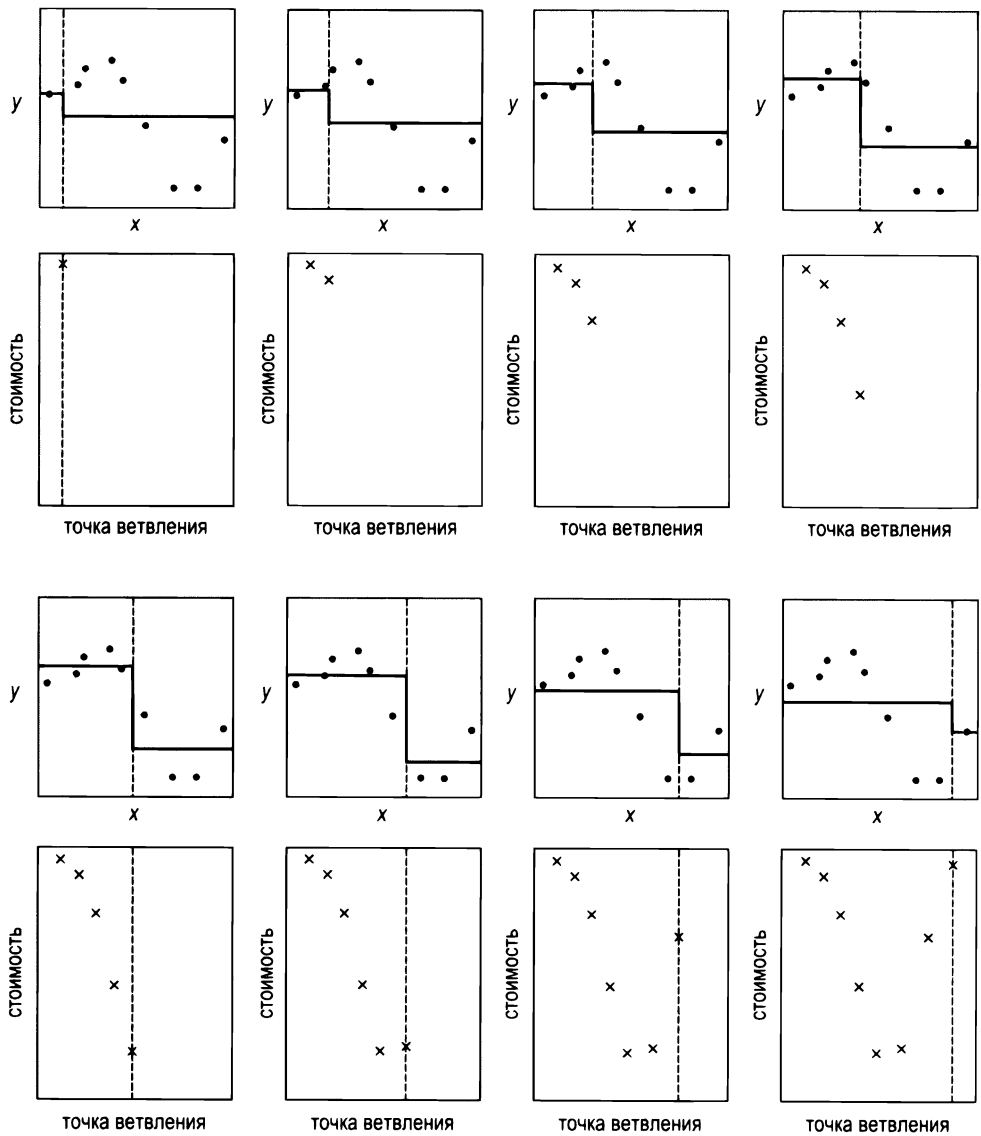


Рис. 14.6. Иллюстрация примера 14.1

14.3.4. Более глубокие регрессионные деревья

Для того чтобы вписать дерево глубиной два в регрессионный набор данных, мы сначала вписываем пень (см. предыдущий раздел), а затем рекурсивно применяем

ту же самую идею на каждом листе пня. Другими словами, мы можем думать о первом подогнанном пне как о делении нашего исходного набора данных на два непересекающихся подмножества, по одному принадлежащему каждому листу. Рассуждая рекурсивно, мы можем затем вписать пень в каждый из этих подмножеств точно так же, как мы вписывали пень в исходный набор данных, разбив каждый лист нашего исходного пня на два и создав дерево глубиной два. Мы можем пойти дальше и повторить этот процесс, разветвляя каждый лист нашего дерева глубиной два, чтобы создать дерево глубиной три, и т. д.

Обратите внимание, что в процессе *выращивания* дерева существуют некоторые условия, при которых мы *не* должны разветвлять лист. Например, нет никакой причины разветвлять лист, содержащий только одну точку данных, или тот, где точки данных, содержащиеся во входном пространстве листа, имеют точно такое же выходное значение (т. к. в обоих случаях наш текущий лист представляет содержащиеся в нем данные совершенно). Оба эти практических условия часто обобщаются во время практического имплементирования рекурсивных древесных регрессоров. Например, прогрессия на листе может быть остановлена, если он содержит меньше заданного числа точек (в отличие от одиночной точки). Отсюда, на практике регрессионное дерево (со скалярным входом) глубины D не может заканчиваться ровно 2^D листьями. Вместо этого некоторые ветви, идущие от корня дерева, могут остановиться раньше, чем другие, причем некоторые ветви дерева, возможно, вырастут до некоторой глубины. Поэтому при применении двоичных деревьев к регрессии (и, как мы увидим, к классификации) мы говорим о деревьях, что они имеют *максимальную глубину*, т. е. самую большую глубину, до которой может вырасти ветвь дерева.

Пример 14.2. Выращивание регрессионного дерева максимальной глубины

Рекурсивная процедура выращивания глубокого регрессионного дерева показана на рис. 14.7. Мы начинаем (слева) с подгонки пня к исходному набору данных. По мере того как мы движемся слева направо, рекурсия продолжается тем, что каждый лист предыдущего дерева разветвляется, создавая следующее, более глубокое дерево. Как видно на самом правом рисунке, дерево с максимальной глубиной четыре способно представить тренировочные данные совершенно.

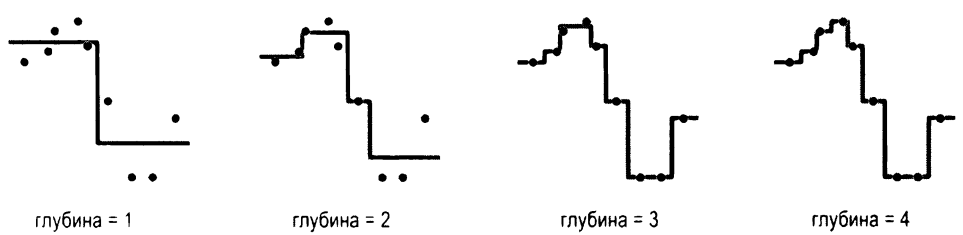


Рис. 14.7. Иллюстрация примера 14.2

14.4. Классификационные деревья

В этом разделе мы обсудим применение древесных универсальных аппроксиматоров к задаче классификации, часто именуемых *классификационными деревьями*. К счастью, практически все, что мы ранее видели относительно регрессионных деревьев в предыдущем разделе, непосредственно связано с задачей классификации. Однако, поскольку классификационные данные имеют *дискретный* выход, для определения надлежащих листовых значений используются разные подходы. Как и в предыдущем разделе, мы начнем с обсуждения надлежащего построения пня, задействовав игрушечный набор данных для иллюстрации ключевых концепций, и будем применять эту идею рекурсивно для создания более глубоких деревьев.

14.4.1. Определение оптимальной точки ветвления при фиксированных листовых значениях

Представьте себе, что мы имеем дело с классификационным набором данных, например с набором, показанным на рис. 14.8, *слева*, и предположим, что мы стремимся правильно вписать пень в эти данные. Если мы попытаемся установить *точку ветвления* нашего пня посредством локальной оптимизации, то мы столкнемся точно с той же проблемой, с которой имели дело в *разд. 14.3.1* в случае регрессии. То есть, не только любая соответствующая классификационная стоимостная функция не будет выпуклой, но и будет состоять из абсолютно плоских, лестницеподобных участков, по которым не сможет эффективно перемещаться ни один алгоритм локальной оптимизации. Отсюда, определение оптимального значения точки ветвления в случае классификации естественным образом должно привести к тому же подходу, который мы видели в случае с регрессией в предыдущем разделе: мы должны протестировать массив кандидатных точек ветвления, чтобы определить, какая из них работает лучше всего. Опять же, ради тех же практических целей, которые мы видели в случае с регрессией, мы можем просто протестировать средние точки между каждой следующей одна за другой парой входных значений вдоль каждой ее входной размерности (или подмножество этих точек, если их число становится непомерно большим).

14.4.2. Определение оптимальных листовых значений при фиксированной точке ветвления

В *разд. 14.3.2* мы видели в случае регрессии, что листовые значения для модели на основе одного пня могут быть интуитивно установлены равными *среднему* выходу тех точек, которые принадлежат каждому листу пня. Мы подкрепили этот интуитивный выбор, показав, что эти параметры являются именно тем, что мы находим, решая условие первого порядка для набора надлежаще определенных стоимостных функций наименьших квадратов. Здесь, в классификационном сценарии, давайте следовать этой логике в обратном порядке и начнем с оптимизирования соответствующей стоимостной функции (например, стоимости персептрона или перекрест-

ной энтропии/мягкого максимума). А затем пойдем путем представления интуитивного выбора, основанного на другой статистике выходных данных — моде.

Предположим, что перед нами стоит задача классификации двухклассового набора данных $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ со значениями меток $y_p \in \{-1, +1\}$ и точка ветвления в нашем пне фиксирована. Мы определяем индексные множества Ω_L и Ω_R, как в уравнении (14.3), чтобы обозначить индексы всех точек, лежащих "слева" и "справа" от нашей разветвленной точки. Для определения оптимальных значений v_L и v_R можно минимизировать две *одномерные* классификационные стоимости (например, мягкого максимума), определенные над точками, принадлежащими индексному множеству соответственно Ω_L или Ω_R, как

$$g(v_L) = \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L}\log\left(1+e^{-y_p v_L}\right) \text{ и } g(v_R) = \frac{1}{|\Omega_R|}\sum_{p\in\Omega_R}\log\left(1+e^{-y_p v_R}\right), \qquad (14.7)$$

где опять же, как и в уравнении (14.5), $|\Omega_L|$ и $|\Omega_R|$ обозначают число точек, принадлежащих к индексным множествам Ω_L и Ω_R соответственно. Можно также взвесить слагаемые таких стоимостных функций (как подробно описано в *разд. 6.9.3*) для того, чтобы лучше справляться с потенциальным *классовым дисбалансом* в листьях.

В любом случае, в отличие от аналогичной пары стоимостей наименьших квадратов в уравнении (14.5), здесь мы не можем вычислить соответствующие условия первого порядка в замкнутой форме и должны полагаться на методы локальной оптимизации. Однако по причине простоты каждой задачи такие оптимизации особенно легко решаются итеративно. Действительно, часто стоимости такого рода приближенно минимизируются путем применения всего *лишь одного* шага метода Ньютона. При этом существенно можно минимизировать стоимости, сохраняя при этом низкие накладные расходы на вычисление[1] и предотвращая потенциальные численные проблемы, связанные со стоимостью мягкого максимума и методом Ньютона (введенные в контексте линейной двухклассовой классификации в *разд. 6.6*).

Обратите внимание, что, как и при любом другом подходе к классификации, после того как были определены надлежащие листовые значения, для выполнения валид-

[1] Как подробно описано в *разд. 4.3*, один шаг в методе Ньютона предусматривает минимизирование наилучшей квадратичной аппроксимации стоимостной функции, обеспечиваемой ее разложением в ряд Тейлора второго порядка, и для общей одновходной стоимости $g(w)$ приводит к простому обновлению в форме

$$w^\star = w^0 - \frac{\frac{d}{dw}g\left(w^0\right)}{\frac{d^2}{dw^2}g\left(w^0\right)+\lambda}, \qquad (14.8)$$

где w^0 — некоторая начальная точка; $\lambda \geq 0$ — регуляризационный параметр, используемый для предотвращения возможного деления на ноль (как описано в *разд. 4.3.3*); $w^\star$ — оптимальное обновление.

ных предсказаний выходные данные классификационного пня должны быть пропущены через соответствующий дискретизатор, например знаковую функцию в случае двухклассовой классификации, с выбором значений меток ±1 (*см. разд. 6.8.1*).

В качестве альтернативы подходу на основе *стоимостной функции*, подробно описанному до этого места, можно также выбрать оптимальные листовые значения, основываясь на простой статистике выходных данных. Поскольку выходной результат классификационных данных является дискретным, мы естественным образом избегаем использования *среднего* значения в качестве нашей предпочтительной статистики (как мы делали с регрессией), и вместо этого склоняемся к использованию *моды* (т. е. самой популярной выходной метки), также именуемой голосом большинства, или *мажоритарным голосом*. Использование моды будет держать наши листовые значения ограниченными дискретными метками наших данных, обеспечивая более подходящие пни.

Однако стандартная мода, как статистическая величина, может привести к нежелательным последствиям, пример которых показан для простого набора данных двух классов на рис. 14.8. Для этого конкретного набора данных из-за распределения мажоритарного класса (здесь это точки со значением метки $y_p = -1$) статистическая мода *по обе* стороны каждого пня всегда будет равна –1, и таким образом все пни будут совершенно плоскими и идентичными. Отсутствие разнообразия пней в этом простом примере не делает невалидным использование стандартной моды. Однако это подчеркивает ее неэффективность в том, что для улавливания нелинейности такого игрушечного набора данных необходимы более глубокие деревья (создание которых обходится дороже).

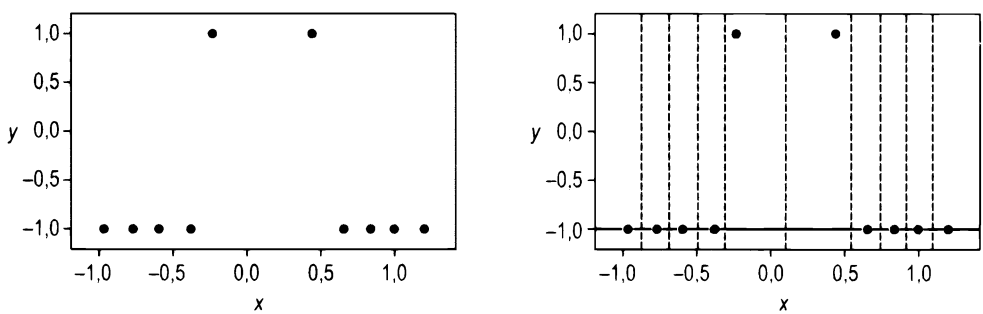


Рис. 14.8. Простой двухклассовый классификационный набор данных (*слева*). Для каждой из девяти точек ветвления, обозначенных вертикальными пунктирными синими линиями, установка листовых значений на основе *стандартной* моды приводит к совершенно плоскому пню, показанному здесь красным цветом (*справа*)

В целях компенсирования классового дисбаланса, подобного показанному здесь, мы можем, по полной аналогии с концепцией весовых стоимостных функций, задействуемых для более качественного улаживания классового дисбаланса (*см. разд. 6.8.1*), выбрать листовые значения, основанные на *сбалансированной моде* или *сбалансированном голосе большинства*. Для того чтобы вычислить стандартную моду на одном листе пня, мы просто подсчитываем число точек, принад-

лежащих каждому классу в листе, и определяем моду, выбирая класс, связанный с наибольшим числом. Для того чтобы вычислить *сбалансированную* моду на одном листе пня, мы сначала подсчитываем число точек, принадлежащих каждому классу на листе, а затем взвешиваем каждое количество инверсно на основе числа точек в каждом классе, принадлежащих обоим листьям пня, определяя сбалансированную моду путем выбора наибольшего результирующего взвешенного числа. Для общего многоклассового набора данных с C классами взвешенное число для c-го класса на одном листе может быть записано как

$$\frac{\text{число баллов из класса } c \text{ в листе}}{\text{число баллов из класса } c \text{ в обоих листьях пня}}. \qquad (14.9)$$

На рис. 14.9 показан результат использования этой стратегии (установки листовых значений на основе сбалансированной моды вместо просто моды) для того же набора данных, что и ранее на рис. 14.8. При использовании сбалансированной моды здесь мы получаем большее разнообразие пней (по сравнению со стандартной модой), что позволяет нам эффективнее улавливать нелинейность, присутствующую в этом наборе данных. Для того чтобы увидеть, как сбалансированное большинство использовалось для определения листовых значений в этом случае, давайте рассмотрим один из пней (шестой — в правой средней панели) повнимательнее. Этот пень имеет шесть точек данных слева и четыре точки данных справа. Из шести точек данных, лежащих слева от его точки ветвления, четыре точки имеют значение метки –1 и две точки — значение метки +1, что приводит к сбалансированному голосу большинства соответственно 4/8 и 2/2 для двух классов (отмечая, что в этом

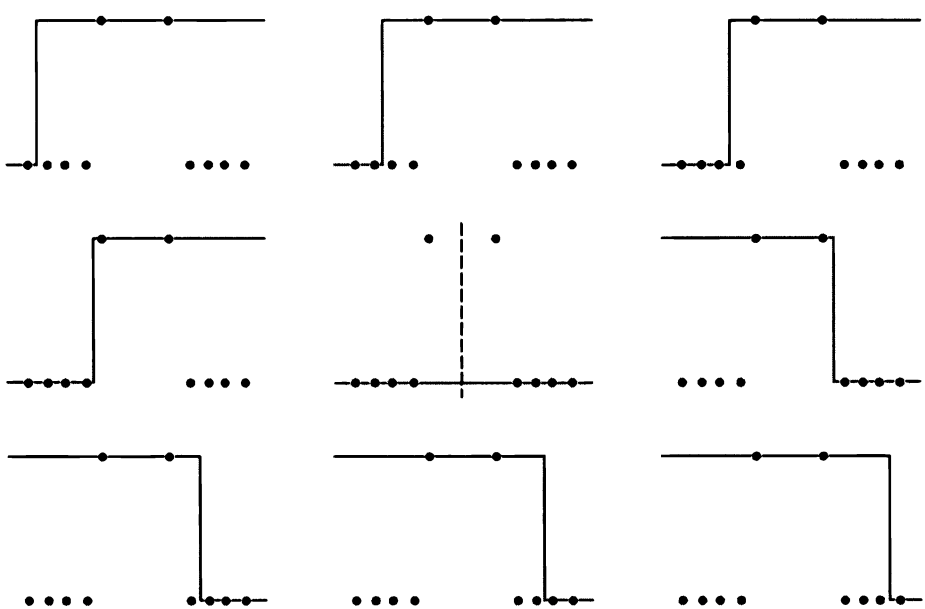


Рис. 14.9. Тот же самый набор данных и набор точек ветвления, которые показаны на рис. 14.8, только здесь для создания листовых значений используется расчет сбалансированной моды из уравнения (14.9)

наборе данных есть в общей сложности восемь точек данных в классе –1 и две в классе +1). Так как $2/2 > 4/8$, листовое значение слева устанавливается равным +1. Сбалансированные голоса большинства справа от точки ветвления для классов –1 и +1 вычисляются похоже соответственно 4/8 и 0/2, и, следовательно, значение правого листа устанавливается равным –1.

14.4.3. Оптимизация классификационных пней

Собрав все вместе, для того чтобы определить оптимальный пень (состоящий из оптимальных значений точки ветвления и листовых значений), мы можем варьировать набор разумно выбранных точек ветвления и строить соответствующие листовые значения для каждого пня, используя либо подход, основанный на *стоимостной функции*, либо подход, основанный *на голосе большинства*, описанные в *разд. 14.4.2*. Для того чтобы определить, какой пень является идеальным для нашего набора данных, мы можем вычислить соответствующую классификационную метрику для каждого экземпляра пня и выбрать тот, который обеспечивает наилучшую результативность. Например, в случае двухклассовой классификации мы можем задействовать метрику точности, подобную введенным в *разд. 6.8*, со *сбалансированной точностью*, обсуждаемой в *разд. 6.8.4*, это самый безопасный выбор, учитывая классовый дисбаланс, с которым мы можем столкнуться на практике, либо более специализированные метрики, такие как *прирост информации* (information gain) [65].

Обратите внимание, что в контексте классификационных деревьев метрики качества часто именуются метриками *чистоты*, поскольку они измеряют чистоту каждого пневого листа с точки зрения представления классов. В идеале выбирается пень, который представляет данные наилучшим образом, в то время как его листья остаются как можно более "чистыми", каждый из которых содержит (главным образом) только члены одного класса, если это возможно.

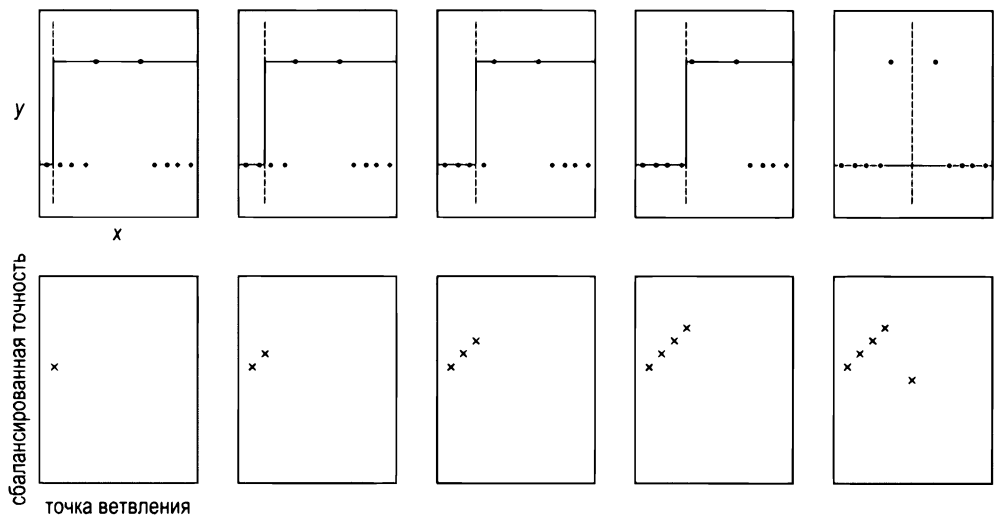


Рис. 14.10. Иллюстрация примера 14.3

Пример 14.3. Подгонка параметров простого классификационного дерева

На рис. 14.10 проиллюстрирована результирующая сбалансированная точность пней, показанных на рис. 14.9. Из-за симметрии этого конкретного набора данных здесь представлены только первые пять пней, из которых четвертый обеспечивает минимальную стоимость и, таким образом, является оптимальным для нашего набора данных.

14.4.4. Более глубокие классификационные деревья

Для того чтобы построить более глубокие классификационные деревья, мы выполняем рекурсию на двух листьях пня и обрабатываем каждый из них так же, как исходные данные, создавая пень из каждого. Как и в случае с регрессионными деревьями (*см. разд. 14.3.4*), это часто приводит к двоичным деревьям *максимальной глубины*, поскольку некоторые ветви останавливаются при очевидных и/или определенных пользователем условиях. В случае классификации одно-единственное условие остановки — это ситуация, когда лист *полностью чист*, т. е. содержит только члены одного класса. В таком случае нет никакой причины продолжать разветвлять такой лист. Другие распространенные условия остановки, часто используемые на практике, включают остановку роста, когда число точек на листе падает ниже некоторого порога и/или ветвления недостаточно повышают точность.

Пример 14.4. Выращивание двух классификационных деревьев максимальной глубины

На рис. 14.11 проиллюстрировано выращивание дерева до максимальной глубины семь на двухклассовом классификационном наборе данных. На рис. 14.12 сделано то же самое для многоклассового классификационного набора данных с $C = 3$ классами. В обоих случаях по мере выращивания дерева обратите внимание на то,

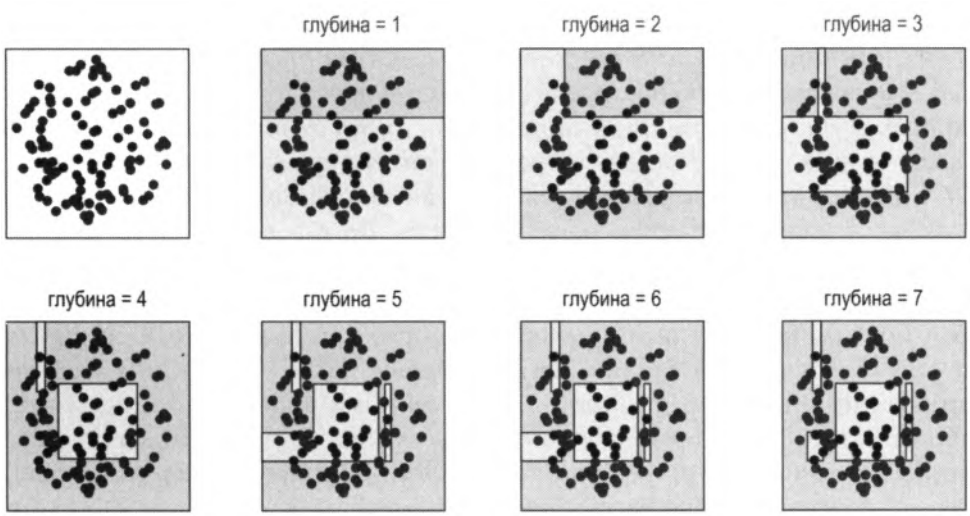


Рис. 14.11. Иллюстрация примера 14.4. Выращивание дерева на двухклассовом классификационном наборе данных

сколько частей входного пространства не изменяется, когда листья на более глубоких ветвях становятся *чистыми*. К тому времени, когда мы достигаем максимальной глубины семь, мы уже находимся в ситуации значительной переподогнанности к обоим наборам данных.

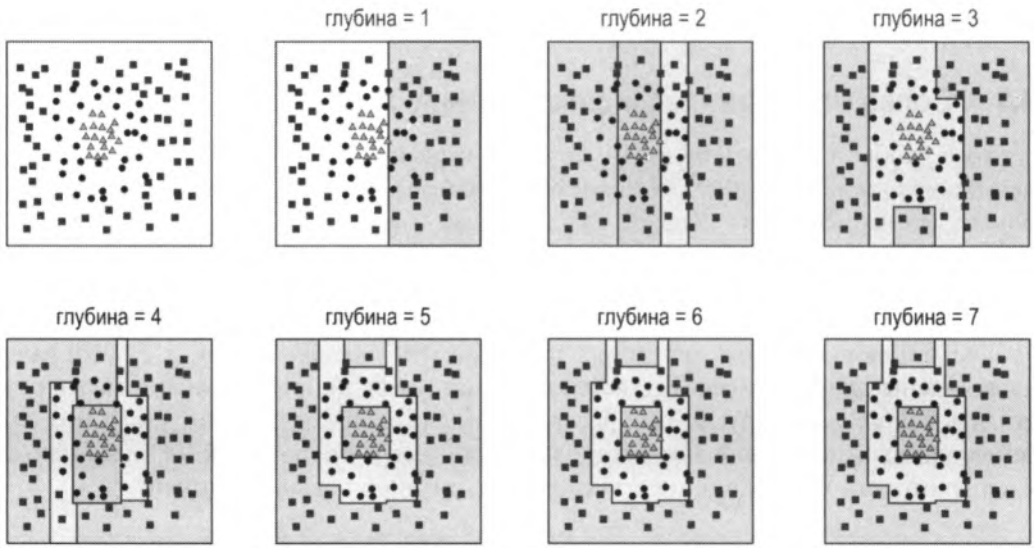


Рис. 14.12. Иллюстрация примера 14.4. Выращивание дерева на многоклассовом классификационном наборе данных

14.5. Бустирование градиента

Как уже упоминалось в *разд. 14.2.3*, универсальные аппроксиматоры на основе глубоких деревьев могут быть построены посредством сложения мелких деревьев. Самый популярный способ построения более глубоких регрессионных и классификационных деревьев посредством сложения состоит в суммировании более мелких деревьев, причем каждое мелкое дерево строится так, как описано в *разд. 14.3* и *14.4*, и в выращивании дерева последовательно по одному мелкому дереву за раз. Эта схема служит примером общего метода бустирования, представленного в *разд. 11.5*. Более того, деревья на самом деле являются самым популярным универсальным аппроксиматором, используемым при применении перекрестной валидации на основе бустирования, причем это сопряжение двух методик очень часто именуется *бустированием* (усилением) *градиента* [66, 67]. Принципы *бустирования*, изложенные в *разд. 11.5*, остаются неизменными в контексте деревьев решений. Однако, имея в распоряжении конкретные знания о том, каким образом следует правильно вписывать регрессионные и классификационные деревья в данные, мы теперь можем подробно рассказать о важных деталях, связанных с бустированием градиента, в которые мы не могли вникнуть ранее.

14.5.1. Правило мелких деревьев

Как описано в *разд. 11.5.1*, *малоемкостные* структурные единицы (любого универсального аппроксиматора) чаще всего используются с бустированием для обеспечения мелкогранулированного модельного поиска. В контексте древесных структурных единиц это приводит к использованию *мелких деревьев*, причем пни и деревья глубиной два являются особенно популярными. Конечно, при построении более глубоких перекрестно-валидированных деревьев посредством бустирования можно использовать более высокоемкостные древесные структурные единицы (глубиной три и выше). Однако бустирование с такими высокоемкостными структурными единицами может легко привести к пропуску оптимальных моделей (как абстрактно показано на рис. 11.30).

14.5.2. Бустирование с деревьями решений

Как подробно описано в *разд. 11.5*, на m-м раунде бустирования мы начинаем с модели, состоящей из полностью настроенной линейной комбинации $m-1$ структурных единиц универсального аппроксиматора (см. уравнение (11.26)). В случае деревьев решений мы можем обойтись без смещения и весов линейной комбинации (поскольку они естественным образом "запекаются" в древесные структурные единицы, как подробно описано в *разд. 11.5.4*) и записать нашу модель как

$$\text{model}_{m-1}(\mathbf{x}, \Theta_{m-1}) = f^{\star}_{s_1}(\mathbf{x}) + f^{\star}_{s_2}(\mathbf{x}) + \ldots + f^{\star}_{s_{m-1}}(\mathbf{x}), \tag{14.10}$$

где каждая функция в этой сумме представляет собой древесную структурную единицу (например, пень), чьи значения точек ветвления и листовые значения были выбраны оптимально. m-й раунд бустирования предусматривает поиск над диапазоном подходящих кандидатов (здесь различных деревьев с разными точками ветвления) и соответствующую оптимизацию листовых значений каждого кандидата. Для построения следующей кандидатной модели мы добавляем перспективную структурную единицу $f_{s_m}(\mathbf{x})$ в $\text{model}_{m-1}(\mathbf{x}, \Theta_{m-1})$, формируя

$$\text{model}_{m}(\mathbf{x}, \Theta_{m}) = \text{model}_{m-1}(\mathbf{x}, \Theta_{m-1}) + f_{s_m}(\mathbf{x}), \tag{14.11}$$

и оптимизируем окончательные значения $f_{s_m}(\mathbf{x})$, используя соответствующую стоимостную функцию (например, стоимость наименьших квадратов для регрессии и стоимость мягкого максимума для классификации) по отношению к тренировочному набору данных. Эта оптимизация листовых значений очень точно отражает подходы, описанные в *разд. 14.3.2* и *14.4.2* соответственно в случае регрессии и классификации.

Например, предположим, что $f_{s_m}(\mathbf{x})$ — это пень, и мы имеем дело с регрессионным случаем со стоимостью наименьших квадратов и набором данных из P точек, обозначаемых через $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$. В соответствии с уравнением (14.5) мы в таком случае должны минимизировать следующую пару стоимостей наименьших квадратов:

$$g(v_L) = \frac{1}{|\Omega_L|} \sum_{p \in \Omega_L} \left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + v_L - y_p\right)^2;$$
$$g(v_R) = \frac{1}{|\Omega_R|} \sum_{p \in \Omega_R} \left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + v_R - y_p\right)^2, \tag{14.12}$$

чтобы надлежаще определить наши два листовых значения v_L и v_R, где Ω_L и Ω_R — индексные множества, определенные в уравнении (14.3), $|\Omega_L|$ и $|\Omega_R|$ обозначают их размеры. Как и стоимостные функции, указанные в уравнении (14.5), эти простые стоимости можно минимизировать совершенно путем проверки условия оптимальности первого порядка (или, что эквивалентно, путем выполнения одного шага метода Ньютона).

Аналогично, если мы имеем дело с двухклассовой классификацией со стоимостью мягкого максимума и значениями меток $y_p \in \{-1, +1\}$, мы устанавливаем листовые значения пня, минимизируя две стоимости (согласно уравнению (14.7)) формы:

$$g(v_L) = \frac{1}{|\Omega_L|} \sum_{p \in \Omega_L} \log\left(1 + e^{-y_p\left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + v_L\right)}\right);$$
$$g(v_R) = \frac{1}{|\Omega_R|} \sum_{p \in \Omega_R} \log\left(1 + e^{-y_p\left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + v_R\right)}\right), \tag{14.13}$$

которые (в обоих случаях) не могут быть минимизированы в замкнутой форме, но должны быть решены с помощью локальной оптимизации. Часто, как описано в *разд. 14.4.2*, это делается просто путем выполнения одного шага метода Ньютона, поскольку он обеспечивает положительный компромисс между качеством минимизации и вычислительными усилиями.

Пример 14.5. Регрессия посредством бустирования градиента

На рис. 14.13 продемонстрировано использование бустирования с регрессионными пнями, которые часто (как обсуждалось в *разд. 11.5.6*) интерпретируются как следующие друг за другом *раунды* подгонки к остатку регрессионного набора данных. Мы видим это в случае простого пня, если переставим члены в уравнении (14.12). Например, $g(v_L)$ в уравнении (14.12) можно переписать как

$$g(v_L) = \frac{1}{|\Omega_L|} \sum_{p \in \Omega_L} \left(v_L - r_p\right)^2, \tag{14.14}$$

где r_p — остаток от p-й точки, определяемый как $r_p = y_p - \text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right)$.

В верхнем ряду этого рисунка приведены исходные наборы данных вместе с результирующей подгонкой, обеспечиваемой моделью, построенной из нескольких раундов бустирования на основе пня. Одновременно в нижнем ряду показаны последующие подгонки на основе пня к остатку, обеспечиваемому самым последним пнем, добавленным в текущую модель.

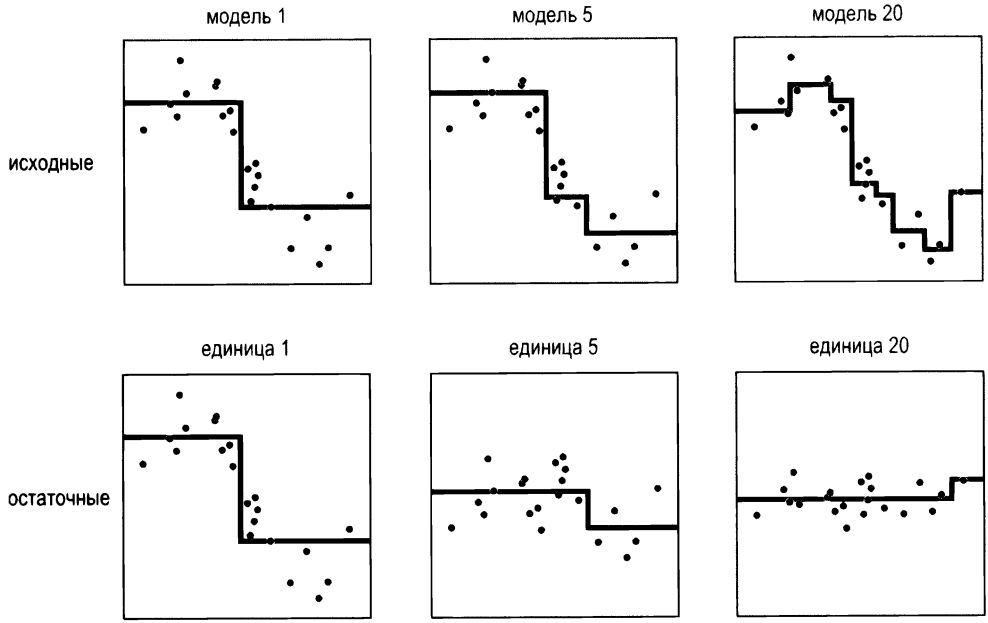


Рис. 14.13. Иллюстрация примера 14.5

Пример 14.6. Обнаружение спама посредством бустирования градиента

В этом примере мы используем бустирование градиента, задействуя пни, и перекрестно валидируем для определения идеального числа раундов бустирования, используя спамный набор данных, впервые описанный в примере 6.10. В этом наборе экспериментов мы используем стоимость мягкого максимума и выделяем 20% этого двухклассового набора данных (случайно) для целей валидации. Мы выполняем 100 раундов бустирования и делаем один шаг метода Ньютона, чтобы настроить каждую пневую функцию. На рис. 14.14 построен график числа неправильных классифицирований как для тренировочных (синим цветом), так и для валидационных (желтым цветом) наборов. Минимальное число неправильных классифициро-

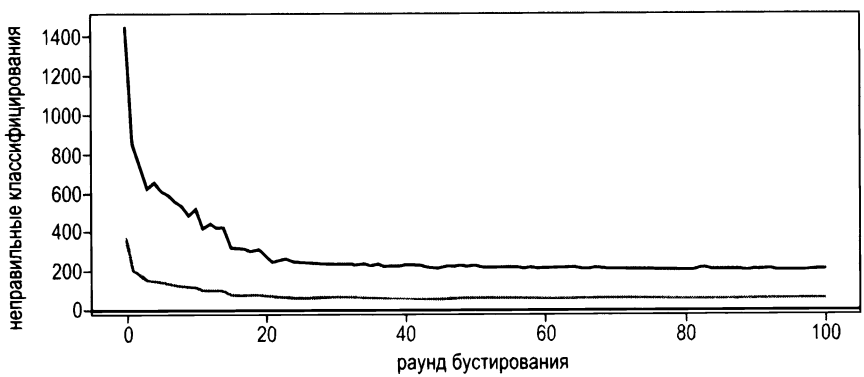


Рис. 14.14. Иллюстрация примера 14.6

ваний в валидационном наборе произошло на шестьдесят пятом раунде бустирования, в результате чего было получено 220 и 50 неправильных классифицирований соответственно на тренировочном и валидационном наборах. Простой линейный классификатор, натренированный на одной и той же части данных, обеспечил 277 и 67 ошибочных классифицирований соответственно на тренировочном и валидационном наборах.

14.6. Случайные леса

Если только интерпретируемость конечной модели человеком не имеет первостепенного значения, то одно рекурсивно определенное регрессионное или классификационное дерево практически никогда не применяется, а вместо этого используется лишь их *бутстрап-агрегированный ансамбль*. Вообще говоря, бутстрап-агрегирование (как подробно описано в *разд. 11.9*) предусматривает совмещение многочисленных перекрестно валидированных моделей для получения одной более высокорезультативной модели. Это можно легко сделать с рекурсивно определенными деревьями, задействовав методы перекрестной валидации, описанные в *разд. 14.7*. Однако на практике часто нет необходимости выращивать каждое дерево с помощью перекрестной валидации, чтобы смягчить их сложность. Вместо этого каждое дерево может быть натренировано на случайной части тренировочных данных, взятых из исходного набора данных, и выращено до заданной максимальной глубины, а затем агрегировано.

В принципе, это может быть сделано с любым универсальным аппроксиматором, но особенно практично с деревьями решений. Это обусловлено как тем, что деревья обходятся дешево в производстве, так и тем, что при выращивании самих отдельных деревьев в качестве *локально* определенных аппроксиматоров (*см. разд. 14.2.2*) совершенно естественно задействовать базовые протоколы остановки ветвления листьев (*см. разд. 14.3.4* и *14.4.4*). В то время как деревья, безусловно, могут быть переподогнаны, даже если они не были перекрестно валидированы, они естественным образом лишены возможности демонстрировать такого рода необузданное осцилляторное переподогнанное поведение, которое возможно с фиксированно-контурными или нейросетевыми моделями[2]. Таким образом, бутстрап-агрегирование набора переподогнанных деревьев часто может успешно бороться с тем видом переподгонки, который представляет каждое дерево в отдельности, что приводит к очень эффективным моделям. Более того, поскольку каждое полностью выращенное дерево в таком ансамбле может быть эффективно усвоено, вычислительный компромисс, т. е. тренировка большого числа полностью выращенных деревьев по сравнению с меньшим числом перекрестно-валидированных деревьев (для построения каждого из которых требуется больше ресурсов), часто оказывается выгодным на практике.

На жаргоне машинного обучения такой ансамбль рекурсивно определенных деревьев часто называется *случайным лесом* [68]. "Случайная" часть названия "слу-

[2] Сравните, например, переподогнанное поведение, демонстрируемое каждым универсальным аппроксиматором на рис. 11.17.

чайный лес" относится к тому факту, что каждое дерево использует случайную часть исходных данных в качестве тренировочной (которая по традиции часто отбирается из исходного набора данных *с возвратом назад*), а также к тому, что часто только случайное подмножество размерностей входных признаков отбирается для жизнеспособных точек ветвления в каждом узле создаваемых деревьев. С целью определения точек ветвления для каждого дерева в таком лесу часто случайно выбирается что-то вроде $\lfloor \sqrt{N} \rfloor$ из N признаков.

Пример 14.7. Классификация на основе случайных лесов

На рис. 14.15 представлен результат бутстрап-агрегирования набора из пяти полностью выращенных классификационных деревьев, натренированных на разных случайных участках простого двухклассового набора данных (в каждом случае для тренировки использовалось 2/3 исходного набора данных, а для валидации — оставшаяся 1/3). Каждая из пяти разбивок показана на небольших панелях слева вместе с границей решения, обеспечиваемой каждой натренированной моделью, причем точки валидационных данных в каждом случае выделены желтой границей. Обратите внимание, что в то время как большинство индивидуальных деревьев переподогнаны к данным, их ансамбль (показанный на большой панели справа) таким не является. Эта ансамблированная модель, подробно описанная в *разд. 11.9*, построена с использованием *моды* пяти классификационных деревьев слева.

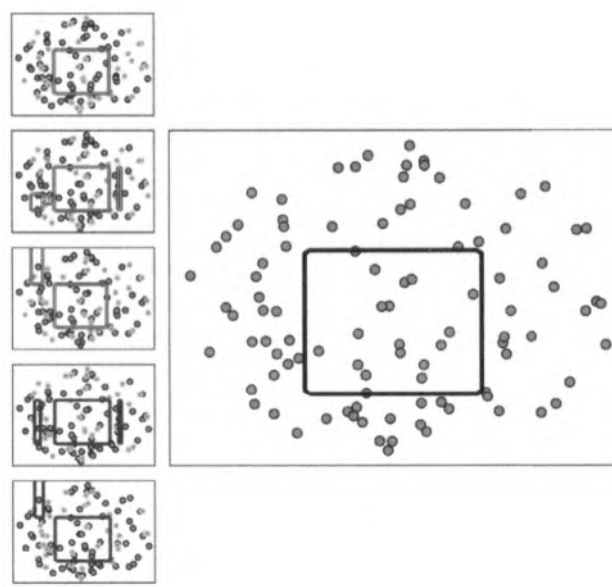


Рис. 14.15. Иллюстрация примера 14.7. Границы решения задаются пятью полностью выращенными классификационными деревьями, каждое из которых выращено на отдельном подмножестве исходных данных (*слева*). Каждое индивидуальное дерево тяготеет к переподгонке к данным, но их агрегированный ансамбль (*справа*) компенсирует это и не переподогнан

14.7. Приемы перекрестной валидации для рекурсивно определенных деревьев

Базовые принципы перекрестной валидации, изложенные в *разд. 11.3–11.6*, в целом применимы к надлежащему построению рекурсивно выращенных регрессионных и классификационных деревьев на практике, с некоторыми техническими различиями, возникающими из-за уникального способа построения таких моделей. Например, мы можем начать с низкоемкостного дерева глубиной один и выращивать его до тех пор, пока не будет достигнута минимальная валидационная ошибка (форма *ранней остановки*, характерная для деревьев). В качестве альтернативы мы можем начать с подгонки к данным глубокого высокоемкостного дерева и постепенно уменьшать его сложность путем *подрезки* листьев, которые не способствуют низкой валидационной ошибке (форма *регуляризации*, характерная для деревьев).

Поскольку рекурсивно определенные деревья в типичной ситуации ансамблируются в случайные леса, причем каждое дерево полностью выращивается до случайной тренировочной части исходного набора данных (как подробно рассмотрено в *разд. 14.6*), описанные здесь методы перекрестной валидации часто используются для смягчения сложности одного регрессионного или классификационного дерева, когда *интерпретируемость* древесной модели *человеком* имеет решающее значение — то, что практически всегда теряется при ансамблировании нескольких нелинейных моделей.

14.7.1. Ранняя остановка

Мы можем легко использовать перекрестную валидацию, чтобы диктовать надлежащую максимальную глубину дерева путем выращивания дерева большой глубины, измеряя валидационную ошибку на каждой глубине дерева и (постфактум) определяя, какая глубина произвела минимальную валидационную ошибку. В качестве альтернативы мы можем остановить рост рано, когда мы будем уверены[3], что была достигнута минимальная валидационная ошибка (нечто аппроксимирующее ее). Этот подход, хотя и используется на практике, приводит к относительно грубому модельному поиску, поскольку емкость дерева растет экспоненциально с увеличением глубины.

Как подробно описано в предыдущих двух разделах, нередко для остановки ветвления листа используются практические соображения (независимо от того, выполняется ли перекрестная валидация). Они включают остановку ветвления, если лист содержит одну-единственную точку данных или предшествующее (малое) число точек, если все точки данных принадлежат к одному и тому же классу (в случае классификации) или имеют приближенно одинаковое выходное значение (в случае

[3] Как и с любой формой ранней остановки, определение того, когда валидационная ошибка является минимальной, "на лету" не всегда является простой задачей, поскольку валидационная ошибка не всегда падает и растет монотонно (как обсуждалось в контексте бустирования и регуляризации соответственно в *разд. 11.5.3* и *11.6.2*).

регрессии). Для того чтобы создать перекрестно-валидационный более мелкозернистый поиск, мы можем добавить критерии, ориентированные на валидационные ошибки, чтобы остановить рост отдельных листьев. Самый простой такой критерий — проверить, приведет ли ветвление листа к *снижению* валидационной ошибки (или снижению тренировочной ошибки за пределы заданного порога): если приведет, то лист разветвляется, в противном случае рост листа прекращается. Этот подход к перекрестной валидации уникален тем, что валидационная ошибка будет всегда монотонно уменьшаться по мере увеличения максимальной глубины дерева, но может привести к *недоподгонке* моделей из-за преждевременной остановки роста листьев.

Пример 14.8. Ранняя остановка по глубине и росту листьев

На рис. 14.16 показан пример перекрестного валидирования максимальной глубины дерева в диапазоне от 1 до 5 для простого регрессионного набора данных. Набор данных, для которого тренировочные данные окрашены в синий цвет, а валидационные данные — в оранжевый цвет, показан вместе с каждой последующей подгонкой в верхнем ряду на рис. 14.16, в то время как соответствующая тренировочная/валидационная ошибка отображена в его нижнем ряду.

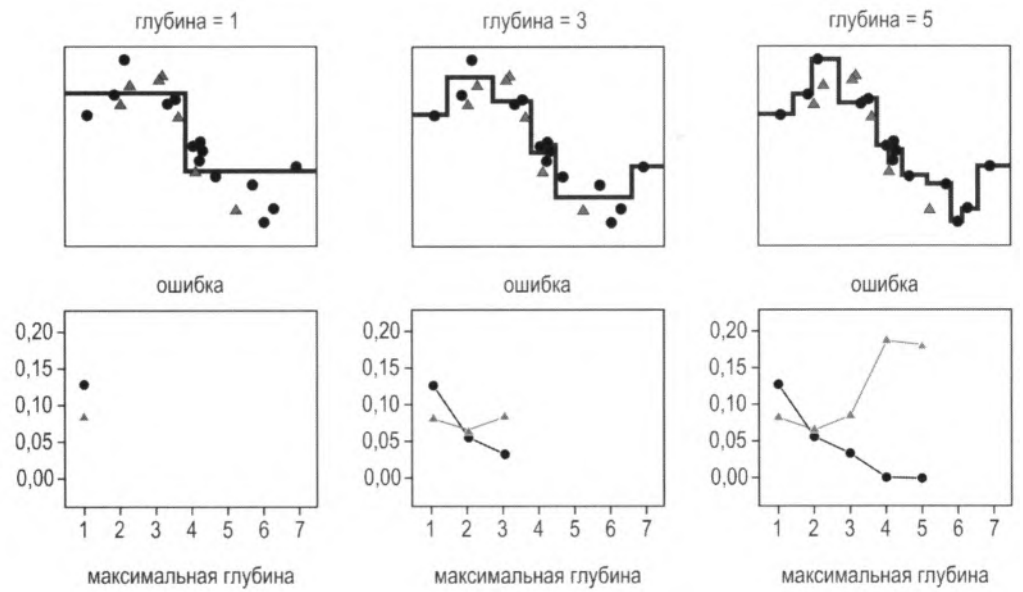


Рис. 14.16. Иллюстрция примера 14.8

На рис. 14.17 показаны два примера перекрестного валидирования максимальной глубины дерева в диапазоне от 1 до 5 еще для одного регрессионного набора данных, где рост листьев теперь останавливается, когда валидационная ошибка не улучшается. В каждом случае набор данных, где тренировочные данные окрашены в синий цвет, а валидационные данные — в оранжевый цвет, изображен вместе с каждой последующей подгонкой вверху, а соответствующая тренировочная/ва-

лидационная ошибка представлена непосредственно под ним. В первом прогоне (показанном в двух верхних рядах на рис. 14.17) из-за особой разбивки тренировочных и валидационных данных рост каждого листа немедленно прекращается, что приводит к недоподогнанному представлению с глубиной один. При втором прогоне на другой тренировочно-валидационной разбивке данных (показанном в двух нижних рядах на рис. 14.17) рост дерева продолжает улучшать валидационную ошибку вплоть до максимальной протестированной глубины, что приводит к значительно более качественному представлению.

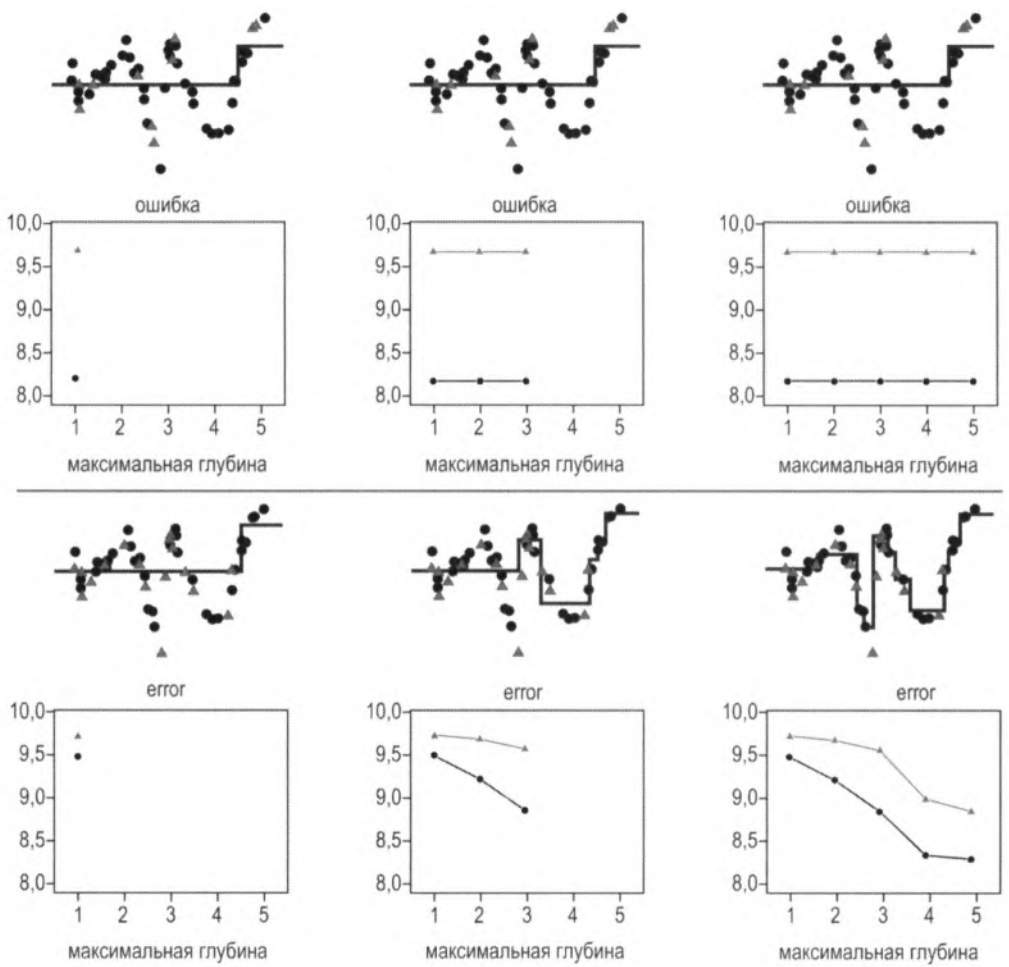


Рис. 14.17. Иллюстрция примера 14.8

14.7.2. Подрезка

В отличие от случая, когда процесс начинается с малоемкостного (мелкого) дерева и выращивания его посредством ранней остановки, мы можем начать с подгонки высокоемкостного (глубокого) дерева и удалять листья, которые не улучшают ва-

лидационную ошибку, до тех пор, пока не останется древесная структура с минимальной валидацией. Этот прием, иллюстрируемый рис. 14.18, называется *подрезкой*, потому что он влечет за собой проверку изначально чрезмерно сложного дерева и подрезку его листьев, подобно тому, как подрезка естественных деревьев производится путем срезания лишних листьев и ветвей. Подрезка — это специфичная для дерева форма *регуляризации*, основанная на перекрестной валидации, обсуждавшейся ранее в *разд. 11.6*.

Хотя ранняя остановка часто является эффективнее с вычислительной точки зрения, чем подрезка, последняя обеспечивает более мелкозернистый модельный поиск при определении древесной структуры с минимальной валидационной ошибкой, поскольку рост дерева/листа беспрепятствен и урезается только постфактум.

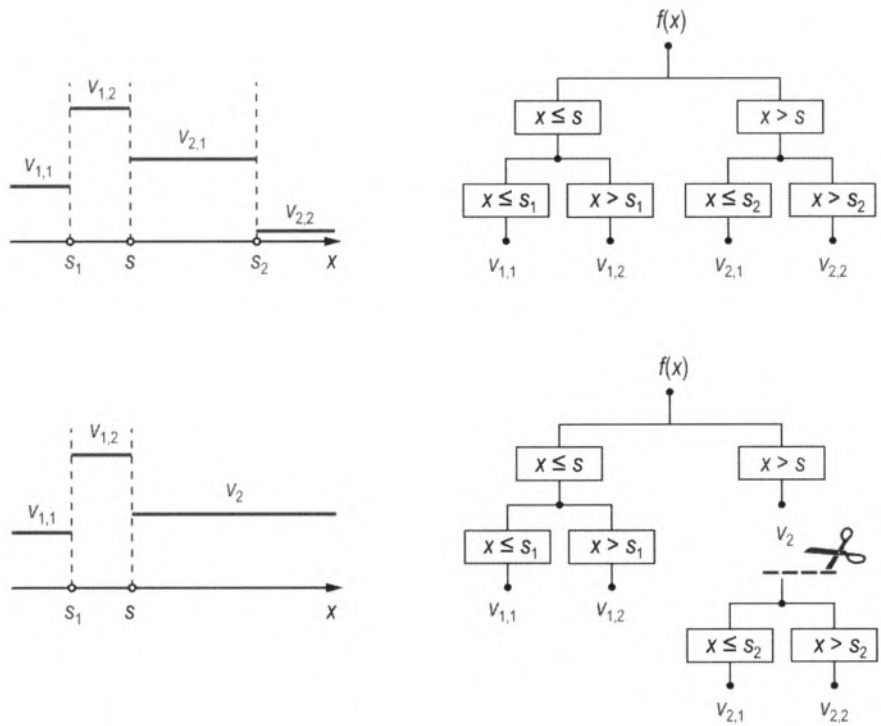


Рис. 14.18. Подрезка. Полностью выросшее дерево глубиной два с четырьмя листьями (*сверху*). Подрезанная версия исходного дерева, в которой листья $v_{2,1}$ и $v_{2,2}$ подрезаются и заменяются одним листом (*снизу*)

14.8. Заключение

В этой главе мы обсудили ряд важных технических вопросов, связанных с древесными универсальными аппроксиматорами, которые впервые были представлены в *разд. 11.2.3*. Мы начали в *разд. 14.2* с более формального описания пней, а также более глубоких деревьев, которые, как подробно описано в этом разделе, могут

быть сформированы посредством рекурсии или суммирования. Рекурсивно определенные регрессионные и классификационные деревья были затем подробно объяснены и описаны в *разд. 14.3* и *14.4*. Бустирование градиента — специфическое применение перекрестной валидации на основе бустирования (подробно описанное в *разд. 11.5*) к деревьям решений — было затронуто в *разд. 14.5*. Схожим образом случайные леса — специализированное применение бутстрап-агрегирования (подробно описанное в *разд. 11.9*) для деревьев решений — были описаны в *разд. 14.6*. Наконец, использование перекрестной валидации с рекурсивно определенными деревьями решений — как с наивной, так и регуляризационной точек зрения — было рассмотрено в *разд. 14.7*.

14.9. Упражнения

† Данные, необходимые для выполнения следующих ниже упражнений, можно скачать из репозитория книги на github по адресу:
github.com/jermwatt/machine_learning_refined.

14.1. Выращивание глубоких деревьев путем сложения

Показать, что в общем случае сложение $2^D - 1$ пней (со скалярным входом) создаст дерево глубиной D (при условии, что пни не имеют общих точек ветвления).

14.2. Подгонка параметров простого регрессионного дерева

Повторить эксперимент, описанный в примере 14.1, и воспроизвести графики, показанные на рис. 14.6.

14.3. Имплементировать регрессионное дерево в коде

Повторить эксперимент, описанный в примере 14.2, путем кодирования рекурсивно определенного регрессионного дерева. От вас не требуется воспроизводить рис. 14.7. Вместо этого следует измерить и построить график ошибки наименьших квадратов на каждой глубине вашего дерева.

14.4. Имплементировать двухклассовое классификационное дерево в коде

Повторить первый эксперимент, описанный в примере 14.4, путем кодирования рекурсивно определенного двухклассового классификационного дерева. От вас не требуется воспроизводить рис. 14.11. Вместо этого следует измерить глубину и построить график числа неправильных классифицирований на каждой глубине вашего дерева.

14.5. Имплементировать многоклассовое классификационное дерево в коде

Повторить второй эксперимент, описанный в примере 14.4, путем кодирования рекурсивно определенного многоклассового классификационного дерева. От вас не требуется воспроизводить рис. 14.12. Вместо этого следует измерить и построить график числа неправильных классифицирований на каждой глубине вашего дерева.

14.6. Бустирование градиента для регрессии

Повторить эксперимент, описанный в примере 14.5, путем кодирования алгоритма бустирования градиента, задействовав регрессионные пни. Воспроизвести рис. 14.13, иллюстрирующий бустированное дерево, а также наилучший пень, подогнанный к остатку на первом, втором и десятом раундах бустирования.

14.7. Бустирование градиента для классификации

Определить значения листьев пня, добавленного на m-м раунде бустирования в классификационное дерево, путем минимизирования стоимости мягкого максимума из уравнения (14.13) посредством одного шага метода Ньютона.

14.8. Случайные леса

Повторить эксперимент, описанный в примере 14.7, путем кодирования случайного леса, построенного из классификационных деревьев. От вас не требуется воспроизводить рис. 14.15. Однако вы можете убедиться, что ваша имплементация работает надлежаще, проверив, что окончательная точность вашего классификатора, основанного на случайном лесе, превосходит точность многочисленных отдельных деревьев в ансамбле, либо в качестве альтернативы вы можете задействовать тестовый набор, отложив малую часть исходных данных.

14.9. Ограничение деревьев вне их тренировочного диапазона

В этой главе мы увидели, что деревья являются эффективными нелинейными аппроксимациями и не страдают от осцилляторного поведения, которое может отрицательно влиять на глобальные аппроксиматоры, такие как полиномы и нейронные сети (*см. разд. 14.6*). Тем не менее деревья решений по своей природе не могут эффективно работать вне своего тренировочного диапазона. В этом упражнении вы поймете, почему это так, натренировав регрессионное дерево с использованием данных о студенческой кредитной задолженности, впервые показанных на рис. 1.8. Используйте свое натренированное дерево, чтобы предсказать, какой будет суммарная задолженность студентов в 2050 году. Есть ли в ней смысл? Объясните почему.

14.10. Наивная перекрестная валидация

Повторить эксперимент, описанный в примере 14.8, результаты которого показаны на рис. 14.16.

Часть IV

Приложения

1 ПРИЛОЖЕНИЕ
Продвинутые методы оптимизации первого и второго порядков

П1.1. Введение

В этом приложении представлены продвинутые приемы оптимизации первого и второго порядков, которые предназначены для смягчения естественных недостатков градиентного спуска и метода Ньютона, подробно описанных в *разд. 3.6* и *4.4* соответственно.

П1.2. Стохастический градиентный спуск с импульсом

В *разд. 3.6* мы обсудили фундаментальный вопрос, связанный с *направлением* отрицательного градиента: он может (в зависимости от минимизируемой функции) быстро осциллировать, приводя к зигзагообразным шагам градиентного спуска, которые замедляют минимизацию. В этом разделе мы опишем популярное усовершенствование шага стандартного градиентного спуска, именуемое *стохастическим градиентным спуском с импульсом*, которое специально разработано для смягчения этой проблемы. Суть идеи исходит из области анализа временны́х рядов и, в частности, представляет собой инструмент для сглаживания данных временно́го ряда, известный как *экспоненциальное среднее*. Мы сначала введем понятие экспоненциального среднего, а затем подробно опишем то, как оно может быть интегрировано в шаг стандартного градиентного спуска, чтобы смягчить некоторые аспекты нежелательного зигзагообразного поведения (когда оно происходит) и, следовательно, ускорить градиентный спуск.

П1.2.1. Экспоненциальное сглаживание

На рис. П1.1 приведен пример данных временно́го ряда. Этот конкретный пример показывает реальный фрагмент цены финансовой акции, измеренный в 450 последовательных временны́х точках. В общем случае данные временного ряда состоят из последовательности K упорядоченных точек w_1, w_2, ..., w_K, т. е. точка w_1 появляется раньше (т. е. создается и/или собирается раньше) w_2, точка w_2 раньше w_3 и т. д. Например, мы генерируем (потенциально многомерный) временной ряд точек всякий раз, когда выполняем схему локальной оптимизации с шагами $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}$, т. к. она производит последовательность упорядоченных точек $\mathbf{w}^1$, $\mathbf{w}^2$, ..., $\mathbf{w}^K$.

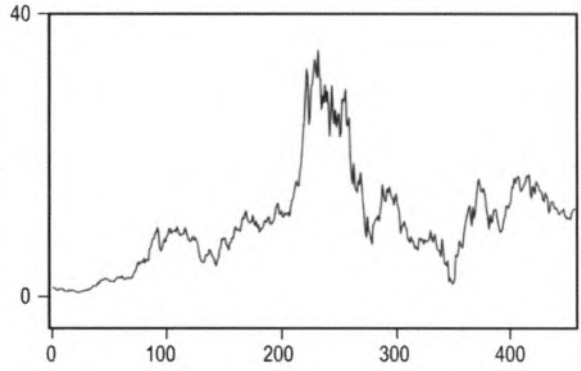


Рис. П1.1. Пример данных временного ряда, представляющих цену финансовой акции, измеренную в 450 следующих друг за другом точках во времени

Поскольку сырые значения временного ряда часто осциллируют, общепринято их *сглаживать* (чтобы устранять эти зигзагообразные движения) для более качественной визуализации или перед дальнейшим анализом.

Экспоненциальное сглаживание является одним из наиболее популярных методов сглаживания временных рядов и используется практически в каждой области применения, в которой возникают такого рода данные. На рис. П1.2 приведен результат сглаживания данных, показанных на рис. П1.1. Прежде чем мы увидим, как вычисляется экспоненциальное среднее, сначала полезно посмотреть, как вычислить кумулятивное среднее из K входных точек w^1, w^2, ..., w^K, т. е. среднее значение первых двух точек, среднее значение первых трех точек и т. д. Обозначив среднее значение первых k точек через h^k, мы можем записать:

$$
\begin{aligned}
h^1 &= w^1; \\
h^2 &= \frac{w^1 + w^2}{2}; \\
h^3 &= \frac{w^1 + w^2 + w^3}{3}; \\
&\ldots \\
h^K &= \frac{w^1 + w^2 + \ldots + w^K}{K}.
\end{aligned}
\qquad \text{(П1.1)}
$$

Обратите внимание, как на каждом шаге h^k по существу *суммирует* входные точки с w^1 по w^k посредством простейшей статистики: их выборочного среднего. То, каким образом кумулятивное среднее записывается в уравнении (П1.1), означает, что нам нужен доступ к каждой сырой точке с w^1 по w^k, чтобы вычислить k-е кумулятивное среднее h^k. В качестве альтернативы мы можем записать это кумулятивное среднее, выразив h^k для $k > 1$ рекурсивным способом, включающим только

его предыдущее кумулятивное среднее h^{k-1} и текущее значение временного ряда w^k, как

$$h^k = \frac{k-1}{k} h^{k-1} + \frac{1}{k} w^k . \qquad (П1.2)$$

С вычислительной точки зрения рекурсивный способ определения кумулятивного среднего гораздо эффективнее, поскольку на *k*-м шаге нам нужно хранить и обрабатывать только два значения, а не *k* из них.

Экспоненциальное среднее представляет собой простое видоизменение в формуле кумулятивного среднего.

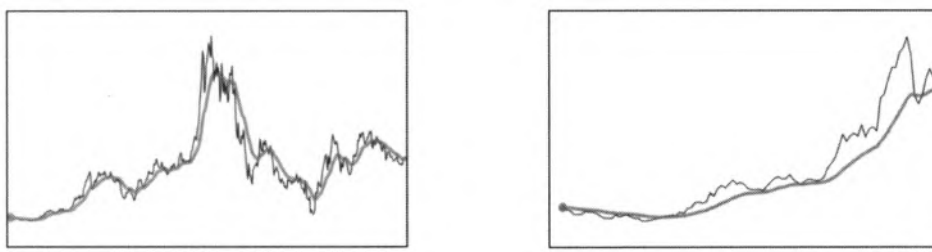


Рис. П1.2. Экспоненциальное среднее значение (сиреневым цветом) данных временного ряда, показанных на рис. П1.1 (*слева*). Экспоненциальное среднее значение только первых 100 точек временного ряда, которое является гладкой аппроксимацией опорных данных временного ряда (*справа*)

Обратите внимание, что на каждом шаге в уравнении (П1.2) коэффициенты на h^{k-1} и w^k *всегда* в сумме составляют 1, т. е. $\frac{k-1}{k} + \frac{1}{k} = 1$. По мере увеличения *k* оба коэффициента изменяются: коэффициент на h^{k-1} становится ближе к 1, а коэффициент на w^k — ближе к 0. С экспоненциальным средним мы эти коэффициенты *замораживаем*. То есть коэффициент на h^{k-1} мы заменяем постоянным значением $\beta \in [0; 1]$, а коэффициент на w^k — значением $1-\beta$, получая аналогичную рекурсивную формулу для экспоненциального среднего как

$$h^k = \beta h^{k-1} + (1-\beta) w^k . \qquad (П1.3)$$

Очевидно, что параметр β здесь управляет компромиссом: чем меньше мы устанавливаем β, тем ближе наше экспоненциальное среднее аппроксимирует сам сырой (зигзагообразный) временной ряд, тогда как чем больше мы его устанавливаем, тем больше каждое последующее среднее выглядит как его предшественник (что приводит к более гладкой кривой). Независимо от того, как мы устанавливаем β, каждый h^k в экспоненциальном среднем по-прежнему можно рассматривать как сводку для w^k и всех точек временного ряда, которые предшествуют ей.

Почему эта слегка скорректированная версия кумулятивного среднего называется *экспоненциальным* средним? Потому что если мы откатим обновление, показанное

в уравнении (П1.3), чтобы выразить h^k только в терминах предшествующих элементов временного ряда, как мы сделали для кумулятивного среднего в уравнении (П1.1), то проявится экспоненциальный (или степенной) шаблон в коэффициентах.

Обратите внимание, что при выведении экспоненциального среднего мы исходили из того, что наши данные временного ряда являются *одномерными*, т. е. каждая исходная точка w^k является скаляром. Однако эта идея соблюдается независимо от входной размерности. Мы также можем определить экспоненциальное среднее временного ряда общих N-мерных точек $\mathbf{w}^1, \mathbf{w}^2, ..., \mathbf{w}^K$ путем инициализирования $\mathbf{h}^1 = \mathbf{w}^1$, а затем для $k > 1$, выстраивая $\mathbf{h}^k$ как

$$\mathbf{h}^k = \beta\mathbf{h}^{k-1} + (1-\beta)\mathbf{w}^k. \qquad (П1.4)$$

Здесь экспоненциальное среднее $\mathbf{h}^k$ на шаге k тоже является N-мерным.

П1.2.2. Смягчение зигзагообразного поведения градиентного спуска

Как упоминалось ранее, последовательность шагов градиентного спуска можно рассматривать как *временной ряд*. Действительно, если мы сделаем K шагов градиентного спуска, то создадим временной ряд упорядоченных шагов градиентного спуска $\mathbf{w}^1, \mathbf{w}^2, ..., \mathbf{w}^K$ и направлений спуска $-\nabla g\left(\mathbf{w}^0\right), -\nabla g\left(\mathbf{w}^1\right), ..., -\nabla g\left(\mathbf{w}^{K-1}\right)$.

Для того чтобы попытаться смягчить некоторое зигзагообразное поведение наших шагов градиентного спуска $\mathbf{w}^1, \mathbf{w}^2, ..., \mathbf{w}^K$ (как подробно описано в *разд. 3.6.3*), мы могли бы вычислить их *экспоненциальное среднее*. Однако мы не хотим сглаживать шаги градиентного спуска *после* того, как они были созданы, поскольку "ущерб уже нанесен" в том смысле, что зигзагообразное движение уже замедлило ход выполнения градиентного спуска. Вместо этого мы хотим сглаживать шаги *по мере их создания*, чтобы наш алгоритм добился большей прогрессии в минимизации.

Каким образом можно сглаживать шаги по мере их создания? Вспомните из *разд. 3.6.3*, что главной причиной зигзагообразного градиентного спуска является осцилляторная природа самих направлений (отрицательного) градиента. Другими словами, если направления спуска $-\nabla g\left(\mathbf{w}^0\right), -\nabla g\left(\mathbf{w}^1\right), ..., -\nabla g\left(\mathbf{w}^{K-1}\right)$ являются зигзагообразными, то такими являются и сами шаги градиентного спуска. Поэтому представляется разумным предположить, что если мы будем сглаживать сами эти направления по мере того, как они создаются во время выполнения градиентного спуска, то сможем произвести шаги градиентного спуска, которые не так сильно оциллируют, и, следовательно, добьемся большей прогрессии в минимизации.

Для этого мы сначала инициализируем $\mathbf{d}^0 = -\nabla g\left(\mathbf{w}^0\right)$, а затем для $k > 1$ вычисляем экспоненциально усредненное направление спуска $\mathbf{d}^{k-1}$ (используя формулу в уравнении (П1.4)) по формуле

$$\mathbf{d}^{k-1} = \beta\mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right). \qquad (П1.5)$$

Затем мы можем использовать это направление спуска в нашем общем каркасе локальной оптимизации, чтобы сделать следующий шаг:

$$\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}. \tag{П1.6}$$

Это экспоненциальное усреднение добавляет в нашу базовую схему градиентного спуска только один дополнительный шаг, образуя шаг *импульсно-ускоренного* градиентного спуска формы[1]:

$$\begin{aligned} \mathbf{d}^{k-1} &= \beta \mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right); \\ \mathbf{w}^k &= \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}. \end{aligned} \tag{П1.8}$$

Термин "импульс" здесь относится к новому экспоненциально усредненному направлению спуска $\mathbf{d}^{k-1}$, которое по определению является функцией *каждого* отрицательного градиента, предшествовавшего ему. Следовательно, $\mathbf{d}^{k-1}$ улавливает среднее значение, или "импульс", предшествующих ему направлений.

Как и в случае любого экспоненциального среднего, выбор $\beta \in [0; 1]$ обеспечивает компромисс. С одной стороны, чем меньше выбран β, тем *больше* экспоненциальное среднее напоминает фактическую последовательность отрицательных направлений спуска, поскольку в обновлении используется *больше* каждого направления отрицательного градиента, но тем *меньше* эти направления спуска суммируют все ранее замеченные отрицательные градиенты. С другой стороны, чем больше выбран β, тем меньше эти экспоненциально усредненные шаги спуска напоминают направления отрицательного градиента, поскольку каждое обновление будет использовать меньше каждого последующего направления отрицательного градиента, но тем *больше* они представляют собой их сводку. Часто на практике выбирают более крупные значения β, например из диапазона [0,7; 1].

Пример П1.1. Ускорение градиентного спуска в простой квадратичной функции

В этом примере мы сравним прогон стандартного градиентного спуска с *импульсно-ускоренной* версией, используя квадратичную функцию вида

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}, \tag{П1.9}$$

где $a = 0$, $\mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ и $\mathbf{C} = \begin{bmatrix} 0,5 & 0 \\ 0 & 9,75 \end{bmatrix}$.

---

[1] Иногда этот шаг записывается несколько иначе: вместо усреднения направлений *отрицательного* градиента сам градиент усредняется экспоненциально, а затем *шаг* делается в их *отрицательном* направлении. Это означает, что мы инициализируем наше экспоненциальное среднее при первом *отрицательном* направлении спуска $\mathbf{d}^0 = -g\left(\mathbf{w}^0\right)$, а для $k > 1$ общее направление спуска и соответствующий шаг вычисляются как:

$$\begin{aligned} \mathbf{d}^{k-1} &= \beta \mathbf{d}^{k-2} + (1-\beta)\nabla g\left(\mathbf{w}^{k-1}\right); \\ \mathbf{w}^k &= \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}. \end{aligned} \tag{П1.7}$$

Сделаем три прогона по 25 шагов: прогон градиентного спуска и два прогона импульсно-ускоренного градиентного спуска с двумя вариантами параметра $\beta \in \{0,2;\ 0,7\}$. Все три прогона инициализируются в одной и той же точке $\mathbf{w}^0 = [10 \quad 1]^T$ и используют одну и ту же длину шага $\alpha = 10^{-1}$.

Результирующие шаги, выполняемые прогоном стандартного градиентного спуска, представлены на рис. П1.3, *сверху* (где присутствует значительный зигзаг) и импульсно-ускоренными версиями с использованием $\beta = 0,2$ и $\beta = 0,7$ соответственно в центре и внизу рисунка. Обе импульсно-ускоренные версии явно превосходят стандартную схему, поскольку достигают точки, более близкой к истинному минимуму квадратической функции. Также обратите внимание, что общий путь, пройденный градиентным спуском, более гладкий на третьем рисунке благодаря большему значению β.

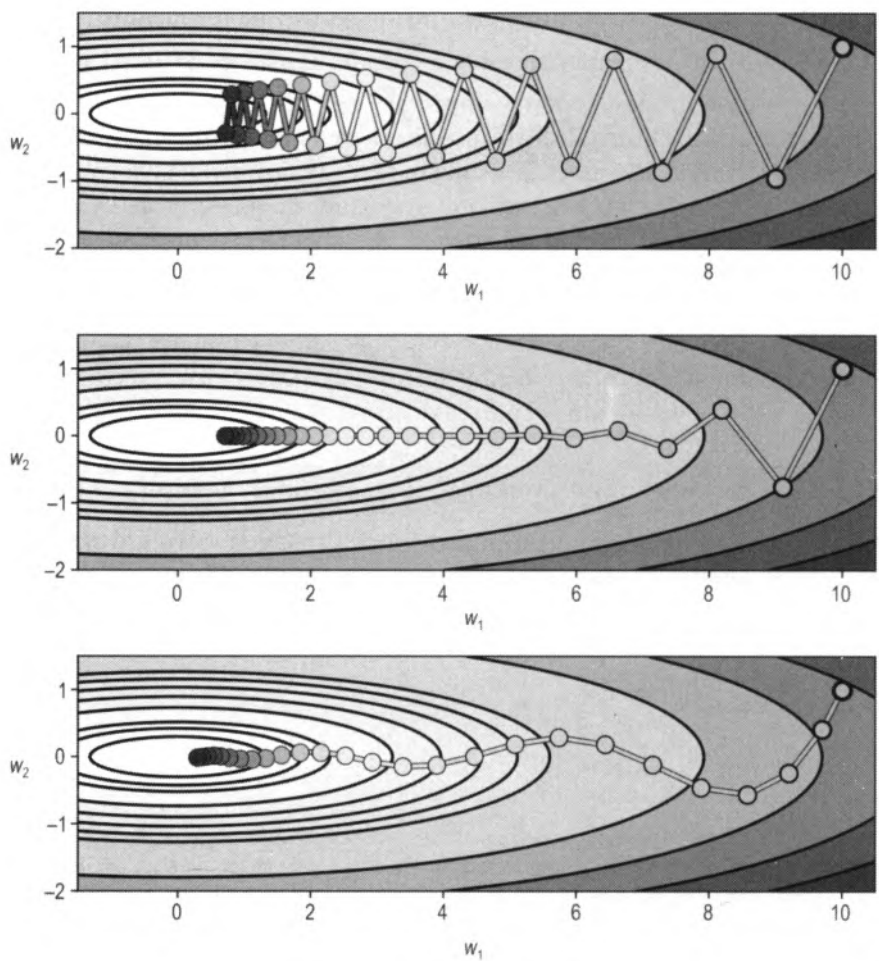


Рис. П1.3. Иллюстрация примера П1.1. Зигзагообразное поведение градиентного спуска можно смягчить с помощью шага импульсно-ускоренного градиентного спуска из уравнения (П1.8)

П1.3. Нормализованный градиентный спуск

В *разд. 3.6* мы обсудили фундаментальный вопрос, связанный с *модулем* отрицательного градиента и тем фактом, что отрицательность исчезает вблизи стационарных точек, заставляя градиентный спуск ползти медленно вблизи стационарных точек. В частности, это означает (в зависимости от минимизируемой функции), что процедура поиска минимума может останавливаться вблизи седловых точек. В этом разделе мы опишем популярное усовершенствование схемы стандартного градиентного спуска, именуемое *нормализованным градиентным спуском*, которое специально разработано для улучшения данной проблемы. Суть идеи заключается в ответе на простой вопрос: поскольку (исчезающий) *модуль* отрицательного градиента заставляет градиентный спуск ползти медленно вблизи стационарных точек или останавливаться в седловых точках, что произойдет, если мы просто проигнорируем модуль на каждом шаге путем его *нормализации*?

П1.3.1. Нормализация модуля полного градиента

В *разд. 3.6.4* мы увидели, как длина шага стандартного градиентного спуска пропорциональна модулю градиента, выраженному алгебраически как $\alpha\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2$.

Более того, мы также выяснили, почему градиентный спуск медленно ползет вблизи стационарных точек: это происходит потому, что вблизи таких точек *модуль* градиента исчезает.

Поскольку модуль градиента виновен в медленном ползании вблизи статических точек, что произойдет, если мы возьмем и просто его проигнорируем, нормировав модуль по шагу, и двинемся в направлении самого отрицательного градиента?

Один из способов нормализовать направление (градиентного) спуска — это разделить его на модуль. Мы получим *нормализованный шаг градиентного спуска* вида

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\frac{\nabla g\left(\mathbf{w}^{k-1}\right)}{\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2}. \qquad (\text{П1.10})$$

При этом мы действительно игнорируем модуль градиента, т. к.

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|-\alpha\frac{\nabla g\left(\mathbf{w}^{k-1}\right)}{\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2}\right\|_2 = \alpha. \qquad (\text{П1.11})$$

Другими словами, если мы нормализуем модуль градиента на каждом шаге градиентного спуска, то длина каждого шага в точности равна значению нашего параметра длины шага α. Именно это мы и сделали с методом случайного поиска в *разд. 2.5.2*.

Обратите внимание: если мы слегка перепишем полностью нормализованный шаг в уравнении (П1.10) как

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{\alpha}{\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2}\nabla g\left(\mathbf{w}^{k-1}\right), \qquad (\text{П1.12})$$

то сможем интерпретировать наш полностью нормализованный по модулю шаг как шаг стандартного градиентного спуска со значением длины шага $\frac{\alpha}{\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2}$, которое *корректирует себя* на каждом шаге, основываясь на модуле градиента, чтобы обеспечить длину каждого шага в точности равным α.

Также обратите внимание на то, что на практике часто бывает полезно добавить в модуль градиента малую константу ε (например, 10^{-7} или меньше), чтобы избежать потенциального деления на ноль (где модуль полностью исчезает):

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{\alpha}{\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2 + \varepsilon} \nabla g\left(\mathbf{w}^{k-1}\right). \qquad \text{(П1.13)}$$

Пример П1.2. Смягчение медленного ползания вблизи минимумов и седловых точек

На рис. П1.4, *слева* показан повтор прогона градиентного спуска, впервые подробно описанного в примере 3.14, только здесь мы выполняем полностью нормализованный шаг градиентного спуска. Мы используем то же число шагов и значение длины шага, что и в этом примере (что привело к медленному ползанию в случае со стандартной схемой). Здесь, однако, нормализованный шаг — без изменения исчезающего модуля градиента — способен легко пройти через плоский участок этой функции и найти точку, очень близкую к минимуму в начале координат. Сравнивая этот прогон с исходным прогоном (стандартного градиентного спуска) на рис. 3.14, мы видим, что нормализованный прогон значительно приближается к глобальному минимуму функции.

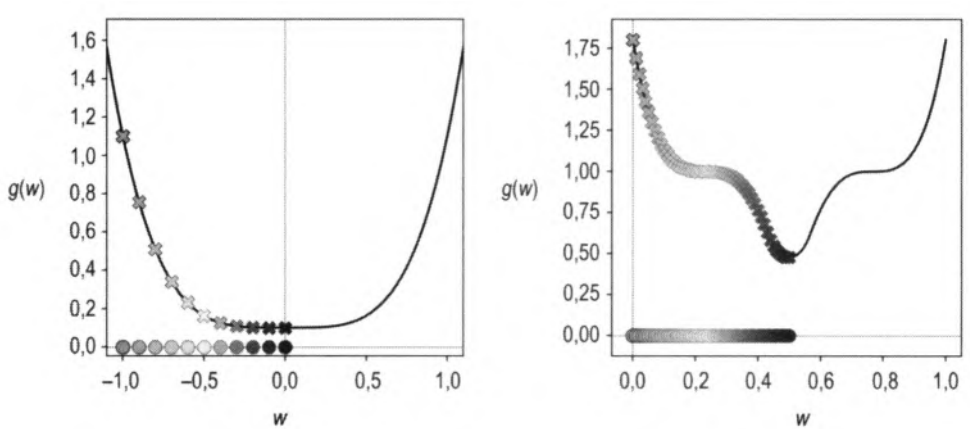


Рис. П1.4. Иллюстрация примера П1.2. Нормализуя градиент, мы можем преодолеть медленное ползание градиентного спуска вблизи минимумов функции (*слева*) и седловых точек (*справа*)

На рис. П1.4, *справа* показан повтор прогона градиентного спуска, впервые подробно описанного в примере 3.14, только здесь мы выполняем полностью нормализованный шаг градиентного спуска. Мы используем то же число шагов и значение

длины шага, что и в этом примере (что привело к остановке в седловой точке в случае со стандартной схемой). Здесь, однако, нормализованный шаг — без влияния исчезающего модуля градиента — может легко проходить через плоский участок седловой точки и достигать точки этой функции, близкой к минимуму.

Пример П1.3. Компромисс при использовании нормализованного градиентного спуска

На рис. П1.5 показано сравнение полностью нормализованного (*слева*) и стандартного (*справа*) градиентного спуска на простой квадратичной функции

$$g(w) = w^2 . \qquad (П1.14)$$

Оба алгоритма используют одну и ту же начальную точку ($w^0 = -3$), параметр длины шага ($\alpha = 0{,}1$) и максимальное число итераций (по 20). Шаги окрашены от зеленого до красного цвета для обозначения начальной и конечной точек каждого прогона, кружки отмечают фактические шаги во входном пространстве, а значки × — соответствующие им результаты вычисления функций.

Обратите внимание, как стандартная версия мчится к глобальному минимуму функции, в то время как нормализованная версия, делая шаги фиксированной длины, добирается туда только частично. Это поведение указывает на то, что нормализованный шаг не сможет использовать градиент, когда он является большим — как это делает стандартный метод, — для того, чтобы делать большие шаги в начале прогона.

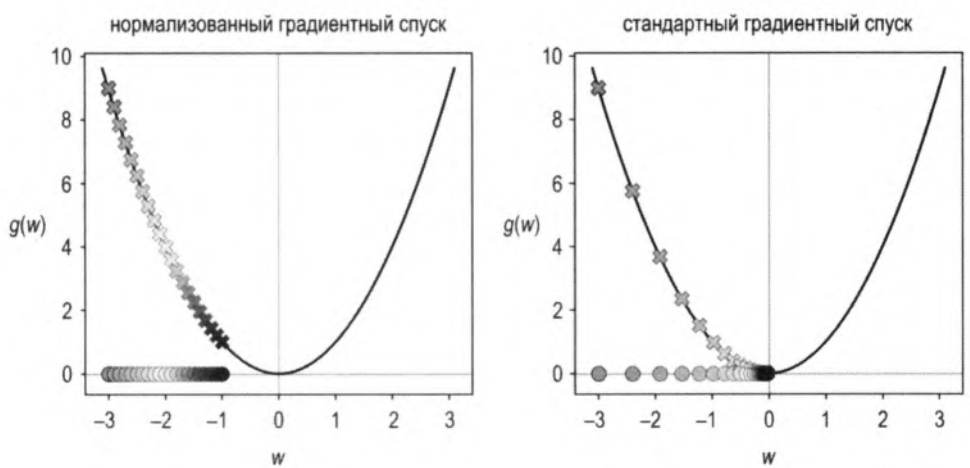


Рис. П1.5. Иллюстрация примера П1.3. В то время как нормализирование градиента ускоряет градиентный спуск вблизи плоских участков функции, где модуль градиента мал, она также не может использовать нередко больший модуль градиента вдали от минимумов функции

П1.3.2. Нормализация модуля покомпонентно

Помните, что градиент — это вектор из N частных производных:

$$\nabla g(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1} g(\mathbf{w}) \\ \frac{\partial}{\partial w_2} g(\mathbf{w}) \\ \vdots \\ \frac{\partial}{\partial w_N} g(\mathbf{w}) \end{bmatrix}, \tag{П1.15}$$

где j-я частная производная $\frac{\partial}{\partial w_j} g(\mathbf{w})$ определяет то, как градиент ведет себя вдоль j-й координатной оси. Посмотрим, что происходит с j-й частичной производной градиента, когда мы нормализуем *полный модуль* градиента:

$$\frac{\frac{\partial}{\partial w_i} g(\mathbf{w})}{\|\nabla g(\mathbf{w})\|_2} = \frac{\frac{\partial}{\partial w_i} g(\mathbf{w})}{\sqrt{\sum_{n=1}^{N} \left(\frac{\partial}{\partial w_n} g(\mathbf{w}) \right)^2}}. \tag{П1.16}$$

Видим, что j-я частная производная нормализуется с помощью суммы модулей каждой частной производной. Это означает, что если j-я частная производная уже сама по себе мала по модулю, то это приведет к стиранию практически всей ее связи с окончательным шагом спуска. Следовательно, нормализация по модулю всего градиента может быть проблематичной во время работы с функциями, содержащими участки, плоские только по некоторым направлениям частных производных, поскольку она *уменьшает* вклад тех самых частных производных, которые мы хотим усилить, игнорируя модуль.

В качестве альтернативы мы можем нормализовать модуль градиента *покомпонентно*. Другими словами, вместо нормализации каждой частной производной по модулю всего градиента мы можем нормализовать каждую частную производную только по отношению к ней самой:

$$\frac{\frac{\partial}{\partial w_i} g(\mathbf{w})}{\sqrt{\left(\frac{\partial}{\partial w_i} g(\mathbf{w}) \right)^2}} = \frac{\frac{\partial}{\partial w_i} g(\mathbf{w})}{\left| \frac{\partial}{\partial w_i} g(\mathbf{w}) \right|} = \text{sign}\left(\frac{\partial}{\partial w_i} g(\mathbf{w}) \right). \tag{П1.17}$$

Поэтому в j-м направлении мы можем записать этот покомпонентно нормализованный шаг градиентного спуска как

$$w_j^k = w_j^{k-1} - \alpha \text{sign}\left(\frac{\partial}{\partial w_i} g\left(\mathbf{w}^{k-1} \right) \right). \tag{П1.18}$$

Тогда мы можем записать весь покомпонентно-нормализованный шаг как

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right). \quad \text{(П1.19)}$$

Здесь знаковая функция действует на градиентный вектор покомпонентно. Мы можем легко вычислить длину одного шага этого покомпонентно-нормализованного шага градиентного спуска (при условии, что все частные производные градиента ненулевые) как

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|-\alpha \text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right)\right\|_2 = \sqrt{N}\alpha. \quad \text{(П1.20)}$$

Обратите внимание: если мы слегка перепишем j-й покомпонентно-нормализованный шаг в уравнении (П1.18) как

$$w_j^k = w_j^{k-1} - \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_i} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \frac{\partial}{\partial w_i} g\left(\mathbf{w}^{k-1}\right), \quad \text{(П1.21)}$$

то сможем интерпретировать наш покомпонентно-нормализованный шаг как шаг стандартного градиентного спуска с индивидуальным значением длины шага:

$$\text{длина шага} = \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_i} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \quad \text{(П1.22)}$$

в расчете на компонент, который настраивается индивидуально на каждом шаге, основываясь на покомпонентном модуле градиента, чтобы обеспечить длину каждого шага в точности равной $\sqrt{N}\alpha$. Действительно, если мы напишем

$$\boldsymbol{\alpha}^{k-1} = \begin{bmatrix} \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_1} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \\ \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_2} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \\ \vdots \\ \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_N} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \end{bmatrix}, \quad \text{(П1.23)}$$

тогда полный покомпонентно-нормализованный шаг спуска можно записать как

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \boldsymbol{\alpha}^{k-1} \circ \nabla g\left(\mathbf{w}^{k-1}\right), \quad \text{(П1.24)}$$

где символ $\circ$ обозначает покомпонентное умножение (*см. разд. П3.2.3*). На практике в знаменатель каждого элемента вектора $\boldsymbol{\alpha}^{k-1}$ добавляют малое $\varepsilon > 0$, чтобы избежать деления на ноль.

Пример П1.4. Полный и покомпонентно-нормализованный градиентные спуски

В этом примере мы используем функцию

$$g(w_1, w_2) = \max(0, \tanh(4w_1 + 4w_2)) + |0,4w_1| + 1, \qquad (П1.25)$$

чтобы показать разницу между шагами полного и покомпонентно-нормализованного градиентного спуска на функции, которая имеет очень узкий плоский участок вдоль только одной размерности ее входа. Здесь эта функция — чьи поверхностные и контурные графики приведены на рис. П1.6 — является очень плоской вдоль размерности w_2 для любого фиксированного значения w_1 и имеет очень узкую долину, ведущую к ее минимумам в размерности w_2, где $w_1 = 0$. Если ини-

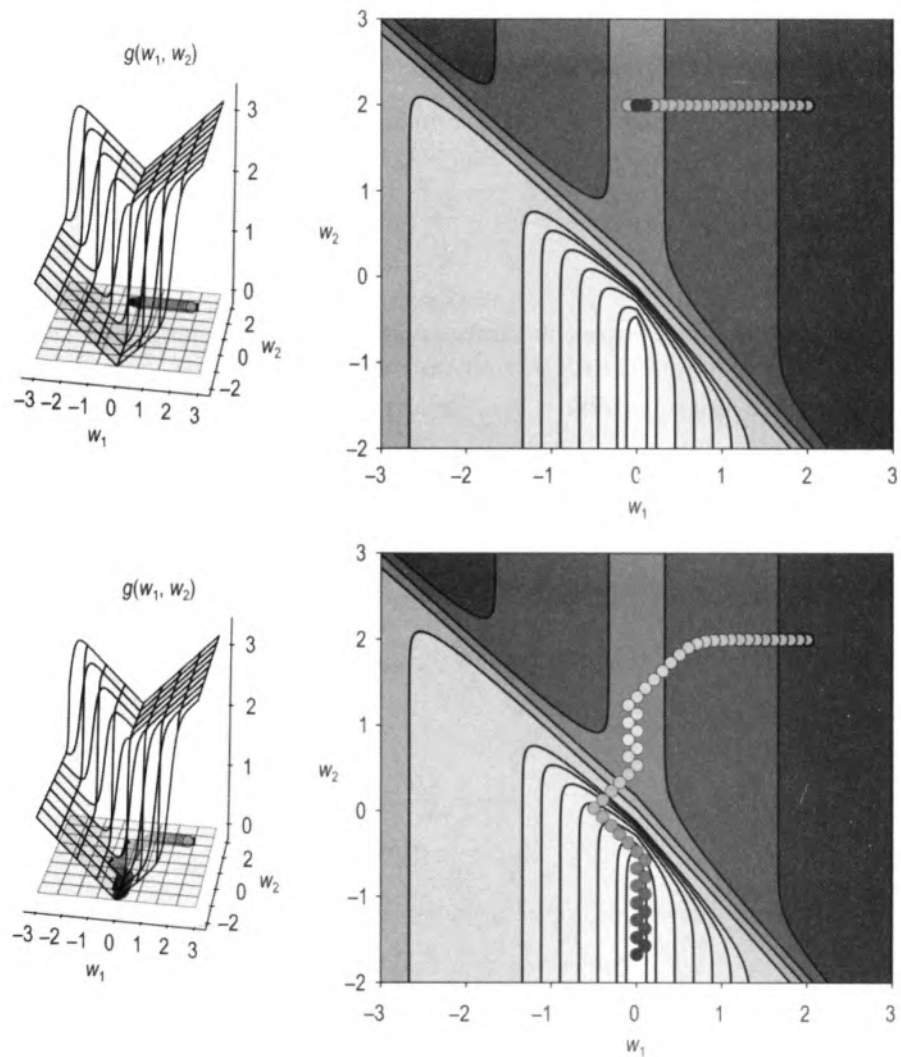


Рис. П1.6. Иллюстрация примера П1.4

циализировать эту функцию в точке, где $w_2 > 2$, то запросто не получится минимизировать ее с помощью стандартного градиентного спуска или полностью нормализованной версии. В последнем случае модуль частной производной в w_2 почти везде равен нулю, и поэтому полная нормализация делает этот вклад меньше и останавливает прогрессию. На рис. П1.6, *сверху* показан результат 1000 шагов полностью нормализованного градиентного спуска, начинающегося в точке $\mathbf{w}^0 = [2 \quad 2]^T$ (точка начала прогона окрашена зеленым цветом, финальная точка — красным). Как видно, прогрессия здесь незначительная.

На рис. П1.6, *снизу* проиллюстрированы результаты использования покомпонентно-нормализованного градиентного спуска, начинающегося с той же инициализации и задействующего ту же длину шага. Здесь нам нужно всего лишь 50 шагов, чтобы добиться значительной прогрессии.

В итоге нормализация модуля градиента — с использованием любого из подходов, подробно описанных ранее, — смягчает проблему медленного ползания стандартного градиентного спуска и позволяет методу пробираться сквозь плоские участки функции с меньшими затратами. К таким участкам относятся плоские участки функции, которые могут привести к локальному минимуму, или участок вокруг седловой точки невыпуклой функции, где стандартный градиентный спуск может остановиться. Однако (как показано в примере П1.3) при нормализировании каждого шага стандартного градиентного спуска мы *укорачиваем* первые несколько шагов прогона, которые обычно являются большими (поскольку случайные инициализации часто далеки от стационарных точек функции). В этом состоит компромисс нормализованного шага по сравнению со схемой стандартного градиентного спуска: мы обмениваем более короткие начальные шаги на более длинные вблизи стационарных точек.

П1.4. Продвинутые градиентные методы

В *разд. П1.2* мы описали понятие импульсно-ускоренного градиентного спуска и выяснили, что он представляет собой естественное средство для решения проблемы зигзагообразного спуска, от которой страдает алгоритм стандартного градиентного спуска, когда работает вдоль длинных узких долин. Как мы видели, направление импульсно-ускоренного спуска $\mathbf{d}^{k-1}$ является просто экспоненциальным средним градиентных направлений спуска, принимающим вид:

$$\begin{aligned} \mathbf{d}^{k-1} &= \beta \mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right); \\ \mathbf{w}^{k} &= \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}, \end{aligned} \tag{П1.26}$$

где $\beta \in [0; 1]$, и в типичной ситуации устанавливается равным $\beta = 0,7$ или больше.

Затем в *разд. П1.3.2* мы увидели, как покомпонентная нормализация направления градиентного спуска помогает справиться с проблемой стандартного градиентного спуска при пересечении плоских участков функции. Там мы увидели, как покомпо-

нентно-нормализованный шаг градиентного спуска принимает форму (для j-й компоненты $\mathbf{w}$)

$$w_j^k = w_j^{k-1} - \alpha \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}}, \tag{П1.27}$$

где на практике в знаменатель часто добавляют малое фиксированное значение $\varepsilon > 0$ во избежание деления на ноль.

Зная, что эти два дополнения в шаг стандартного градиентного спуска помогают решить две фундаментальные проблемы, связанные с направлением спуска, используемым при градиентном спуске, попытаемся *совместить их*, чтобы задействовать оба усовершенствования.

Один из способов совмещения этих двух идей мог бы состоять в том, чтобы покомпонентно нормализовать экспоненциальное среднее направление спуска, вычисляемое в импульсно-ускоренном градиентном спуске. То есть можно вычислить экспоненциальное среднее направление в верхней строке уравнения (П1.8), а затем нормализовать *его* (вместо исходного направления градиентного спуска). С учетом этой идеи мы можем написать обновление для j-й компоненты результирующего направления спуска как

$$d_j^{k-1} = \operatorname{sign}\left(\beta d_j^{k-2} - (1-\beta)\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right). \tag{П1.28}$$

Многие популярные первопорядковые шаги, используемые для настройки моделей машинного обучения, — в частности, те, которые связаны с глубокими нейронными сетями (*см. главу 13*), — сочетают импульс и нормализованный градиентный спуск именно таким образом. Рассмотрим несколько примеров, включая популярные первопорядковые шаги Adam и RMSProp.

Пример П1.5. Адаптивное оценивание импульса (Adam)

Адаптивное оценивание импульса (adaptive moment estimation, Adam) [69] представляет собой покомпонентно нормализованный градиентный шаг, задействующий независимо вычисляемые экспоненциальные средние как для направления спуска, так *и* для его модуля. То есть мы определяем j-ю координату обновленного направления спуска, сначала вычисляя экспоненциальное среднее направления градиентного спуска d_j^k и квадратичный модуль h_j^k отдельно вдоль этой координаты как:

$$\begin{aligned} d_j^{k-1} &= \beta_1 d_j^{k-2} - (1-\beta_1)\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right); \\ h_j^{k-1} &= \beta_2 d_j^{k-2} - (1-\beta_2)\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2, \end{aligned} \tag{П1.29}$$

где β_1 и β_2 — экспоненциальные средние параметры из промежутка [0; 1]. Популярные значения параметров этого шага обновления таковы: $\beta_1 = 0{,}9$ и $\beta_2 = 0{,}999$. Обратите внимание, что, как и в случае любого экспоненциального среднего, оба обновления применяются при $k > 1$ и должны быть инициализированы[2] в первых значениях из ряда, который они соответственно моделируют, т. е.

$$d_j^0 = \frac{\partial}{\partial w_j} g\left(\mathbf{w}^0\right) \text{ и } h_j^0 = \left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^0\right)\right)^2.$$

В таком случае шаг Adam является покомпонентно нормализованным шагом спуска, использующим это экспоненциальное среднее направление спуска и модуль, причем шаг в j-й координате принимает вид

$$w_j^k = w_j^{k-1} - \alpha \frac{d_j^{k-1}}{\sqrt{h_j^{k-1}}}. \qquad (П1.30)$$

Пример П1.6. Распространение ошибки корня из среднеквадратической ошибки (RMSProp)

Этот популярный первопорядковый шаг (Root Mean Squared Error Propagation, RMSProp) является вариантом покомпонентно-нормализованного шага, где вместо нормализации каждой компоненты градиента по ее модулю каждая компонента нормализуется экспоненциальным средним покомпонентно-нормированных модулей предыдущих направлений градиента.

Обозначив через h_j^k экспоненциальное среднее квадратичного модуля j-й частной производной на шаге k, мы имеем

$$h_j^k = \gamma h_j^{k-1} + (1-\gamma)\left(\frac{\partial}{\partial w_i} g\left(\mathbf{w}^{k-1}\right)\right)^2. \qquad (П1.31)$$

Тогда шаг распространения ошибки корня из среднеквадратической ошибки (RMSProp) [70] является покомпонентно нормализованным шагом спуска, использующим это экспоненциальное среднее, причем шаг в j-й координате принимает вид

$$w_j^k = w_j^{k-1} - \alpha \frac{\frac{\partial}{\partial w_i} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{h_j^{k-1}}}. \qquad (П1.32)$$

Популярные значения параметров этого шага обновления таковы: $\gamma = 0{,}9$ и $\alpha = 10^{-2}$.

---

[2] Авторы этого конкретного шага обновления предложили, чтобы каждое экспоненциальное среднее инициализировалось в нуле, т. е. $d_j^0 = 0$ и $h_j^0 = 0$, вместо первого шага в каждой серии, которую они соответственно моделируют. Эта инициализация — наряду со значениями для β_1 и β_2, значения которых в типичной ситуации выбираются больше 0,9, — приводит к тому, что первые несколько шагов обновления этих экспоненциальных средних также "смещаются" к нулю. По этой причине они также задействуют термин "корректировка смещения", дабы уравновесить эту инициализацию.

П1.5. Мини-пакетная оптимизация

В приложениях машинного обучения мы почти всегда ставим задачу минимизировать *сумму P функций одного и того же вида*. Записанная алгебраически, такая стоимость принимает вид

$$g(\mathbf{w}) = \sum_{p=1}^{P} g_p(\mathbf{w}), \tag{П1.33}$$

где g_1, g_2, ..., g_P — это функции одного и того же типа, например выпуклые квадратичные функции (как и в случае линейной регрессии на основе наименьших квадратов, рассмотренной в *главе 5*), все параметризованные с использованием одного и того же множества весов $\mathbf{w}$.

Эта *специальная структура суммирования* позволяет получить простое, но очень эффективное дополнение практически в любую схему локальной оптимизации, именуемое *мини-пакетной оптимизацией*. Мини-пакетная оптимизация чаще всего используется в сочетании с градиентным шагом.

П1.5.1. Простая идея с мощными следствиями

Мотивация для мини-пакетной оптимизации основана на простом вопросе: что произойдет с функцией g из уравнения (П1.33), если вместо одного шага спуска в g, т. е. одного шага спуска во всей сумме функций g_1, g_2, ..., g_P *одновременно*, мы возьмем последовательность P шагов спуска в g_1, g_2, ..., g_P и *последовательно* будем спускаться сначала в g_1, потом в g_2 и т. д., пока наконец не спустимся в g_P? Во многих случаях эта идея может привести к значительно более быстрой оптимизации такой функции. Хотя этот вывод в значительной степени является *эмпирическим*, он может быть интерпретирован в рамках машинного обучения (*см. разд. 7.8*).

Суть этой идеи показана на рис. П1.7 для случая $P = 3$, где мы графически сравниваем идею одновременного выполнения шага спуска в g_1, g_2, ..., g_P против последовательности из P шагов спуска в g_1, затем g_2 и т. д., вплоть до g_P.

Предпринимая первый шаг локального метода для минимизации стоимостной функции g в уравнении (П1.33), мы начинаем с некоторой начальной точки $\mathbf{w}^0$, определяем направление спуска $\mathbf{d}^0$ и переходим к новой точке $\mathbf{w}^1$:

$$\mathbf{w}^1 = \mathbf{w}^0 + \alpha\mathbf{d}^0. \tag{П1.34}$$

По аналогии если бы мы следовали описанной выше мини-пакетной идее, то это повлекло бы за собой последовательность P шагов. Если мы назовем нашу начальную точку $\mathbf{w}^{0,0} = \mathbf{w}^0$, то мы сначала определяем направление спуска $\mathbf{d}^{0,1}$ только в g_1, первую функцию в сумме для g, и делаем шаг в этом направлении следующим образом:

$$\mathbf{w}^{0,1} = \mathbf{w}^{0,0} + \alpha\mathbf{d}^{0,1}. \tag{П1.35}$$



Рис. П1.7. Абстрактная иллюстрация подходов к локальной оптимизации на основе пакетного и мини-пакетного спусков

Далее мы определяем направление спуска $\mathbf{d}^{0,2}$ в g_2, вторую функцию в сумме для g, и делаем шаг в этом направлении

$$\mathbf{w}^{0,2} = \mathbf{w}^{0,1} + \alpha\mathbf{d}^{0,2}, \tag{П1.36}$$

и т. д. Продолжая, мы делаем последовательность из P шагов, где $\mathbf{d}^{0,P}$ — это направление спуска, найденное в g_P, которое принимает следующий вид:

$$\begin{aligned}
&\mathbf{w}^{0,1} = \mathbf{w}^{0,0} + \alpha\mathbf{d}^{0,1};\\
&\mathbf{w}^{0,2} = \mathbf{w}^{0,1} + \alpha\mathbf{d}^{0,2};\\
&\ldots\\
&\mathbf{w}^{0,P} = \mathbf{w}^{0,P-1} + \alpha\mathbf{d}^{0,P};\\
&\ldots\\
&\mathbf{w}^{0,P} = \mathbf{w}^{0,P-1} + \alpha\mathbf{d}^{0,P}.
\end{aligned} \tag{П1.37}$$

Эта последовательность обновлений образует один проход через функции g_1, g_2, ..., g_P и обычно упоминается как *эпоха*. Если мы продолжим и сделаем еще один проход через каждую из P функций по шаблону (П1.37), то выполним вторую эпоху шагов, и т. д.

П1.5.2. Спуск с более крупными размерами мини-пакета

Вместо того чтобы делать P последовательных шагов в отдельных функциях g_p по одной за раз (мини-пакет *размером 1*), мы можем в более общем случае делать меньше шагов в одну эпоху, но выполнять каждый шаг относительно *нескольких* функций g_p, например двух функций одновременно или трех функций одновременно и т. д. С этим легким видоизменением идеи, описанной выше, мы делаем

меньше шагов за эпоху, но выполняем каждый по отношению к более крупным непересекающимся подмножествам функций g_1, g_2, ..., g_P и все равно проносимся через каждую g_p ровно один раз за эпоху.

Размер/мощность используемых подмножеств называется *размером пакета* указанного процесса (мини-пакетная оптимизация с использованием размера пакета 1 также часто называется *стохастической оптимизацией*). Размер пакета, который лучше всего работает на практике — с точки зрения обеспечения наибольшего ускорения оптимизации, — варьируется и часто зависит от задачи.

П1.5.3. Общая результативность

Стоит ли дополнительных усилий компромисс — делать больше шагов за эпоху в случае с мини-пакетным подходом в противовес полному шагу спуска? Как правило, *да*. Часто на практике во время минимизирования функций машинного обучения эпоха мини-пакетных шагов, подобная описанным выше, будет значительно превосходить аналогичный полный шаг спуска. Такая эпоха часто именуется *полнопакетной* или просто *пакетной эпохой* в контексте мини-пакетной оптимизации. Это особенно верно, когда P является большим, обычно равным тысячам или более.

Прототипическое сравнение истории стоимостной функции, задействующей пакет, и соответствующие эпохи мини-пакетной оптимизации, выполненной для той же самой гипотетической функции g (с той же инициализацией), показаны на рис. П1.8. Поскольку с мини-пакетным подходом мы делаем гораздо больше шагов и каждая g_p принимает одинаковый вид, каждая эпоха мини-пакетного подхода обычно превосходит свой полнопакетный аналог. Даже если принять во внимание,

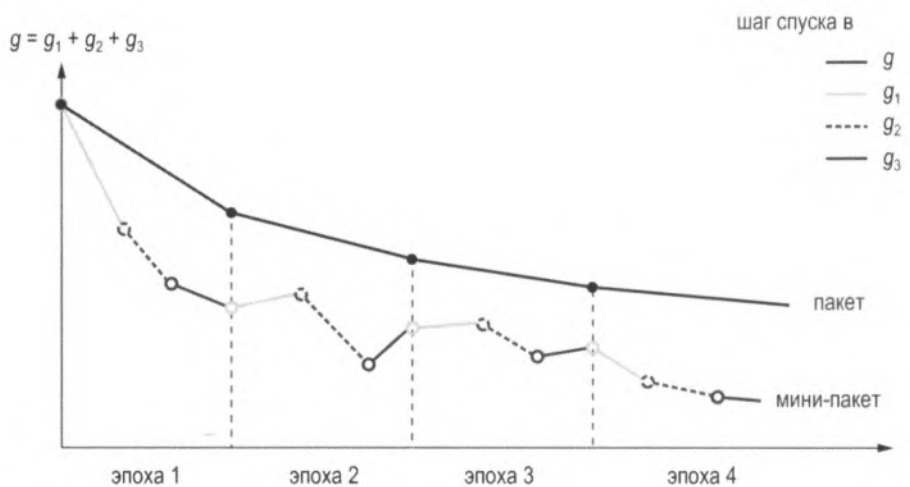


Рис. П1.8. Сравнение истории прототипической стоимостной функции при пакетном и мини-пакетном спуске. Здесь $P = 3$. Эпоха за эпохой мини-пакетный подход тяготеет к опережению полнопакетного шага, достигая точек, более близких к локальным минимумам функций из уравнения (П1.33), быстрее, чем полнопакетный подход

что за эпоху мини-пакетной оптимизации предпринимается гораздо больше шагов спуска, этот метод часто значительно превосходит свой полнопакетный аналог (см., например, упражнение 7.11) — опять же, в особенности когда P является большим.

П1.6. Консервативные правила длины шага

В *разд. 3.5* мы описали, как в приложениях машинного обучения параметр длины шага α для шага градиентного спуска — фиксированный для всех итераций либо убывающий — очень часто определяется методом проб и ошибок. Однако существует возможность математически вывести надлежащие настройки параметра длины шага, которые гарантированно приведут к сходимости алгоритма. Эти варианты шага нередко являются довольно *консервативными*, специально разработаны для принудительного спуска в функции на *каждом шаге* и поэтому обходятся довольно дорого с вычислительной точки зрения. В этом разделе мы кратко рассмотрим такие схемы длины шага, которые будут инересны заинтересованному читателю.

П1.6.1. Градиентный спуск и простые квадратичные функции-заменители

Решающее значение для анализа вариантов параметра длины шага, теоретически способствующего сходимости, для градиентного спуска имеет следующая квадратичная функция

$$h_{\alpha}(\mathbf{w}) = g\left(\mathbf{w}^0\right) + \nabla g\left(\mathbf{w}^0\right)^T\left(\mathbf{w} - \mathbf{w}^0\right) + \frac{1}{2\alpha}\left\|\mathbf{w} - \mathbf{w}^0\right\|_2^2, \qquad \text{(П1.38)}$$

где $\alpha > 0$. Первые два члена в правой части равенства представляют собой аппроксимацию $g(\mathbf{w})$ в точке $\mathbf{w}^0$ с использованием ряда Тейлора первого порядка или, другими словами, формулу для касательной гиперплоскости в указанной точке. Заключительный член в правой части является простейшей квадратичной компонентой, которую можно себе представить, превращающей касательную гиперплоскость — независимо от того, является ли она касательной в точке кривой, которая локально выпукла или вогнута, — в выпуклую и совершенно симметричную квадратичную функцию, кривизна которой контролируется в каждой размерности параметром α. Как и гиперплоскость, эта квадратичная функция по-прежнему является касательной к $g(\mathbf{w})$ в точке $\mathbf{w}^0$, совпадая как с функцией, так и с производными в этой точке.

Что происходит с этой квадратичной функцией, когда мы меняем значение α? На рис. П1.9 проиллюстрирована аппроксимация для двух различных значений α в случае с общей выпуклой функцией. Обратите внимание на связь с α: чем больше значение α, тем шире становится связанная квадратичная функция.

Одна из прекрасных вещей в такой простой квадратичной аппроксимации, как h_{α}, заключается в том, что мы можем легко вычислить уникальный глобальный мини-

мум для него, независимо от значения α, проверив первопорядковое условие оптимальности (*см. разд. 3.2*). Установив его градиент равным нулю, мы имеем

$$\nabla h_\alpha(\mathbf{w}) = \nabla g\left(\mathbf{w}^0\right) + \frac{1}{\alpha}\left(\mathbf{w} - \mathbf{w}^0\right) = 0. \qquad (\text{П1.39})$$

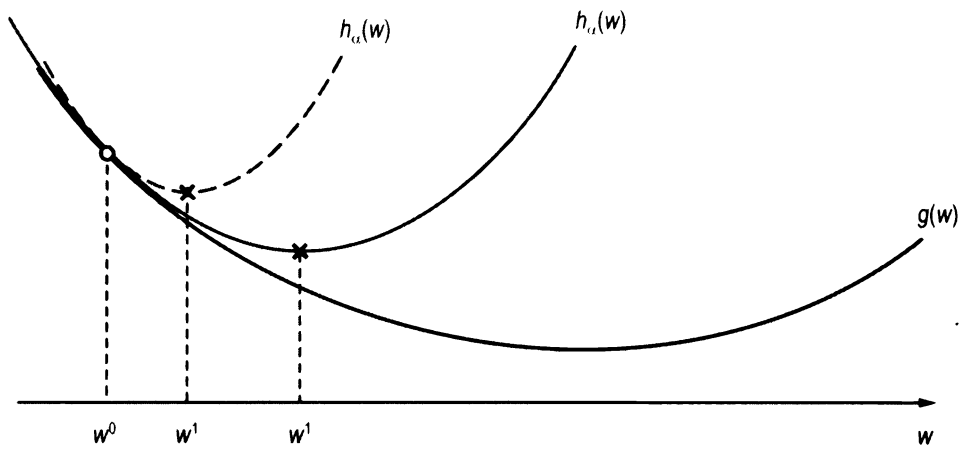


Рис. П1.9. Две квадратичные функции, аппроксимирующие функцию g вокруг w^0, заданную квадратичной аппроксимацией из уравнения (П1.38). Значение α больше у красной квадратичной функции (кривая *в центре*), чем у синей пунктирной (кривая *слева*)

Выполнив преобразования формулы (П1.39), мы можем найти минимизатор h_α, который назовем $\mathbf{w}^1$:

$$\mathbf{w}^1 = \mathbf{w}^0 - \alpha \nabla g\left(\mathbf{w}^0\right). \qquad (\text{П1.40})$$

Таким образом, минимум нашей простой квадратичной аппроксимации — это как раз шаг стандартного градиентного спуска в $\mathbf{w}^0$ с параметром длины шага α.

Если мы продолжим предпринимать шаги таким образом, то k-е обновление будет найдено как минимум простой квадратичной аппроксимации, ассоциированной с предыдущим обновлением $\mathbf{w}^{k-1}$, что подобно

$$\nabla h_\alpha(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T \left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2\alpha}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2, \qquad (\text{П1.41})$$

где минимум снова задается как k-й шаг градиентного спуска:

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right). \qquad (\text{П1.42})$$

Таким образом, наше упражнение с простой квадратичностью демонстрирует альтернативу алгоритму стандартного градиентного спуска (подробно описанному в *разд. 3.5*): мы можем интерпретировать градиентный спуск как алгоритм, использующий линейную аппроксимацию для движения к минимуму функции, или одно-

временно как алгоритм, использующий простые квадратичные аппроксимации для того же самого. Эта новая точка зрения на градиентный спуск: двигаясь по направлению самого крутого спуска гиперплоскости, переходя от ступени к ступени, мы одновременно "прыгаем" вниз по глобальным минимумам этих простых квадратичных аппроксимаций. Обе точки зрения проиллюстрированы одновременно на рис. П1.10.

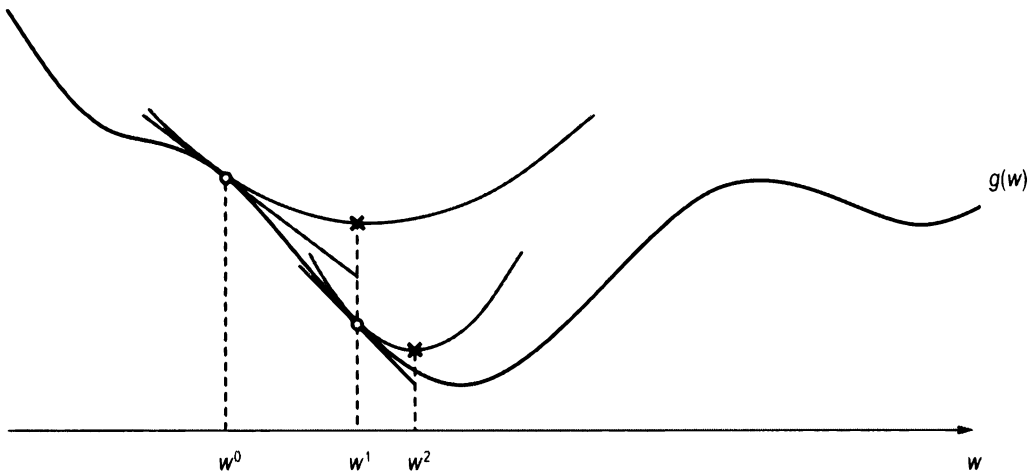


Рис. П1.10. Градиентный спуск можно одновременно рассматривать как использование линейных или простых квадратичных функций-заменителей для отыскания стационарной точки функции *g*. На каждом шаге его длина определяет то, насколько далеко вдоль касательной мы перемещаемся, прежде чем "прыгнуть" обратно на функцию *g*, а также ширину простого квадратичного заменителя, которую мы минимизируем, чтобы достичь той же точки на *g*

П1.6.2. Линейный поиск с отслеживанием в обратном порядке

Поскольку отрицательный градиент является направлением спуска, если мы находимся на шаге $\mathbf{w}^{k-1}$, то — с достаточно малым α — шаг градиентного спуска к $\mathbf{w}^k$ будет уменьшать значение *g*, т. е. $g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right)$. Наша первая точка зрения на градиентный спуск (подробно описанная в *разд. 3.5*) говорит нам о том, что это будет работать, потому что по мере сжатия α мы проходим меньшее расстояние в направлении спуска касательной гиперплоскости в $\mathbf{w}^{k-1}$, и если мы сжимаем это расстояние в достаточной мере, опорная функция также должна уменьшаться в этом направлении. Наша вторая точка зрения дает нам другой, но полностью эквивалентный взгляд на этот вопрос: она говорит нам, что при сжатии α мы *увеличиваем* кривизну связанной квадратичной аппроксимации, показанной в уравнении (П1.41) (минимум которой является точкой, к которой мы будем двигаться), в результате чего минимальная точка квадратичной аппроксимации лежит *выше* функции *g*. Шаг к такой точке должен уменьшить значение функции, потому что в этой

точке квадратичная аппроксимация по определению находится на самом низком уровне и, в частности, ниже точки, касательной к g.

Каким образом можно найти значение α, которое делает именно это в точке $\mathbf{w}^{k-1}$? Если некоторый k-й шаг градиентного спуска равен $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$, то мы хотим определить значение α так, чтобы в $\mathbf{w}^k$ функция была ниже минимума квадратичной функции, т. е. $g\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^k\right)$ (рис. П1.11). Мы могли бы выбрать большое число значений для α и проверить это условие, сохранив то, которое обеспечивает наибольшее уменьшение. Однако этот вариант обходится вычислительно дорого и является несколько громоздкой перспективой. Вместо этого мы тестируем значения α посредством эффективного процесса *бисекции*, которым можно постепенно уменьшать α от некоторого начального значения до тех пор, пока неравенство не будет удовлетворено. Эта процедура, именуемая линейным поиском *с отслеживанием в обратном порядке* (backtracking line search[3]), обычно выполняется следующим образом.

1. Выбрать начальное значение для α, например $\alpha = 1$, и скалярный "демпфирующий коэффициент" $t \in (0; 1)$.
2. Создать кандидатный шаг спуска $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$.

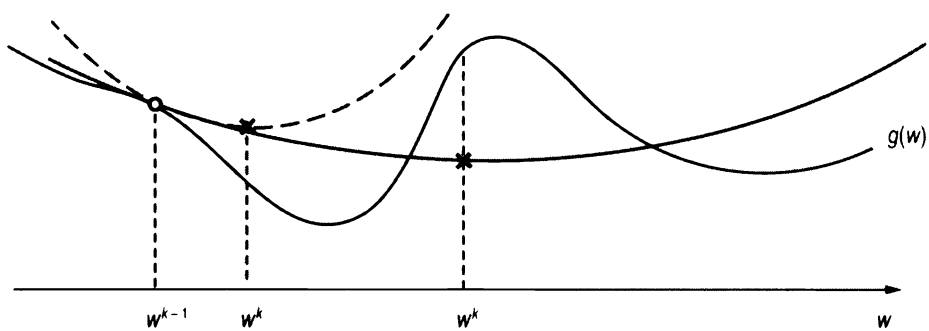


Рис. П1.11. Иллюстрация нашей второй точки зрения на то, как выбрать значение параметра длины шага α, которое гарантированно уменьшит значение опорной функции после одного шага градиентного спуска. Значение α должно уменьшаться до тех пор, пока минимум опорной функции не окажется над минимизируемой функцией. Шаг к такой точке должен уменьшить значение функции, потому что в этой точке квадратичная функция по определению находится на самом низком уровне и, в частности, ниже точки, касательной к g. Здесь значение α, связанное с квадратичной функцией (красная слошная линия), слишком велико, в то время как значение, связанное с квадратичной функцией, нарисованной синей пунктирной линией, достаточно мало, чтобы квадратичная функция лежала выше функции. Шаг (градиентного спуска) к этой точке уменьшает значение функции g

[3] Как вариант — поиск по прямой с отслеживанием в обратном порядке. — *Прим. пер.*

3. Проверить, что $g\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^k\right)$. Если условие выполняется, то выбрать $\mathbf{w}^k$ в качестве следующего шага градиентного спуска; в противном случае уменьшить значение α, выполнив замену $\alpha \leftarrow t\alpha$, и вернуться к шагу 2.

Обратите внимание, что неравенство $g\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^k\right)$ можно записать эквивалентным образом как

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right) - \frac{\alpha}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \qquad (П1.43)$$

путем подключения шага $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)$ к квадратичной функции $h_\alpha\left(\mathbf{w}^k\right)$. Эта эквивалентная версия говорит нам о том, что до тех пор, пока мы не достигли стационарной точки g, член

$$\frac{\alpha}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \qquad (П1.44)$$

всегда будет положительным. Следовательно, отыскание значения α, удовлетворяющего нашему неравенству, означает, что $g\left(\mathbf{w}^k\right)$ будет строго меньше $g\left(\mathbf{w}^{k-1}\right)$.

Идея состоит в том, чтобы сначала опробовать большое значение для α, а затем уменьшать его до тех пор, пока мы не удовлетворим неравенство, которое в иных случаях препятствует громоздкому числу тестов, подлежащих здесь выполнению. Обратите внимание на то, как демпфирующий коэффициент $t \in (0; 1)$ управляет тем, насколько грубо мы отбираем значения α: устанавливая t ближе к 1, мы уменьшаем величину, на которую α сжимается при каждом отказе, что может означать, что для определения адекватного значения α требуется больше вычислений. И наоборот, установка t ближе к 0 значительно сжимает α с каждым отказом, что приводит к более быстрому завершению, но, возможно, за счет вывода малого значения для α (и, следовательно, короткого шага градиентного спуска).

Линейный поиск с отслеживанием в обратном порядке является удобным методом для определения значения длины шага на каждой итерации градиентного спуска и работает "прямо из коробки". Однако каждый градиентный шаг с использованием линейного поиска с отслеживанием в обратном порядке, по сравнению с использованием фиксированного значения длины шага, обычно предусматривает более высокие вычислительные затраты из-за поиска надлежащей длины шага.

П1.6.3. Точный линейный поиск

Как еще можно автоматически настроить значение длины шага α? Возможно, попытаться определить длину шага α, которая минимизирует функцию g непосредственно вдоль k-го шага градиентного спуска $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)$, т. е.

$$\underset{\alpha>0}{\text{минимизировать}}\ g\left(\mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)\right). \qquad (П1.45)$$

Эта идея известна как *точный линейный поиск*. Эту идею можно реализовать посредством подхода, аналогичного подходу на основе линейного поиска с отслеживанием в обратном порядке, который мы видели ранее — последовательно рассматривая все меньшие значения до тех пор, пока не отыщем значение α на k-м шаге $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$ такое, что

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right). \tag{П1.46}$$

П1.6.4. Консервативно оптимальные фиксированные значения длины шага

Предположим, что мы строим простую квадратичную аппроксимацию в виде, приведенном в уравнении (П1.41), и отыскиваем значение α в простой квадратичной аппроксимации таким образом, чтобы оно отражало наибольшую кривизну или изменение в первой производной функции. Установка параметра квадратичной функции α равным этой максимальной кривизне, именуемой *константой Липшица*, означает, что квадратичная аппроксимация будет лежать полностью над функцией везде, кроме ее точки соприкосновения с функцией в $\left(\mathbf{w}^{k-1}, g\left(\mathbf{w}^{k-1}\right)\right)$.

Когда α задано так, что вся квадратичная функция сама по себе лежит выше функции, то минимум квадратичной функции лежит выше функции. Другими словами, наш шаг градиентного спуска должен привести к меньшему вычисленному значению g:

$$g\left(\mathbf{w}^k\right) < h_\alpha\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^{k-1}\right) = g\left(\mathbf{w}^{k-1}\right). \tag{П1.47}$$

Как подробно описано в *разд. 4.1*, информация о кривизне функции находится в ее второй производной. Выражаясь конкретнее, для одновходной функции максимальная кривизна определяется как максимум (по абсолютному значению), взятый ее второй производной, т. е.

$$\max_w \left| \frac{d^2}{dw^2} g(w) \right|. \tag{П1.48}$$

Аналогичным образом для определения максимальной кривизны многовходной функции $g(\mathbf{w})$ мы должны определить максимально возможное собственное значение (по модулю) ее матрицы Гессе или, записав алгебраически, задействуя спектральную норму $\|\cdot\|_2$ (*см. разд. П3.5*)

$$\max_{\mathbf{w}} \left\| \nabla^2 g(\mathbf{w}) \right\|_2. \tag{П1.49}$$

Какой бы сложной ни казалась эта задача, на самом деле для целого ряда распространенных функций машинного обучения она может быть выполнена аналитически, включая линейную регрессию (двухклассовую и многоклассовую), логистическую регрессию, машины опорных векторов, а также мелкие нейронные сети.

После определения эта максимальная кривизна L — или верхняя граница на ней — дает фиксированную[4] длину шага $\alpha = 1/L$, которую можно использовать так, чтобы k-й шаг спуска

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{1}{L}\nabla g\left(\mathbf{w}^{k-1}\right) \tag{П1.51}$$

гарантированно всегда спускался в функции g[5]. С этой длиной шага мы можем инициализировать градиентный спуск в любом месте входной области значений функции, и градиентный спуск будет сходиться к стационарной точке.

Это правило консервативно оптимальной длины шага очень удобно использовать на практике. Однако, как следует из названия, указанное правило по своей природе действительно является консервативным. Следовательно, на практике его можно использовать в качестве эталона для поиска более крупных фиксированных значений длины шага, обеспечивающих сходимость. Другими словами, с вычисленной длиной шага $\alpha = 1/L$ можно легко протестировать более крупные длины шага в виде $\alpha = t/L$ для любой константы $t > 1$. И действительно, в зависимости от задачи значения t в интервале от 1 до 100 могут хорошо работать на практике.

Пример П1.7. Вычисление липшицевой константы одновыходной синусоиды

Давайте вычислим константу Липшица (или максимальную кривизну) синусоидной функции

$$g(w) = \sin(w). \tag{П1.55}$$

---

[4] Если для определения длины шага мы используем локальную кривизну вместо глобальной, то наш соответствующий шаг примет вид

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{1}{\left\|\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right\|_2}\nabla g\left(\mathbf{w}^{k-1}\right), \tag{П1.50}$$

который мы можем интерпретировать как шаг градиентного спуска с самокорректирующейся длиной шага (который самонастраивается на основе локальной кривизны функции g). Метод Ньютона, рассмотренный в *разд. 4.3*, можно рассматривать как расширение этой идеи.

[5] Довольно легко строго показать, что простая квадратичная замещающая касательная к g в точке $\left(\mathbf{w}^{k-1}, g\left(\mathbf{w}^{k-1}\right)\right)$ с $\alpha = 1/L$

$$h_{\frac{1}{L}}(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{L}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2 \tag{П1.52}$$

действительно лежит полностью выше функции g во всех точках. Выписав формулу ряда Тейлора первого порядка для g с центром в $\mathbf{w}^{k-1}$, мы имеем

$$g(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2}\left(\mathbf{w} - \mathbf{w}^{k-1}\right)^T\nabla^2 g(\mathbf{c})\left(\mathbf{w} - \mathbf{w}^{k-1}\right), \tag{П1.53}$$

где $\mathbf{c}$ — это точка на отрезке прямой, соединяющем $\mathbf{w}$ и $\mathbf{w}^{k-1}$. Поскольку $\nabla^2 g \le L\mathbf{I}_{N\times N}$, мы имеем

$$\mathbf{a}^T\nabla^2 g(\mathbf{c})\mathbf{a} \le L\|\mathbf{a}\|_2^2 \tag{П1.54}$$

для всех $\mathbf{a}$ и, в частности, для $\mathbf{a} = \mathbf{w} - \mathbf{w}^{k-1}$, из чего следует $g(\mathbf{w}) \le h_{\frac{1}{L}}(\mathbf{w})$.

Мы можем легко вычислить вторую производную этой функции как

$$\frac{d^2}{dw^2}g(w) = -\sin(w). \tag{П1.56}$$

Максимальное значение, которое может принять эта функция, равно 1. Следовательно, $L = 1$, и поэтому $\alpha = 1/L = 1$ гарантирует спуск на каждом шаге.

Пример П1.8. Вычисление липшицевой константы многовходной квадратичной функции

В этом примере мы рассмотрим вычисление липшицевой константы квадратичной функции

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}, \tag{П1.57}$$

где $a = 1$, $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ и $\mathbf{C} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$.

Здесь гессиана просто равна $\nabla^2 g(\mathbf{w}) = \mathbf{C} + \mathbf{C}^T = 2\mathbf{C}$ для всех входов $\mathbf{w}$, и поскольку собственные значения диагональной матрицы являются как раз ее диагональными элементами, максимальное (по модулю) собственное значение явно равно 4. Таким образом, мы можем установить $L = 4$, получив консервативное оптимальное значение длины шага $\alpha = 1/4$.

П1.6.5. Доказательства сходимости

Для того чтобы подготовить почву для материала этого раздела, будет полезно кратко указать на конкретный набор необходимых условий, удовлетворяемых всеми стоимостными функциями, которые мы стремимся минимизировать в этой книге, поскольку эти условия явно используются в предстоящих доказательствах сходимости. Эти три базовых условия перечислены ниже.

1. Функции имеют кусочно-дифференцируемую первую производную.
2. Функции ограничены снизу.
3. Функции имеют ограниченную кривизну.

Градиентный спуск с фиксированной длиной шага по Липшицу

Поскольку градиент функции g является непрерывным по Липшицу с константой L, из *разд. П1.6.4* мы знаем, что на k-й итерации градиентного спуска мы имеем соответствующую квадратичную верхнюю границу на g формы

$$g(\mathbf{w}) \leq g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{L}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2 \tag{П1.58}$$

для всех $\mathbf{w}$ в области значений g. Теперь, вставив формулу градиентного шага $\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{1}{L}\nabla g\left(\mathbf{w}^{k-1}\right)$ в вышеприведенную формулу и упростив, мы получим

$$g(\mathbf{w}) \leq g\left(\mathbf{w}^{k-1}\right) - \frac{1}{2L}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2. \tag{П1.59}$$

Поскольку $\left\|\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \geq 0$, формула (П1.59) действительно показывает, что последовательность градиентных шагов уменьшается. Для того чтобы показать, что она сходится к стационарной точке, где градиент исчезает, мы вычитаем $g\left(\mathbf{w}^{k-1}\right)$ из обеих частей уравнения (П1.59) и суммируем результат для $1 \leq k \leq K$, получив

$$\sum_{k=1}^{K}\left[g\left(\mathbf{w}^{k}\right)-g\left(\mathbf{w}^{k-1}\right)\right]=g\left(\mathbf{w}^{k}\right)-g\left(\mathbf{w}^{0}\right)\leq-\frac{1}{2L}\sum_{k=1}^{K}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2. \qquad (\text{П1.60})$$

Здесь важно обратить внимание на то, что поскольку g ограничена снизу, приняв $K \to \infty$, мы *должны* иметь:

$$\sum_{k=1}^{\infty}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 < \infty. \qquad (\text{П1.61})$$

Следовательно, из того факта, что вышеприведенная бесконечная сумма должна быть конечной, следует, что по мере того, как $k \to \infty$, мы имеем,

$$\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \to 0, \qquad (\text{П1.62})$$

что означает, что последовательность шагов градиентного спуска с длиной шага, определяемой константой Липшица градиента g, производит последовательность с исчезающим градиентом, сходящуюся к стационарной точке g. Обратите внимание, что мы могли бы выдвинуть тот же самый приведенный выше аргумент, используя любую фиксированную длину шага, меньшую $1/L$.

Градиентный спуск с использованием линейного поиска с отслеживанием в обратном порядке

Из допущения, что g имеет максимальную ограниченную кривизну L, следует, что при любом фиксированном выборе начальной длины шага $\alpha > 0$ и $t \in (0; 1)$ мы всегда можем отыскать целое число n_0 такое, что

$$t^{n_0}\alpha \leq \frac{1}{L}. \qquad (\text{П1.63})$$

Таким образом, найденная за счет возврата назад длина шага на k-м шаге градиентного спуска всегда будет больше этой нижней границы, т. е.

$$\alpha_k \geq t^{n_0}\alpha > 0 \qquad (\text{П1.64})$$

для всех k.

Вспомните из уравнения (П1.43), что путем выполнения процедуры отслеживания в обратном порядке на k-м градиентном шаге мы имеем

$$g\left(\mathbf{w}^{k}\right) \leq g\left(\mathbf{w}^{k-1}\right)-\frac{\alpha_k}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2. \qquad (\text{П1.65})$$

Для того чтобы показать, что последовательность градиентных шагов сходится к стационарной точке g, мы сначала вычитаем $g\left(\mathbf{w}^{k-1}\right)$ из обеих частей уравнения (П1.65) и суммируем результат для $1 \le k \le K$, что дает

$$\sum_{k=1}^{K}\left[g\left(\mathbf{w}^{k}\right)-g\left(\mathbf{w}^{k-1}\right)\right]=g\left(\mathbf{w}^{k}\right)-g\left(\mathbf{w}^{0}\right)\le-\frac{1}{2}\sum_{k=1}^{K}\alpha_{k}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_{2}^{2}. \quad \text{(П1.66)}$$

Поскольку g ограничена снизу, принимая $K \to \infty$, мы должны иметь

$$\sum_{k=1}^{K}\alpha_{k}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_{2}^{2}<\infty. \quad \text{(П1.67)}$$

Теперь, из уравнения (П1.64) мы знаем, что

$$\sum_{k=1}^{K}\alpha_{k}\ge Kt^{n_0}\alpha, \quad \text{(П1.68)}$$

из чего вытекает:

$$\sum_{k=1}^{K}\alpha_{k}=\infty. \quad \text{(П1.69)}$$

Для того чтобы уравнения (П1.67) и (П1.69) соблюдались одновременно, мы *должны* иметь

$$\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_{2}^{2}\to 0 \quad \text{(П1.70)}$$

по мере того, как $k \to \infty$. Это показывает, что последовательность градиентных шагов, определяемая линейным поиском с отслеживанием в обратном порядке, сходится к стационарной точке g.

П1.7. Метод Ньютона, регуляризация и невыпуклые функции

Как мы видели в *разд. 4.3*, метод Ньютона, естественно, не способен надлежаще минимизировать общие невыпуклые функции. В этом разделе описан упорядоченный шаг Ньютона как общий подход к улучшению этой конкретной проблемы (который мы уже видели раньше, хотя и без более подробного объяснения, приводимого нами здесь).

П1.7.1. Поднятие

В *разд. 4.3.3* мы видели, как добавление очень малого положительного значения ε во вторую производную одновходной функции или взвешенной матрицы тождественности формы $\varepsilon\mathbf{I}_{N\times N}$ в гессиан в многовходном случае помогает методу Ньютона избежать численных проблем в плоских участках выпуклой функции. Этот скорректированный шаг метода Ньютона, который принимает вид

$$\mathbf{w}^{k}=\mathbf{w}^{k-1}-\left(\nabla^{2}g\left(\mathbf{w}^{k-1}\right)+\varepsilon\mathbf{I}_{N\times N}\right)^{-1}\nabla g\left(\mathbf{w}^{k-1}\right), \quad \text{(П1.71)}$$

может быть интерпретирован как стационарная точка слегка скорректированной аппроксимации с использованием ряда Тейлора второго порядка с центром в точке $\mathbf{w}^{k-1}$:

$$h(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T \left(\mathbf{w} - \mathbf{w}^{k-1}\right)^T + \\ + \frac{1}{2}\left(\mathbf{w} - \mathbf{w}^{k-1}\right)^T \nabla^2 g\left(\mathbf{w}^{k-1}\right)\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \\ + \frac{\varepsilon}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2. \qquad \text{(П1.72)}$$

Первые три члена (в правой части равенства) по-прежнему представляют собой ряд Тейлора второго порядка в $\mathbf{w}^{k-1}$. В правую часть мы также добавили $\frac{\varepsilon}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2$, выпуклую и совершенно симметричную квадратичную функцию, центрированную в $\mathbf{w}^{k-1}$ с N положительными собственными значениями (каждое равно $\varepsilon/2$). Другими словами, мы имеем сумму двух квадратичных функций. Когда $\mathbf{w}^{k-1}$ находится в невыпуклой или плоской части функции, первая квадратичная функция также является невыпуклой или плоской. Однако вторая *всегда* является выпуклой, и чем больше задано ε, тем больше ее (выпуклая) кривизна. Это означает, что если мы зададим большее ε, то сможем сделать *выпуклой* всю аппроксимацию целиком, заставляя стационарную точку, решение которой мы отыскиваем, быть минимальной, а направление, в котором мы движемся, — одним из гарантированных спусков.

Пример П1.9. Эффект регуляризации

На рис. П1.12 проиллюстрирована регуляризация невыпуклой функции

$$h_1(w_1, w_2) = w_1^2 - w_2^2, \qquad \text{(П1.73)}$$

использующая выпуклый регуляризатор

$$h_2(w_1, w_2) = w_1^2 + w_2^2. \qquad \text{(П1.74)}$$

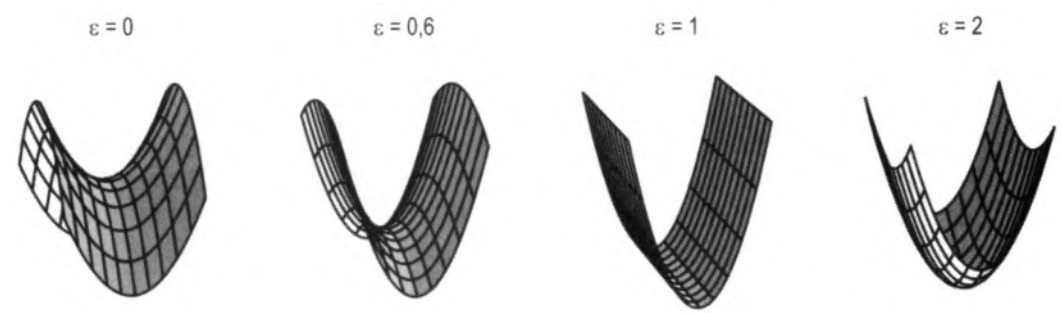


Рис. П1.12. Иллюстрация примера П1.9. Слева направо невыпуклая квадратичная функция медленно превращается в выпуклую функцию посредством взвешенного добавления выпуклой квадратичной функции

В частности, показано, как выглядит результирующая сумма $h_1 + \varepsilon h_2$ при четырех поступательно возрастающих значениях ε, от $\varepsilon = 0$ (слева) до $\varepsilon = 2$ (справа).

Поскольку h_1 не является выпуклой функцией и имеет единственную стационарную точку, которая является седловой точкой в начале координат, добавление h_2 растягивает ее в направлении вниз. Неудивительно, что по мере увеличения ε форма контура суммарной функции все больше и больше определяется h_2. В конце концов, достаточно увеличив ε, мы получим выпуклую поверхность.

Пример П1.10. Минимизация невыпуклой функции

На рис. П1.13 приведены пять регуляризованных ньютоновых шагов (с использованием шага обновления из уравнения (П1.71)), выполненных для минимизации невыпуклой функции

$$g(w) = 2 - e^{-w^2}. \qquad (П1.75)$$

Мы инициализируем алгоритм в точке локальной невыпуклости для этой функции и постепенно увеличиваем ε от $\varepsilon = 0$ (см. рис. П1.13, *сверху*), где метод Ньютона расходится, до $\varepsilon = 4$ (см. рис. П1.13, *снизу*), который сделал аппроксимирующую функцию *значительно выпуклой*, а это привело алгоритм к тому, чтобы делать шаги вниз по склону по направлению к минимуму функции. На рисунке шаги от каждого прогона окрашены от зеленого (первый шаг) до красного (последний шаг), причем регуляризованная аппроксимация второго порядка на каждом шаге окрашена соответствующим образом.

Для того чтобы определить, насколько высоко нам нужно поднять ε, чтобы сделать выпуклой регуляризованную аппроксимацию с использованием ряда Тейлора второго порядка, вспомните из *разд. 4.2*, что квадратичная функция является выпуклой, *если и только если* она имеет все неотрицательные собственные значения. Отсюда следует, что ε должно быть задано больше, чем модуль наименьшего собственного значения гессиана $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$ для того, чтобы регуляризованная квадратичная функция была выпуклой. Для одновходной функции это сводится к ε, более крупному по модулю, чем значение второй производной функции в точке $\mathbf{w}^{k-1}$ (если она является там отрицательной).

В то время как ε в регуляризованном ньютоновом шаге в уравнении (П1.71) обычно устанавливается равным относительно малому значению, интересно отметить, что по мере увеличения ε направление, в котором мы движемся, наклоняется к направлению градиентного спуска в $\mathbf{w}^{k-1}$. Другими словами, когда ε велико, направление, в котором мы движемся при выполнении регуляризованного ньютонова шага, становится направлением градиентного спуска (хотя и с очень малым модулем):

$$-\left(\nabla^2 g\left(\mathbf{w}^{k-1}\right) + \varepsilon \mathbf{I}_{N\times N}\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right) \approx -\left(\varepsilon \mathbf{I}_{N\times N}\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right) = -\frac{1}{\varepsilon} \nabla g\left(\mathbf{w}^{k-1}\right). \qquad (П1.76)$$

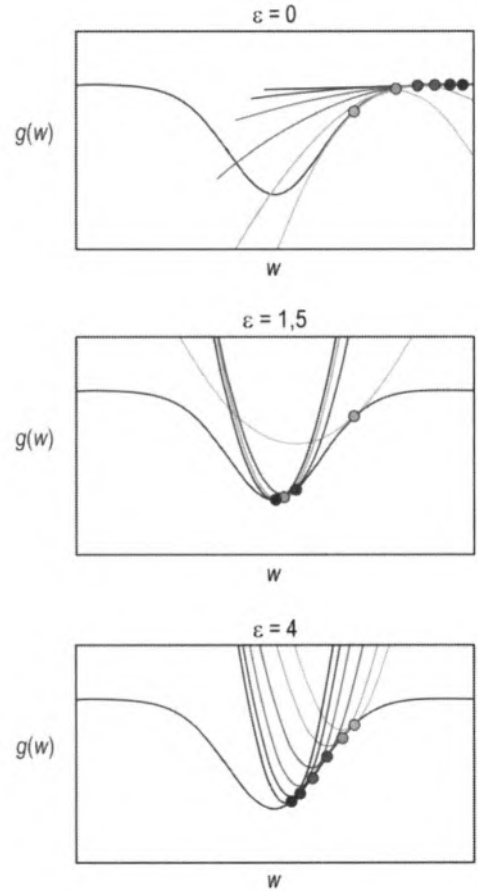


Рис. П1.13. Иллюстрация примера П1.10

П1.8. Безгессиановы методы

Хотя метод Ньютона является мощным методом, который демонстрирует быструю сходимость благодаря задействованию вторых производных, он естественным образом ограничен входной размерностью N общей функции $g(\mathbf{w})$. Выражаясь конкретнее, $N \times N$-матрица Гессе $\nabla^2 g(\mathbf{w})$ с ее N^2 записями естественным образом лимитирует использование метода Ньютона случаями, когда N измеряется тысячами, т. к. такую матрицу трудно даже хранить, когда N больше (не говоря уже о вычислениях с ней).

В этом разделе мы обсудим две вариации метода Ньютона, именуемые *безгессиановыми методами оптимизации*, которые смягчают эту проблему, заменяя гессиан (на каждом шаге метода Ньютона) близкой аппроксимацией, которая не страдает от той же проблемы масштабирования. По этой причине оба подхода обменивают прецизионность каждого шага метода Ньютона на возможность масштабирования базового алгоритма до высокоразмерных входных данных. Первый из этих подхо-

дов является самым простым с концептуальной точки зрения и предусматривает взятие *подвыборки* из гессиана, использующей только часть ее записей. Последний метод, часто именуемый *квазиньютоновым*, предполагает замену гессиана низкоранговой аппроксимацией, которую можно эффективно вычислить.

П1.8.1. Взятие подвыборок из гессиана

Самый простой способ решить проблему масштабирования, присущую методу Ньютона, а именно проблему огромного числа записей в $N \times N$-матрице Гессе $\nabla^2 g(\mathbf{w})$ при увеличении N, — это просто брать *подвыборки* из гессиана. То есть вместо того, чтобы использовать матрицу Гессе целиком, мы используем лишь часть ее записей, устанавливая остальные записи равными нулю. Это, конечно, разреживает мощность полной информации, используемой на каждом ньютоновом шаге, и, следовательно, соответствующую эффективность каждого соответствующего ньютонова шага, но спасает несостоятельный метод при слишком большом N. Существуют самые разные способы, которые позволяют взять подвыборку из гессиана и служат компромиссом между поддержанием информации второго порядка и обеспечением изящного масштабирования вместе с входной размерностью (см., например, [15, 71]).

Одна популярная схема взятия подвыборок предусматривает простое сохранение только диагональных элементов матрицы Гессе, т. е. только N чистых частных вторых производных формы $\frac{\partial^2}{\partial w_n^2} g(w)$ для $n = 1, 2, ..., N$. Это резко сокращает число элементов гессиана и значительно упрощает ньютонов шаг до линейной системы уравнений

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right) \tag{П1.77}$$

и далее до прямомого покомпонентного обновления для каждого $n = 1, 2, ..., N$:

$$w_n^k = w_n^{k-1} - \frac{\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{k-1}\right)}{\frac{\partial^2}{\partial w_n^2} g\left(\mathbf{w}^{k-1}\right)}. \tag{П1.78}$$

Другими словами, сохраняя только диагональные элементы гессиана, мы "отщепляем" каждую координату и отказываемся от системы уравнений для решения. Недостатком, конечно, является то, что мы игнорируем все перекрестно-частные производные, сохраняя только информацию о вторых производных, соответствующую кривизне вдоль каждой входной размерности независимо. Тем не менее этот подвыборочный ньютонов шаг может быть довольно эффективным на практике для минимизации стоимостных функций в машинном обучении (например, см. упражнение 9.8) и может масштабироваться так же изящно, как метод, использующий производные первого порядка, такой как градиентный спуск.

П1.8.2. Методы секущей

При изучении метода Ньютона как алгоритма отыскания на k-м шаге решения уравнения для первой производной $\frac{d}{dw}g(w)=0$ мы формируем ряд Тейлора первого порядка

$$h(w)=\frac{d}{dw}g\left(w^{k-1}\right)+\frac{d^2}{dw^2}g\left(w^{k-1}\right)\left(w-w^{k-1}\right) \tag{П1.79}$$

и пытаемся найти точку, в которой эта линейная функция равна нулю, что задается соответствующим ньютоновым обновлением:

$$w^k=w^{k-1}-\frac{\frac{d}{dw}g\left(w^{k-1}\right)}{\frac{d^2}{dw^2}g\left(w^{k-1}\right)}. \tag{П1.80}$$

Если мы заменим наклон касательной прямой (выраженный здесь второй производной $\frac{d^2}{dw^2}g\left(w^{k-1}\right)$) наклоном, обеспечиваемым тесно связанной[6] секущей линией

$$\frac{d^2}{dw^2}g\left(w^{k-1}\right)\approx\frac{\frac{d}{dw}g\left(w^{k-1}\right)-\frac{d}{dw}g\left(w^{k-2}\right)}{w^{k-1}-w^{k-2}}, \tag{П1.81}$$

то создадим алгоритм, близко родственный методу Ньютона (но без необходимости задействовать вторую производную). Заменив вторую производную этой аппроксимацией на нашем ньютоновом шаге, мы имеем обновление на основе *метода секущей*

$$w^k=w^{k-1}-\frac{\frac{d}{dw}g\left(w^{k-1}\right)}{\frac{\frac{d}{dw}g\left(w^{k-1}\right)-\frac{d}{dw}g\left(w^{k-2}\right)}{w^{k-1}-w^{k-2}}}, \tag{П1.82}$$

который мы можем записать в менее громоздкой манере:

$$w^k=w^{k-1}-\frac{\frac{d}{dw}g\left(w^{k-1}\right)}{s^{k-1}}, \tag{П1.83}$$

---

[6] Вспомните, что производная одновходной функции определяет наклон касательной линии к функции в точке касания. Этот наклон может быть грубо аппроксимирован как наклон ближайшей секущей линии, т. е. линии, проходящей через ту же точку, а также через еще одну ближайшую точку на графике функции (*см. разд. П2.2.1*).

где s^{k-1} используется для обозначения наклона секущей линии:

$$s^{k-1} = \frac{\frac{d}{dw} g\left(w^{k-1}\right) - \frac{d}{dw} g\left(w^{k-2}\right)}{w^{k-1} - w^{k-2}}. \qquad (П1.84)$$

Хотя этот подход менее точен, чем метод Ньютона, который не опирается непосредственно на вторую производную, его все же можно широко использовать для решения уравнения первого порядка и отыскания стационарных точек функции $g(w)$. Этот факт, хотя и довольно несущественный для одновходной функции с $N = 1$, приобретает значительно большее значение, когда метод секущей обобщается на многовходные функции. Это происходит потому, что, как мы уже обсуждали, именно размер матрицы Гессе запрещает использование метода Ньютона для функций с большими значениями N.

Все, что мы обсуждали для общего одновходного случая, также относится к многовходному экземпляру. Обратите внимание, если в уравнении (П1.84) умножить правую и левую части на $w^{k-1} - w^{k-2}$, мы получим

$$s^{k-1}\left(w^{k-1} - w^{k-2}\right) = \frac{d}{dw} g\left(w^{k-1}\right) - \frac{d}{dw} g\left(w^{k-2}\right). \qquad (П1.85)$$

Это уравнение часто называют *одновходным условием секущей.*

Замена каждой компоненты уравнения (П1.85) его многовходным аналогом дает многовходное условие секущей

$$\mathbf{S}^{k-1}\left(\mathbf{w}^{k-1} - \mathbf{w}^{k-2}\right) = \nabla g\left(\mathbf{w}^{k-1}\right) - \nabla g\left(\mathbf{w}^{k-2}\right). \qquad (П1.86)$$

Здесь мы заменили скаляр s^{k-1} его аналогом — $N \times N$ -матрицей $\mathbf{S}^{k-1}$, а одномерные члены w^{k-1}, w^{k-2}, $\frac{d}{dw} g\left(w^{k-1}\right)$ и $\frac{d}{dw} g\left(w^{k-2}\right)$ соответствующими им N-мерными аналогами: $\mathbf{w}^{k-1}$, $\mathbf{w}^{k-2}$, $\nabla g\left(\mathbf{w}^{k-1}\right)$ и $\nabla g\left(\mathbf{w}^{k-2}\right)$. На мгновение предположив, что $\mathbf{S}^{k-1}$ является инвертируемой матрицей, мы также можем выразить условие секущей как

$$\mathbf{w}^{k-1} - \mathbf{w}^{k-2} = \left(\mathbf{S}^{k-1}\right)^{-1}\left(\nabla g\left(\mathbf{w}^{k-1}\right) - \nabla g\left(\mathbf{w}^{k-2}\right)\right). \qquad (П1.87)$$

В любом случае мы видим, что метод секущей требует, чтобы мы нашли решение для матрицы $\mathbf{S}^{k-1}$ или ее обратной матрицы $\left(\mathbf{S}^{k-1}\right)^{-1}$. В отличие от одномерного экземпляра условия секущей в уравнении (П1.85), где каждое обновление имеет единственное решение, с N-мерным случаем в уравнении (П1.86) мы должны решить систему уравнений, которая обычно будет иметь бесконечно много решений, т. к. в матрице $\mathbf{S}^{k-1}$ существует только N уравнений, но N^2 записей для решения.

П1.8.3. Квазиньютоновы методы

Как описано в *разд. 4.3.2*, стандартный ньютонов шаг

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right) \quad (П1.88)$$

является примером шага локальной оптимизации общей формы $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^k$ с направлением спуска, задаваемым так:

$$\mathbf{d}^k = -\left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right). \quad (П1.89)$$

Термин "квазиньютонов метод" — это жаргонное словосочетание, используемое для любого шага спуска формы

$$\mathbf{d}^k = -\left(\mathbf{S}^{k-1}\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right), \quad (П1.90)$$

где истинная матрица Гессе $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$ заменяется аппроксимацией секущей $\mathbf{S}^{k-1}$. Как и в одновходном случае, этот вид обновления — хотя и менее точный на каждом шаге, чем истинный метод Ньютона, — все же может определять эффективный метод локальной оптимизации в зависимости от того, как построена $\mathbf{S}^{k-1}$ (и все это без необходимости прямой информации о вторых производных).

Другими словами, при выполнении квазиньютоновых шагов нам больше не требуется вычислять последовательность матриц Гессе:

$$\nabla^2 g\left(\mathbf{w}^0\right),\ \nabla^2 g\left(\mathbf{w}^1\right),\ \nabla^2 g\left(\mathbf{w}^2\right) \text{ и т. д.} \quad (П1.91)$$

Вместо этого мы строим последовательность соответствующих матриц секущих:

$$\mathbf{S}^1,\ \mathbf{S}^2,\ \mathbf{S}^3 \text{ и т. д.}, \quad (П1.92)$$

как аппроксимацию последовательности гессианов. Для того чтобы построить эту последовательность секущих, обратите внимание на следующее.

- $\mathbf{S}^{k-1}$ должна быть решением условия секущей. Введя для удобства обозначения $\mathbf{a}^k = \mathbf{w}^{k-1} - \mathbf{w}^{k-2}$ и $\mathbf{b}^k = \nabla g\left(\mathbf{w}^{k-1}\right) - \nabla g\left(\mathbf{w}^{k-2}\right)$, условие секущей в уравнении (П1.86) можно записать так: $\mathbf{S}^{k-1}\mathbf{a}^k = \mathbf{b}^k$. Приняв $\mathbf{F}^k = \left(\mathbf{S}^{k-1}\right)^{-1}$, получим, что $\mathbf{a}^k = \mathbf{F}^k \mathbf{b}^k$.
- $\mathbf{S}^{k-1}$ должна быть симметричной. Поскольку гессиан $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$ всегда является симметричной матрицей и в идеале мы хотим, чтобы $\mathbf{S}^{k-1}$ близко имитировала гессиан, разумно ожидать, что $\mathbf{S}^{k-1}$ также будет симметричной.
- Последовательность секущих должна сходиться. По мере развертывания квазиньютонова метода последовательность шагов $\mathbf{w}^{k-1}$ должна сходиться (к минимуму функции g), так же как и последовательность секущих $\mathbf{S}^{k-1}$ (к гессиану, вычисляемому в этом минимуме).

Теперь мы изучим на примере ряд способов построения последовательностей секущих, удовлетворяющих этим условиям. Эти конструкции являются рекурсивными по своей природе и принимают общую форму

$$\mathbf{F}^k = \mathbf{F}^{k-1} + \mathbf{D}^{k-1}, \tag{П1.93}$$

где разностная матрица $\mathbf{D}^{k-1}$ размера $N \times N$ построена *симметричной*, определенного *низкого ранга* и *уменьшающейся* по модулю.

Симметрия $\mathbf{D}^{k-1}$ гарантирует, что если мы инициализируем самый первый $\mathbf{F}^0$ симметричной матрицей (чаще всего матрицей тождественности), то это рекурсивное обновление будет сохранять наше желаемое свойство симметрии (т. к. сумма двух симметричных матриц всегда является симметричной). Аналогично, если $\mathbf{D}^{k-1}$ построена как положительно определенная и инициализация $\mathbf{F}^0$ также положительно определена (как и матрица тождественности), то это свойство наследуется всеми матрицами $\mathbf{F}^k$. Ограничение $\mathbf{D}^{k-1}$ низким рангом делает ее структурно простой и позволяет вычислять ее в замкнутой форме на каждом шаге. Наконец, норма $\mathbf{D}^{k-1}$ должна уменьшаться по мере увеличения k, иначе $\mathbf{F}^k$ просто не будет сходиться.

Пример П1.11. Разностная матрица ранга 1

В этом примере мы опишем одну из простейших рекурсивных формул для $\mathbf{S}^k$ (или, точнее, ее инверсию $\mathbf{F}^k$), где разностная матрица $\mathbf{D}^{k-1}$ в уравнении (П1.93) является одноранговой матрицей внешнего произведения

$$\mathbf{D}^{k-1} = \mathbf{u}\mathbf{u}^T. \tag{П1.94}$$

Сначала приняв допущение о том, что эта форма разностной матрицы действительно удовлетворяет условию секущей, мы можем вернуться от нее *назад*, чтобы определить правильное значение для $\mathbf{u}$.

Подставив $\mathbf{F}^{k-1} + \mathbf{u}\mathbf{u}^T$ вместо $\mathbf{F}^k$ в условие секущей, получаем

$$\left(\mathbf{F}^{k-1} + \mathbf{u}\mathbf{u}^T\right)\mathbf{b}^k = \mathbf{a}^k, \tag{П1.95}$$

или после преобразования:

$$\mathbf{u}\mathbf{u}^T\mathbf{b}^k = \mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k. \tag{П1.96}$$

Умножив обе части равенства на $\left(\mathbf{b}^k\right)^T$:

$$\left(\mathbf{b}^k\right)^T \mathbf{u}\mathbf{u}^T\mathbf{b}^k = \left(\mathbf{b}^k\right)^T \mathbf{a}^k - \left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k \tag{П1.97}$$

и взяв квадратный корень из обеих частей, получим

$$\mathbf{u}^T\mathbf{b}^k = \left(\left(\mathbf{b}^k\right)^T \mathbf{a}^k - \left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k\right)^{\frac{1}{2}}. \tag{П1.98}$$

Подставив значение $\mathbf{u}^T\mathbf{b}^k$ из уравнения (П1.98) в уравнение (П1.96), приходим к желаемой форме для вектора $\mathbf{u}$:

$$\mathbf{u} = \frac{\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k}{\left(\left(\mathbf{b}^k\right)^T \mathbf{a}^k - \left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k\right)^{\frac{1}{2}}} \tag{П1.99}$$

с соответствующей рекурсивной формулой для $\mathbf{F}^k$, задаваемой как

$$\mathbf{F}^k = \mathbf{F}^{k-1} + \frac{\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)^T}{\left(\mathbf{b}^k\right)^T \mathbf{a}^k - \left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k}. \tag{П1.100}$$

Пример П1.12. Метод Дэвидона — Флетчера — Пауэлла (Davidon — Fletcher — Powell, DFP)

Мы можем использовать более сложную структуру для разностной матрицы, построив ее как сумму двух матриц ранга 1:

$$\mathbf{D}^{k-1} = \mathbf{u}\mathbf{u}^T + \mathbf{v}\mathbf{v}^T. \tag{П1.101}$$

Допуская разность ранга 2 между последующими матрицами (в отличие от разности ранга 1), мы кодируем дополнительный уровень сложности в нашу аппроксимацию последующих вычислений обратных гессианов.

Для того чтобы определить надлежащие значения для **u** и **v**, подставим $\mathbf{F}^{k-1} + \mathbf{D}^{k-1}$ вместо $\mathbf{F}^k$ в условие секущей (как мы это сделали в примере П1.11), получив

$$\left(\mathbf{F}^{k-1} + \mathbf{u}\mathbf{u}^T + \mathbf{v}\mathbf{v}^T\right)\mathbf{b}^k = \mathbf{a}^k, \tag{П1.102}$$

или переставив эквивалентным образом

$$\mathbf{u}\mathbf{u}^T\mathbf{b}^k + \mathbf{v}\mathbf{v}^T\mathbf{b}^k = \mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k. \tag{П1.103}$$

Обратите внимание, что здесь мы имеем только *одно* уравнение, и поэтому для наших *двух* неизвестных векторов **u** и **v** существует бесконечно много вариантов. Очень простой, но распространенный способ определения единого множества значений для **u** и **v** состоит в том, чтобы предположить, что первый/второй члены в левой части уравнения (П1.103) равны соответствующим членам в правой части, т. е.

$$\mathbf{u}\mathbf{u}^T\mathbf{b}^k = \mathbf{a}^k \text{ и } \mathbf{v}\mathbf{v}^T\mathbf{b}^k = -\mathbf{F}^{k-1}\mathbf{b}^k. \tag{П1.104}$$

Это добавленное допущение позволяет нам найти решение для допустимой пары **u** и **v**, которое близко отражает метод решения из примера П1.11. Сначала мы умножаем каждый член равенств на $\left(\mathbf{b}^k\right)^T$ и получаем:

$$\left(\mathbf{b}^k\right)^T \mathbf{u}\mathbf{u}^T\mathbf{b}^k = \left(\mathbf{b}^k\right)^T \mathbf{a}^k \text{ и } \left(\mathbf{b}^k\right)^T \mathbf{v}\mathbf{v}^T\mathbf{b}^k = -\left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k. \tag{П1.105}$$

Взяв квадратный корень из обеих частей в обоих уравнениях, получим набор уравнений:

$$\mathbf{u}^T\mathbf{b}^k = \left(\left(\mathbf{b}^k\right)^T \mathbf{a}^k\right)^{\frac{1}{2}} \text{ и } \mathbf{v}^T\mathbf{b}^k = \left(-\left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k\right)^{\frac{1}{2}}. \tag{П1.106}$$

Подставив эти значения вместо $\mathbf{u}^T\mathbf{b}^k$ и $\mathbf{v}^T\mathbf{b}^k$ из уравнения (П1.106) в уравнение (П1.104), имеем

$$\mathbf{u}=\frac{\mathbf{a}^k}{\left(\left(\mathbf{b}^k\right)^T\mathbf{a}^k\right)^{\frac{1}{2}}} \text{ и } \mathbf{v}=\frac{-\mathbf{F}^{k-1}\mathbf{b}^k}{\left(-\left(\mathbf{b}^k\right)^T\mathbf{F}^{k-1}\mathbf{b}^k\right)^{\frac{1}{2}}} \tag{П1.107}$$

с соответствующей рекурсивной формулой для $\mathbf{F}^k$, задаваемой как

$$\mathbf{F}^k=\mathbf{F}^{k-1}+\frac{\mathbf{a}^k\left(\mathbf{a}^k\right)^T}{\left(\mathbf{b}^k\right)^T\mathbf{a}^k}-\frac{\left(\mathbf{F}^{k-1}\mathbf{b}^k\right)\left(\mathbf{F}^{k-1}\mathbf{b}^k\right)^T}{\left(\mathbf{b}^k\right)^T\mathbf{F}^{k-1}\mathbf{b}^k}. \tag{П1.108}$$

Это уравнение именуется методом Дэвидона — Флетчера — Пауэлла (Davidon — Fletcher — Powell, DFP), названным по именам авторов, которые впервые выдвинули это решение [14, 72].

В то время как выведенное здесь обновление относится к обратной матрице $\mathbf{F}^k=\mathbf{S}^{k-1}$, полностью аналогичное рекурсивное выражение может быть сформулировано и для самой $\mathbf{S}^k$, что приводит к обновлению Бройдена — Флетчера — Гольдфарба — Шанно (Broyden — Fletcher — Goldfarb — Shanno, BFGS), названному в честь его авторов [14, 72, 73] (см. упражнение 4.10).

П1.8.4. Квазиньютоновы методы с низким потреблением памяти

В предыдущих примерах мы увидели, как построить последовательность (обратных) матриц секущих посредством рекурсии

$$\left(\mathbf{S}^k\right)^{-1}=\left(\mathbf{S}^{k-1}\right)^{-1}+\mathbf{D}^{k-1}, \tag{П1.109}$$

чтобы заменить истинную последовательность гессианов. Тогда *направление спуска* для *k*-го шага квазиньютонова метода принимает форму

$$\mathbf{d}^k=-\left(\mathbf{S}^{k-1}\right)^{-1}\nabla g\left(\mathbf{w}^{k-1}\right), \tag{П1.110}$$

где $\mathbf{S}^{k-1}$ — это наша аппроксимация $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$. Обратите внимание, однако, на то что в соответствии с уравнением (П1.110) вычисление направления спуска попрежнему включает явную форму $N\times N$-матрицы — инверсию $\mathbf{S}^{k-1}$. Но вспомним, что именно наличие $N\times N$-матрицы Гессе (с ее значениями N^2) заставило нас рассматривать квазиньютоновы методы. Поэтому на первый взгляд кажется, что мы не избежали серьезных проблем масштабирования, связанных с задействованием $N\times N$-матрицы: первоначально это была матрица Гессе, а теперь матрица секущих.

К счастью, мы можем избежать явного построения (обратной) матрицы секущих, сосредоточившись на том, как она действует на градиентный вектор $\nabla g\left(\mathbf{w}^{k-1}\right)$ в уравнении (П1.110). Например, обозначив для удобства нотации $\mathbf{z}^{k-1}=\nabla g\left(\mathbf{w}^{k-1}\right)$

и $\mathbf{F}^k = \mathbf{S}^{k-1}$, направление спуска $\mathbf{d}^k$ с использованием рекурсивного обновления, выведенного в примере П1.11, можно записать как

$$\mathbf{d}^k = -\mathbf{F}^k\mathbf{z}^{k-1} = -\left(\mathbf{F}^{k-1} + \frac{\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)^T}{\left(\mathbf{b}^k\right)^T\mathbf{a}^k - \left(\mathbf{b}^k\right)^T\mathbf{F}^{k-1}\mathbf{b}^k}\right)\mathbf{z}^{k-1} =$$

$$= -\mathbf{F}^{k-1}\mathbf{z}^{k-1} - \frac{\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)^T\mathbf{z}^{k-1}}{\left(\mathbf{b}^k\right)^T\mathbf{a}^k - \left(\mathbf{b}^k\right)^T\mathbf{F}^{k-1}\mathbf{b}^k}\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right). \qquad (П1.111)$$

Обратите внимание, что везде, где матрица $\mathbf{F}^{k-1}$ появляется в финальной строке уравнения (П1.111), она всегда присоединяется к какому-то другому вектору, в данном случае к $\mathbf{b}^k$ и $\mathbf{z}^{k-1}$. Таким образом, если нам каким-то образом удастся вычислить $\mathbf{F}^{k-1}\mathbf{b}^k$ и $\mathbf{F}^{k-1}\mathbf{z}^{k-1}$ непосредственно как векторы, то мы сможем избежать явного формирования $\mathbf{F}^{k-1}$.

Именно здесь приходит на помощь наша конкретная рекурсивная конструкция $\mathbf{F}^k$:

$$\mathbf{F}^k = \mathbf{F}^{k-1} + \mathbf{u}^{k-1}\left(\mathbf{u}^{k-1}\right)^T. \qquad (П1.112)$$

Откатив эту рекурсию полностью назад к $\mathbf{F}^0$ и инициализировав $\mathbf{F}^0 = \mathbf{I}_{N\times N}$, мы можем записать ее эквивалентным образом как

$$\mathbf{F}^k = \mathbf{I}_{N\times N} + \left(\mathbf{u}^0\left(\mathbf{u}^0\right)^T + \ldots + \mathbf{u}^{k-1}\left(\mathbf{u}^{k-1}\right)^T\right). \qquad (П1.113)$$

Умножение любого вектора $\mathbf{t}$ на $\mathbf{F}^k$ можно записать как

$$\mathbf{F}^k\mathbf{t} = \mathbf{I}_{N\times N}\mathbf{t} + \left(\mathbf{u}^0\left(\mathbf{u}^0\right)^T + \ldots + \mathbf{u}^{k-1}\left(\mathbf{u}^{k-1}\right)^T\right)\mathbf{t} \qquad (П1.114)$$

и упрощенно как

$$\mathbf{F}^k\mathbf{t} = \mathbf{t} + \mathbf{u}^0\left(\left(\mathbf{u}^0\right)^T\mathbf{t}\right) + \ldots + \mathbf{u}^{k-1}\left(\left(\mathbf{u}^{k-1}\right)^T\mathbf{t}\right). \qquad (П1.115)$$

Для того чтобы вычислить $\mathbf{F}^k\mathbf{t}$, как записано в уравнении (П1.115), нам нужен только доступ к векторам $\mathbf{t}$, а также ко всем $\mathbf{u}$ от $\mathbf{u}^0$ до $\mathbf{u}^{k-1}$, а также нужно выполнить простое векторное внутреннее умножение.

Мы можем получить очень похожие типы аргументов с любой квазиньютоновой одноранговой или двухранговой формулой обновления, и, следовательно, любая из формул обновления, приведенных в предыдущих примерах, может быть имплементирована таким образом, что нам ни разу не потребуется явно строить секущую или обратную матрицу секущих (см. [14, 72]).

2 ПРИЛОЖЕНИЕ
Производные и автоматическое дифференцирование

П2.1. Введение

Понятие *производной* из дифференциального исчисления является основополагающим для формирования интуитивного и практического понимания методов локальной оптимизации, которые являются рабочей лошадкой машинного обучения. По причине своей критической важности это приложение содержит автономный обзор производных, практическое вычисление производных посредством автоматического дифференцирования (включая набросок *алгоритма обратного распространения*) и аппроксимации с использованием ряда Тейлора.

П2.2. Производная

Производная — это простой инструмент для понимания (непрерывной) математической функции *локально*, т. е. в одной точке и вокруг нее. В этом разделе мы рассмотрим понятие производной в точке и то, как она естественным образом определяет наилучшую *линейную аппроксимацию* (т. е. прямую в двух размерностях, гиперплоскость в более высоких измерениях), которая совпадает с данной функцией в этой точке, а также прямой/гиперплоскости.

П2.2.1. Секущие и касательные

На рис. П2.1 представлен график одновходной функции $g(w)$ вместе с касательной к этой функции в различных точках ее входной области. В любой входной точке *наклон* касательной к этой функции называется ее *производной*. Обратите внимание на то, как в разных входных точках эта величина (т. е. наклон касательной линии), по всей видимости, совпадает с общей локальной крутизной самой базовой функции. Производная естественным образом кодирует эту информацию.

Для того чтобы формально определить производную, мы начнем с рассмотрения *секущей линии*, образованной путем взятия любых двух точек на (одновходной) функции и соединения их прямой линией. Уравнение любой секущей линии, построенной таким образом, легко найти, т. к. ее наклон задается разностью по формуле углового коэффициента, т. е. наклона[1], между двумя точками, и уравнение

[1] То есть по формуле углового коэффициента $(y_2 - y_1)/(x_2 - x_1)$, где $(y_2 - y_1)$ — это величина изменения координаты Y; $(x_2 - x_1)$ — изменение координаты X. В английском языке формула углового

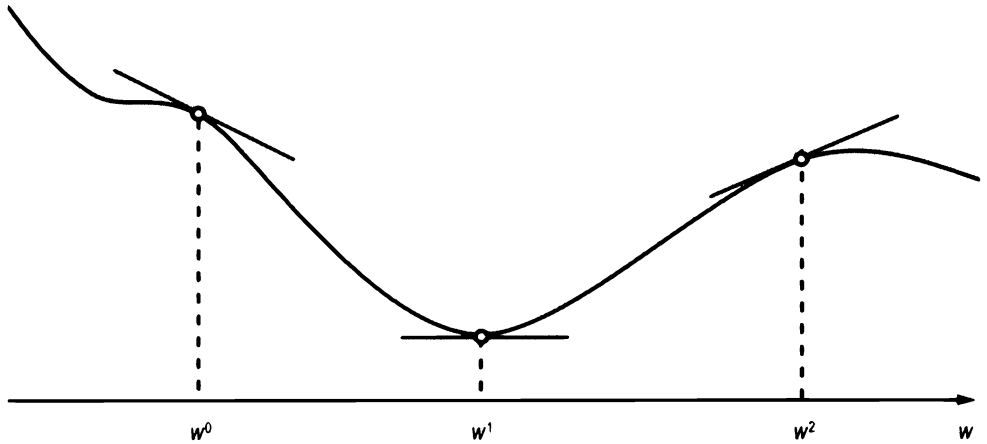


Рис. П2.1. Обобщенная функция с касательной линией, проведенной в различных точках ее входной области определения

любой линии может быть записано с использованием ее наклона и любой точки на этой прямой. Для универсальной одновходной функции g и входных точек w^0 и w^1 секущая линия h проходит через точки $(w^0, g(w^0))$ и $(w^1, g(w^1))$, причем ее наклон определяется как

$$\frac{g(w^1)-g(w^0)}{w^1-w^0}, \tag{П2.1}$$

а ее уравнение определяется как

$$h(w)=g(w^0)+\frac{g(w^1)-g(w^0)}{w^1-w^0}(w-w^0). \tag{П2.2}$$

Теперь представим, что мы фиксируем точку w^0 и начинаем медленно подталкивать к ней точку w^1. По мере того как w^1 становится все ближе и ближе к w^0, секущая линия все больше напоминает касательную линию в w^0. Другими словами, если мы обозначим разницу между w^1 и w^0 через ε:

$$w^1=w^0+\varepsilon, \tag{П2.3}$$

то наклон секущей линии, соединяющей $(w^0, g(w^0))$ с $(w^1, g(w^1))$, заданный как

$$\frac{g(w^1)-g(w^0)}{w^1-w^0}=\frac{g(w^0+\varepsilon)-g(w^0)}{w^0+\varepsilon-w^0}=\frac{g(w^0+\varepsilon)-g(w^0)}{\varepsilon}, \tag{П2.4}$$

совпадает с наклоном касательной линии в точке w^0 (т. е. производной в точке w^0) по мере того, как $|\varepsilon|$ становится бесконечно малым. Обратите внимание, что здесь

коэффициента (или наклона) нередко называется формулой "rise over run", где изменение в y называется *подъемом* (rise), а изменение в x — *пробегом* (run). — *Прим. пер.*

он может быть положительным или отрицательным в зависимости от того, лежит ли w^1 справа или слева от w^0. Для того чтобы производная была определена в w^0, величина в уравнении (П2.4) должна сходиться к тому же значению, которое приближается к нулю, независимо от математического знака (или, что эквивалентно, направления, в котором w^1 приближается к w^0). Например, производная в $w^0 = 0$ определяется для функции $g(w) = \max\left(0, w^2\right)$, т. к.

$$\frac{g\left(w^0 + \varepsilon\right) - g\left(w^0\right)}{\varepsilon} = \frac{\max\left(0, \varepsilon^2\right) - \max\left(0, 0\right)}{\varepsilon} = \frac{\varepsilon^2}{\varepsilon} = \varepsilon, \qquad (П2.5)$$

которая сходится к нулю по мере приближения к нулю. С другой стороны, производная в $w^0 = 0$ не определена для функции $g(w) = \max(0, w)$, т. к.

$$\frac{g\left(w^0 + \varepsilon\right) - g\left(w^0\right)}{\varepsilon} = \frac{\max(0, w)}{\varepsilon} \qquad (П2.6)$$

равно нулю, когда ε приближается к нулю слева, и равно единице, когда ε приближается к нулю справа.

Одна распространенная нотация, используемая для обозначения производной от g в точке w^0, такова

$$\frac{dg\left(w^0\right)}{dw}, \qquad (П2.7)$$

где символ d означает "бесконечно малое изменение в величине". Обратите внимание, что именно этим и выражается дробь в уравнении (П2.4), когда $|\varepsilon|$ бесконечно мало. Распространенная вариация этой нотации записывается так:

$$\frac{d}{dw} g\left(w^0\right). \qquad (П2.8)$$

Существуют также другие нотации, обычно используемые на практике для обозначения производной g в w^0, например $g'\left(w^0\right)$. Наконец, обратите внимание, что с помощью этой нотации уравнение касательной прямой к g в точке w^0 можно записать как

$$h(w) = g\left(w^0\right) + \frac{d}{dw} g\left(w^0\right)\left(w - w^0\right). \qquad (П2.9)$$

П2.2.2. Численное дифференцирование

Для того чтобы избежать необходимости аналитически находить производное выражение в уравнении (П2.4), в особенности когда функция g является довольно сложной, мы можем создать калькулятор *производной*, который оценивает ее производную в определенной пользователем точке w^0, просто установив ε равным некоторому малому числу (в абсолютном значении), например $\varepsilon = 0{,}0001$. Для

большей робастности в нашей аппроксимации производной мы можем заменить "односторонний" наклон секущей в уравнении (П2.8) средним наклоном правой и левой секущих линий, как

$$\frac{d}{dw}g\left(w^0\right) \approx \frac{g\left(w^0+\varepsilon\right)-g\left(w^0-\varepsilon\right)}{2\varepsilon}. \qquad (П2.10)$$

В любом случае чем меньше мы задаем ε, тем лучше становится наша оценка фактического значения производной. Однако задание слишком малого значения ε может привести к *ошибкам округления* из-за того, что числовые значения (независимо от того, получены они из математической функции или нет) могут быть представлены на компьютере только с определенной прецизионностью и что числитель и знаменатель в уравнениях (П2.4) и (П2.10) быстро стремятся к нулю. Эта проблема числовой стабильности не полностью сводит на нет числовое дифференцирование, но об этом стоит знать.

П2.3. Правила производных для элементарных функций и операций

Для некоторых элементарных функций и операций нам не нужно прибегать к численному дифференцированию, т. к. найти точное значение их производной довольно просто. Мы собрали формулы производных для популярных *элементарных функций* и *операций* соответственно в табл. П2.1 и П2.2. Мы привели формальное доказательство для одной элементарной функции из табл. П2.1 (многочлен степени d) и одной элементарной операции из табл. П2.2 (умножение) соответственно в примерах П2.1 и П2.2. Остальные правила производных можно легко подтвердить, следуя схожему алгоритму или обратившись к любому стандартному справочнику по дифференциальному исчислению.

Таблица П2.1. Формулы производных для элементарных функций

Элементарная функция	Уравнение	Производная
Синус	$\sin(w)$	$\cos(w)$
Косинус	$\cos(w)$	$-\sin(w)$
Экспонента	e^w	e^w
Логарифмическая	$\log(w)$	$\frac{1}{w}$
Гиперболический тангенс	$\tanh(w)$	$1-\tanh^2(w)$
Выпрямленная линейная единица (ReLU)	$\max(0, w)$	$\begin{cases}0, & \text{если } w<0;\\ 1, & \text{если } w>0\end{cases}$

Таблица П2.2. Формулы производных для элементарных операций

Элементарная операция	Уравнение	Производная
Добавление константы	$c+g(w)$	$\frac{d}{dw}g(w)$
Умножение на константу	$cg(w)$	$c\frac{d}{dw}g(w)$
Сложение функций	$f(w)+g(w)$	$\frac{d}{dw}f(w)+\frac{d}{dw}g(w)$
Умножение функций	$f(w)g(w)$	$\left[\frac{d}{dw}f(w)\right]g(w)+f(w)\left[\frac{d}{dw}g(w)\right]$
Сложная функция (композиция функций)	$f\left(g(w)\right)$	$\frac{d}{dg}f(g)\frac{d}{dw}g(w)$

Пример П2.1. Производная от общих одночленов

Начиная с одночлена второй степени $g(w)=w^2$, мы можем написать для общего w и малого ε:

$$\frac{g(w+\varepsilon)-g(w)}{\varepsilon}=\frac{(w+\varepsilon)^2-w^2}{\varepsilon}=\frac{(w^2+2w\varepsilon+\varepsilon^2)-w^2}{\varepsilon}=\frac{2w\varepsilon+\varepsilon^2}{\varepsilon}=2w+\varepsilon. \quad (П2.11)$$

Устремив ε к нулю, получим

$$\frac{d}{dw}g(w)=2w. \quad (П2.12)$$

Теперь рассмотрим общий одночлен степени d, $g(w)=w^d$. Здесь нам нужно лишь разложить $(w+\varepsilon)^d$ и соответствующим образом переставить его члены как

$$(w+\varepsilon)^d=\sum_{j=0}^{d}\binom{d}{j}\varepsilon^j w^{d-j}=w^d+d\varepsilon w^{d-1}+\varepsilon^2\sum_{j=2}^{d}\binom{d}{j}\varepsilon^{j-2}w^{d-j}, \quad (П2.13)$$

где

$$\binom{d}{j}=\frac{d!}{j!(d-j)!}. \quad (П2.14)$$

Подключив это разложение в определение производной

$$\frac{(w+\varepsilon)^d-w^d}{\varepsilon}=dw^{d-1}+\varepsilon\sum_{j=2}^{d}\binom{d}{j}\varepsilon^{j-2}w^{d-j}, \quad (П2.15)$$

мы видим, что второй член справа исчезает по мере того, как $\varepsilon\to 0$.

Пример П2.2. Правило произведения

С участием двух функций $f(w)$ и $g(w)$ хотим вычислить

$$\frac{f(w+\varepsilon)g(w+\varepsilon)-f(w)g(w)}{\varepsilon}, \tag{П2.16}$$

по мере того как ε приближается к нулю. Прибавив и вычтя $f(w+\varepsilon)g(w)$ в числителе, получим

$$\frac{f(w+\varepsilon)g(w+\varepsilon)-f(w+\varepsilon)g(w)+f(w+\varepsilon)g(w)-f(w)g(w)}{\varepsilon}, \tag{П2.17}$$

которое затем упрощается до

$$\frac{f(w+\varepsilon)-f(w)}{\varepsilon}g(w)+f(w+\varepsilon)\frac{g(w+\varepsilon)-g(w)}{\varepsilon}. \tag{П2.18}$$

Обратите внимание, что по мере того как $\varepsilon \to 0$, первый и второй члены уравнения (П2.18) уходят соответственно в $\left[\frac{d}{dw}f(w)\right]g(w)$ и $f(w)\left[\frac{d}{dw}g(w)\right]$, вместе давая

$$\frac{d}{dw}[f(w)g(w)]=\left[\frac{d}{dw}f(w)\right]g(w)+f(w)\left[\frac{d}{dw}g(w)\right]. \tag{П2.19}$$

П2.4. Градиент

Градиент — это прямое обобщение понятия производной для многовходной функции $g(w_1, w_2, ..., w_N)$. При трактовке всех входов, кроме первого (т. е. w_1), как фиксированных значений (а *не переменных*) функция g мгновенно сводится к функции одного переменного; а как определять производную для нее, мы уже видели (по отношению к ее единственному входному аргументу w_1). Эта *частная производная*, записанная как $\frac{\partial}{\partial w_1}g(w_1, w_2, ..., w_N)$, определяет наклон касательной гиперплоскости к g в данной точке вдоль первой входной размерности. Повторение этого действия для всех входов в g дает N частных производных (по одной вдоль каждой входной размерности), которые коллективно определяют множество наклонов *касательной гиперплоскости*. Это полностью аналогично одновходному случаю, когда производная обеспечивает наклон касательной.

Для удобства нотации эти частные производные обычно собирают в вектор, именуемый *градиентом* и обозначаемый через $\nabla g(w_1, w_2, ..., w_N)$:

$$\nabla g(w_1, w_2, ..., w_N)=\begin{bmatrix}\frac{\partial}{\partial w_1}g(w_1, w_2, ..., w_N)\\ \frac{\partial}{\partial w_2}g(w_1, w_2, ..., w_N)\\ \vdots \\ \frac{\partial}{\partial w_N}g(w_1, w_2, ..., w_N)\end{bmatrix}. \tag{П2.20}$$

Уложив все N входов (с w_1 по w_N) схожим образом в вектор-столбец

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}, \tag{П2.21}$$

гиперплоскость, касательную к функции $g(\mathbf{w})$ в точке $\left(\mathbf{w}^0, g\left(\mathbf{w}^0\right)\right)$, компактно можно охарактеризовать так:

$$h(\mathbf{w}) = g\left(\mathbf{w}^0\right) + \nabla g\left(\mathbf{w}^0\right)^T \left(\mathbf{w} - \mathbf{w}^0\right), \tag{П2.22}$$

что является прямым обобщением в более высоких размерностях формулы для касательной, определяемой производной одновходной функции, приведенной в уравнении (П2.9), как показано на рис. П2.2.

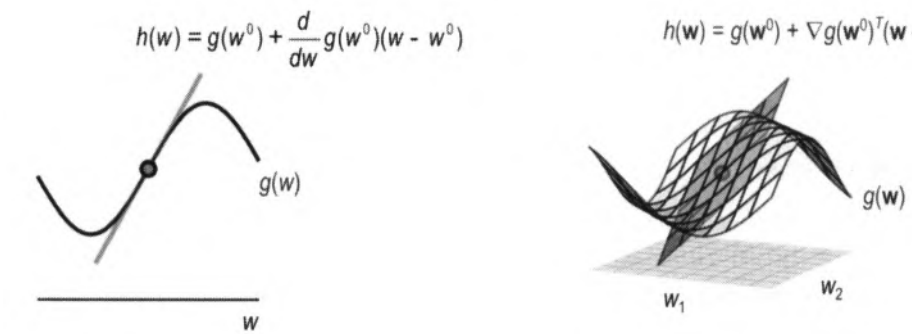


Рис. П2.2. Производная одновходной функции определяет наклон касательной линии в точке (*слева*). Градиент многовходной функции аналогично определяет множество наклонов касательной гиперплоскости в точке (*справа*). Здесь $N = 2$. На каждом рисунке точка касания выделена на функции зеленым кружком

П2.5. Вычислительный граф

Практически любая функция g, выраженная алгебраической формулой, может быть разбита (подобно тому, как физические вещества могут быть разбиты на их атомарные части) на комбинацию элементарных функций (например, $\sin(\cdot)$, $e^{(\cdot)}$, $\log(\cdot)$ и т. д.) и операции (например, сложение, умножение, композиция и т. д.). Одним из очень полезных способов организации элементарной декомпозиции обобщенной функции является так называемый *вычислительный граф*. Вычислительный граф функции g не только помогает нам понять ее анатомию как комбинацию элементарных функций и операций, но и позволяет вычислить функцию программно. Далее мы опишем вычислительный граф, изучив два простых примера, задействующих одновходную и многовходную функции.

Пример П2.3. Вычислительный граф одновходной функции

Возьмем одновходную функцию

$$g(w) = \tanh(w)\cos(w) + \log(w)\,. \qquad (П2.23)$$

Мы можем представить эту функцию, разложив ее на простейшие части, как показано на рис. П2.3, *сверху*. Это графовое изображение наподобие чертежа точно показывает нам то, как g строится из элементарных функций и операций. Прочитаем этот вычислительный граф слева направо, начиная с входного узла, представляющего w, и заканчивая полным вычислением $g(w)$ справа. Каждый желтый узел в графе (за исключением входного узла, который окрашен по-разному в серый цвет) представляет собой одну элементарную функцию или операцию и помечается как таковой. Направленные стрелки, или *рёбра*, соединяющие пары узлов, показывают, как выполняется действие при вычислении $g(w)$.

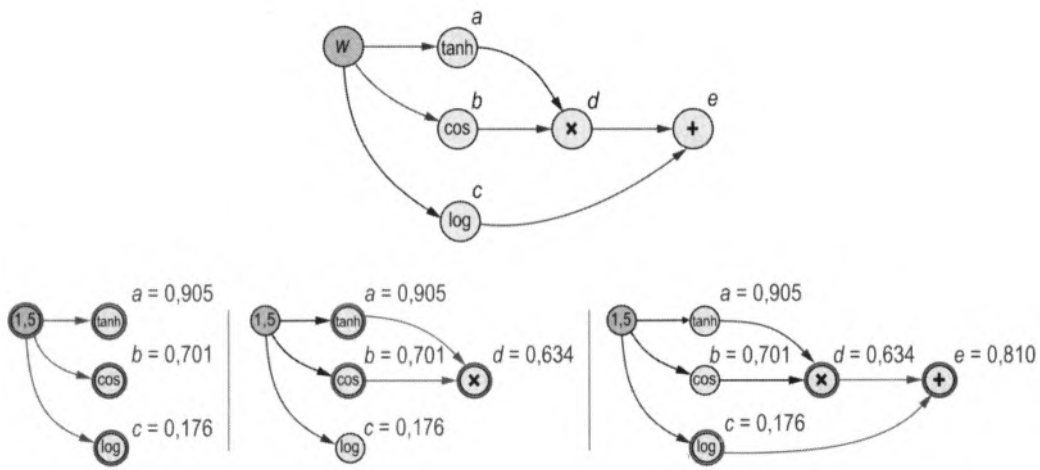


Рис. П2.3. Иллюстрация примера П2.3. Вычислительный граф для одновходной функции, определенной в уравнении (П2.23) (*сверху*). Визуализация потока вычислений через этот граф (*снизу*)

Термины *"родитель"* и *"потомок"* часто используются для описания локальной топологии вычислительных графов для любой пары узлов графа, соединенных посредством направленного ребра. Родительский узел — это место, откуда берется ребро/стрелка, а дочерний узел (ребенок, потомок) — это место, на которое ребро указывает. Поскольку отношения "родитель — потомок" определяются локально, конкретный узел может быть как родителем, так и потомком по отношению к другим узлам графа. Например, в вычислительном графе на рис. П2.3, *сверху* входной узел w (окрашенный в серый цвет) является родительским для узлов a, b и c, в то время как сами a и b являются родительскими для узла d.

Если бы мы выписали формулу каждого дочернего узла в терминах его родителя (родитей), то получили бы следующий список формул:

$$
\begin{aligned}
a &= \tanh(w);\\
b &= \cos(w);\\
c &= \log(w);\\
d &= a \times b;\\
e &= c + d,
\end{aligned}
\qquad \text{(П2.24)}
$$

где финальный узел вычисляет всю функцию, т. е. $e = g(w)$. Еще раз обратите внимание на то, что каждый дочерний узел является функцией своего родителя (родителей). Поэтому мы можем, например, записать a как $a(w)$, т. к. w является (единственным) родительским узлом a, или аналогично записать d как $d(a, b)$, т. к. и a, и b являются родителями d. В то же время если мы разложим определение каждого узла, то обнаружим, что каждый узел действительно является функцией только входа w. Другими словами, мы также можем записать каждый узел как функцию их *общего предка* w, например $d(w) = \tanh(w) \times \cos(w)$.

Вычисление выполняется *вперед* через граф (слева направо) в множествах родительско-дочерних узлов. На рис. П2.3, *снизу* проиллюстрировано, как $g(w)$ вычисляется для конкретного входного значения $w = 1,5$ с помощью вычислительного графа, показанного в верхней части рисунка. Начиная слева, мы подставляем значение $w = 1,5$ во входной узел и вычисляем каждый из дочерних узлов входного узла: $a(1,5) = \tanh(1,5) = 0,905$, $b(1,5) = \cos(1,5) = 0,701$ и $c(1,5) = \log(1,5) = 0,176$, как показано на левой нижней части рисунка, где родительский узел выделен синим цветом, а дочерние — красным. Далее вычисление переходит к любому потомку, чьи родители уже были все вычислены, здесь узел d, как показано на средней части рисунка, где мы использовали ту же раскраску для обозначения связей между родителями и детьми. Обратите внимание, что при вычислении $d(1,5) = a(1,5) \times b(1,5) = 0,634$ нам нужен только доступ к его вычисленным родителям, т. е. $a(1,5)$ и $b(1,5)$, которые мы действительно уже вычислили. Затем мы вычисляем финальный дочерний узел e в графе для нашего входного значения. Опять-таки, чтобы вычислить $e(1,5) = c(1,5) + d(1,5) = 0,810$, нам нужен только доступ к результатам вычисления на его родителях $c(1,5)$ и $d(1,5)$, которые уже были вычислены.

Пример П2.4. Вычислительный граф многовходной функции

Вычислительные графы можно построить схожим образом и для многовходных функций. Например, на рис. П2.4, *сверху* представлен вычислительный граф для простой многовходной квадратичной функции

$$g(w_1, w_2) = w_1^2 + w_2^2. \qquad \text{(П2.25)}$$

Здесь два входа w_1 и w_2, каждый представлен отдельным узлом. Как и в одновходном случае в примере П2.3, вычисления проходят слева направо, или вперед, через граф. Кроме того, как и в одновходном случае, одного прямого прохода через граф достаточно для вычисления любого значения $g(w_1, w_2)$.

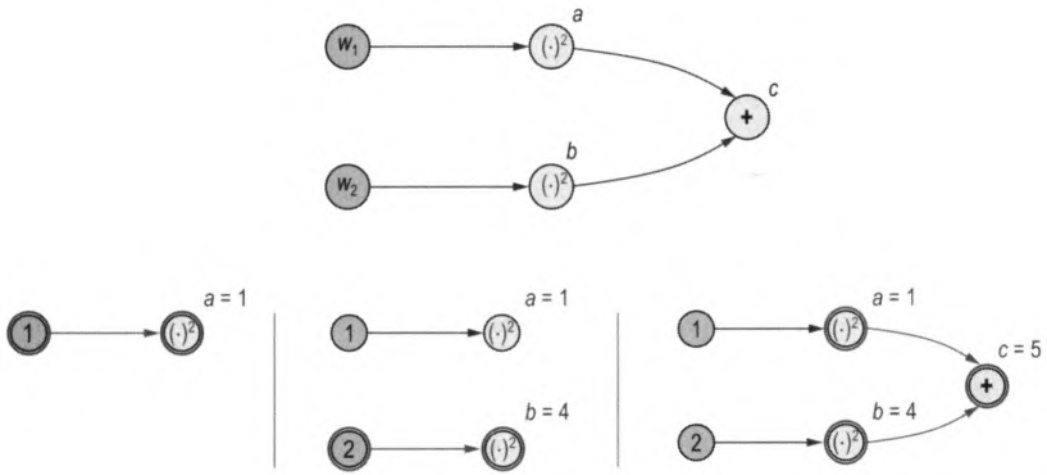


Рис. П2.4. Иллюстрация примера П2.4. Вычислительный граф для многовходной квадратичной функции, определенной в уравнении (П2.25) (*сверху*). Визуализация потока вычислений через этот граф (*снизу*)

На рис. П2.4, *снизу* показано, как $g(w_1, w_2)$ вычисляется для конкретных входных значений $w_1 = 1$ и $w_2 = 2$. Начиная слева, мы сперва подставляем $w_1 = 1$ и вычисляем его единственного ребенка $a(1) = 1^2$ (см. рис. П2.4, *слева снизу*). Затем мы делаем то же самое для второго входного узла, подставляя $w_2 = 2$ и вычисляя его единственного ребенка $b(2) = 2^2$ (см. рис. П2.4, *в центре снизу*). Наконец, мы переходим к последнему потомку, вычисляя его как $c = a(1) + b(2) = 5$, где значения $a(1)$ и $b(2)$ уже были вычислены в предыдущих шагах процесса.

Понятие вычислительного графа является достаточно гибким, т. к. функции можно разложить различными способами. Здесь мы разбили два примера функций на их простейшие, наиболее элементарные части. Однако полезнее разложить более сложные функции (например, полносвязные нейронные сети, как подробно описано в *разд. 13.2*) на вычислительные графы, состоящие из более сложных элементарных строительных блоков, таких как матричное умножение, векторные функции и т. д.

П2.6. Прямой режим автоматического дифференцирования

В предыдущем разделе мы увидели, как представление функции посредством ее вычислительного графа позволяет нам вычислять ее в любой входной точке, проходя граф в *прямом направлении* слева направо, рекурсивно оценивая каждый узел в графе относительно исходного входного значения функции. Вычислительный граф функции можно схожим образом использовать для формирования и вычисления градиента функции путем аналогичного прохода вперед через вычислительный граф функции слева направо, формируя и вычисляя градиент каждого узла относительно исходного входного значения функции. При этом в каждом узле мы также естественным образом вычисляем исходную функцию вместе с вычислением гра-

диента. Этот рекурсивный алгоритм, именуемый *прямым режимом автоматического дифференцирования*, легко программируется и позволяет перенести утомительную рутину вычисления градиента в компьютерную программу, что ускоряет вычисление градиента и делает его надежнее (чем при выполнении вручную), а затем жестко закодированным в компьютерную программу. Более того, в отличие от численного дифференцирования (*см. разд. П2.2.2*), автоматическое дифференцирование обеспечивает точную производную или оценку градиента, а не просто аппроксимацию. Здесь мы описываем прямой режим автоматического дифференцирования путем изучения двух примером, задействовав указанный алгоритм для дифференцирования одновходной и многовходной функций.

Пример П2.5. Дифференцирование одновходной функции в прямом режиме

Возьмем функцию $g(w) = \tanh(w)\cos(w) + \log(w)$, вычислительный граф которой мы ранее проиллюстрировали на рис. П2.3. Для того чтобы оценить производную $\frac{d}{dw}g(w)$, используя прямой режим автоматического дифференцирования, мы проходим этот вычислительный граф почти так же, как мы вычисляли его оценку $g(w)$ в примере П2.3. Начнем с входного узла слева, где w и его производная $\frac{d}{dw}w$ имеют известные значения: значение w выбирается пользователем, а $\frac{d}{dw}w$ всегда тривиально равно 1. Имея в руках форму как узла, так и его производной, мы затем переходим к дочерним узлам входного узла, т. е. узлам a, b и c. Для каждого дочернего узла мы формируем как узел, так и его производную по отношению к входному узлу, т. е. a и $\frac{d}{dw}a$ для первого дочернего узла, b и $\frac{d}{dw}b$ для второго дочернего узла и c и $\frac{d}{dw}c$ для третьего. Эти первые шаги проиллюстрированы на вычислительном графе функции на рис. П2.5, *сверху*, где родительский узел (вход) выделен синим цветом, а дочерние узлы (узлы a, b и c) — красным.

Для формирования каждого дочернего узла и его производной требуются только значения, вычисленные на его родителе (родителях), а также правила производных для элементарных функций и операций, описанные в *разд. П2.3*, которые говорят нам о том, как объединять производные, вычисленные на родителе (родителях), чтобы вычислить дочернюю производную. Для каждого дочернего узла мы можем сформировать их производные по отношению к входу w как:

$$\begin{aligned}\frac{d}{dw}a(w) &= 1 - \tanh^2(w);\\ \frac{d}{dw}b(w) &= -\sin(w);\\ \frac{d}{dw}c(w) &= \frac{1}{w},\end{aligned} \qquad \text{(П2.26)}$$

используя табл. П2.1 в качестве справочного материала.

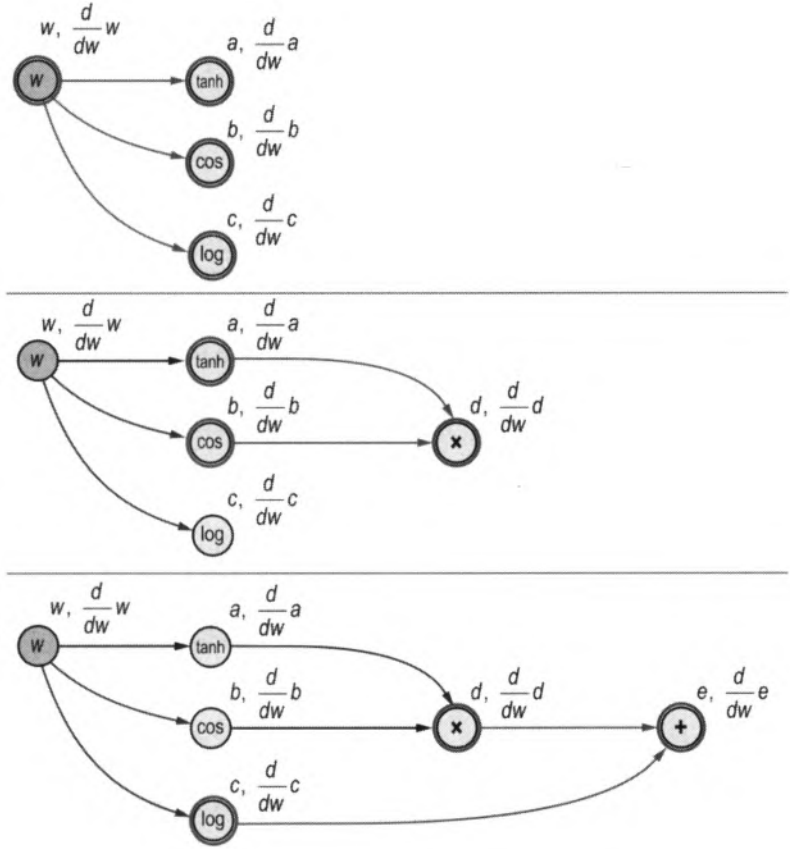


Рис. П2.5. Иллюстрация примера П2.5. Вычисление производной одновходной функции, определенной в уравнении (П2.23), в прямом режиме. На каждом шаге процесса дочерний узел и его производная формируются относительно w и рекурсивно строятся с использованием результатов вычисления узла/производной его родителя (родителей)

Вычислив текущие значения функции/производные относительно w, мы переходим к следующим дочерним узлам графа, где мы увидим, как проявится тот же шаблон в попытках сформировать узлы и их производные относительно w. Рассматривая вычислительный граф на рис. П2.5, *сверху*, видим, что мы уже сформировали всех родителей узла d (т. е. узлы a и b), а также их производные, и, следовательно, переходим к следующему узлу d и формируем $d(w) = a(w) \times b(w)$, используя только что вычисленные значения $a(w)$ и $b(w)$. Для вычисления производной $\frac{d}{dw} d(a, b)$ мы задействуем *цепное правило* и записываем его в терминах производных его родителей/входных данных как:

$$\frac{d}{dw} d(a, b) = \frac{\partial}{\partial a} d(a, b) \times \frac{d}{dw} a(w) + \frac{\partial}{\partial b} d(a, b) \times \frac{d}{dw} b(w). \qquad (П2.27)$$

Обратите внимание, что поскольку мы уже сформировали производные от a и b по отношению к w, нам нужно только вычислить родительско-дочерние производные

$\frac{\partial}{\partial a}d(a,b)$ и $\frac{\partial}{\partial b}d(a,b)$. Поскольку родительско-дочерние связи здесь являются мультипликативными, то обе эти производные можно найти, используя *правило произведения* из табл. П2.2, как:

$$\frac{\partial}{\partial a}d(a,b)=b;$$
$$\frac{\partial}{\partial b}d(a,b)=a. \qquad (П2.28)$$

Объединяя все вместе, получаем полную форму производной в узле d как

$$\frac{d}{dw}d(a,b)=\left(1-\tanh^2(w)\right)\cos(w)-\tanh(w)\sin(w), \qquad (П2.29)$$

которая наглядно проиллюстрирована на рис. П2.5, *в центре*.

Теперь, когда мы вычислили d и его производную, можем приступить к работе над финальным узлом e, который является потомком узлов d и c, определяемым в их терминах как $e(d,c)=d+c$. Вновь используя цепное правило, получаем производную e по отношению к w:

$$\frac{d}{dw}e(d,c)=\frac{\partial}{\partial d}e(d,c)\times\frac{d}{dw}d(w)+\frac{\partial}{\partial c}e(d,c)\times\frac{d}{dw}c(w). \qquad (П2.30)$$

Мы уже вычислили производные d и c по отношению к w, поэтому, вставив 1 как для $\frac{\partial}{\partial d}e(d,c)$, так и для $\frac{\partial}{\partial c}e(d,c)$, мы имеем нашу желаемую производную:

$$\frac{d}{dw}g(w)=\frac{d}{dw}e(d,c)=\left(1-\tanh^2(w)\right)\cos(w)-\tanh(w)\sin(w)+\frac{1}{w}. \qquad (П2.31)$$

Вычислив форму производной в каждом узле графа, мы можем представить себе, как каждая производная добавляется в соответствующий ей узел, оставляя нас с обновленным вычислительным графом для нашей функции $g(w)$, которая теперь может оценивать как значения функции, так и значения производной. Это делается путем подключения значения для w на старте графа и распространения функции и производной вперед по графу слева направо. Вот почему этот метод называется *прямым режимом* автоматического дифференцирования, т. к. все вычисления выполняются продвижением по вычислительному графу вперед. Например, на рис. П2.6 показано, как $g(1,5)$ и $\frac{d}{dw}g(1,5)$ вычисляются вместе с помощью продвижения вперед по вычислительному графу. Обратите внимание, что при вычислении производной таким образом мы естественным образом вычисляем и функцию.

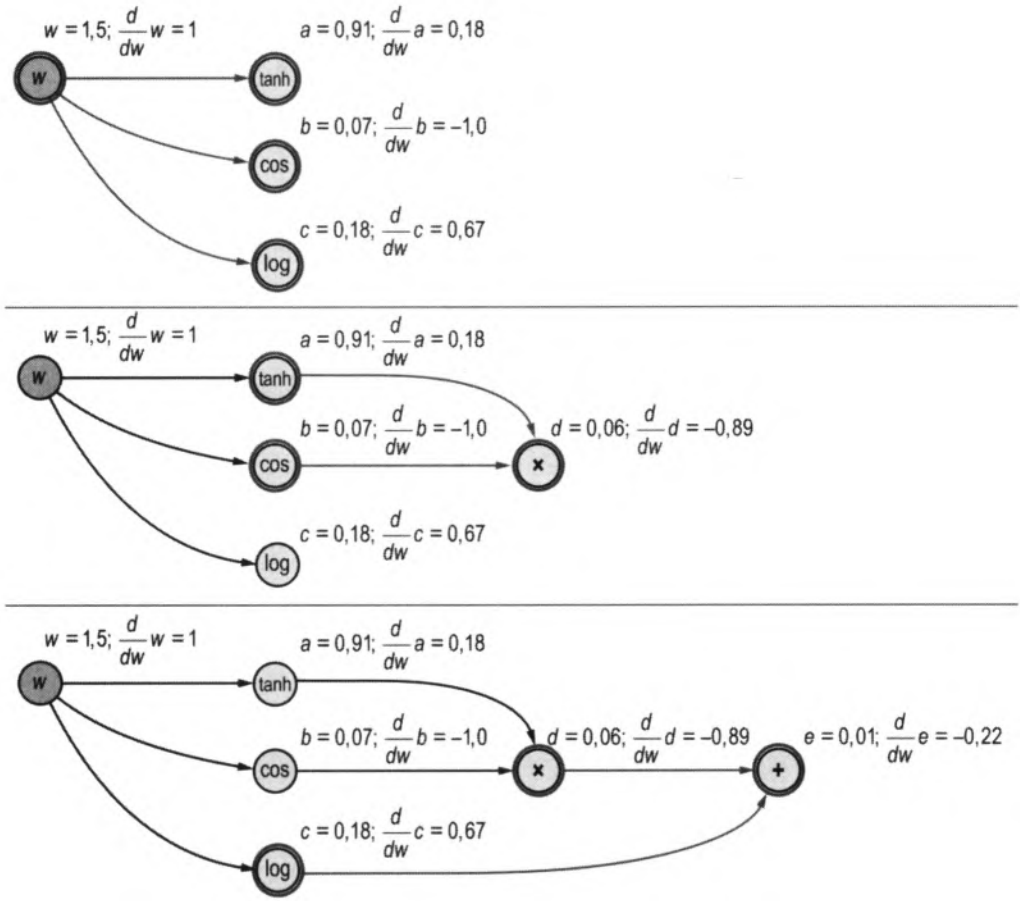


Рис. П2.6. Иллюстрация примера П2.5. Вычисление функции $g(w)$ из уравнения (П2.23) и ее производной во входной точке $w = 1,5$ с использованием прямого режима автоматического дифференцирования

Пример П2.6. Дифференцирование многовходной функции в прямом режиме

Прямой режим автоматического дифференцирования работает схожим образом и для многовходных функций, только теперь мы должны вычислить форму градиента в каждом узле графа (вместо одной производной). Рассмотрим, как это делается с помощью многовходной квадратичной функции $g(w_1, w_2) = w_1^2 + w_2^2$, вычислительный граф которой впервые был показан на рис. П2.4.

Следуя шаблону, изложенному в случае с одновходной функцией в примере П2.5, мы начнем с вычисления градиента в каждом входном узле, которые тривиально равны

$$\nabla \mathbf{w}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ и } \nabla \mathbf{w}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \qquad (П2.32)$$

Затем перейдем к дочерним узлам наших входных узлов, начиная с узла a, где мы вычисляем *градиент a* или, другими словами, частные производные $a(w_1) = w_1^2$ по отношению как к w_1, так и к w_2, в форме

$$\frac{\partial}{\partial w_1} a = 2w_1 \text{ и } \frac{\partial}{\partial w_2} a = 0. \tag{П2.33}$$

Аналогично, мы можем вычислить частные производные $b(w_2) = w_2^2$ по отношению к w_1 и w_2 в форме

$$\frac{\partial}{\partial w_1} b = 0 \text{ и } \frac{\partial}{\partial w_2} b = 2w_2. \tag{П2.34}$$

Эти два шага показаны на рис. П2.7, *слева*. Вычислив форму градиента в узлах a и b, мы можем наконец вычислить градиент в их общем дочернем узле c. Задействуя цепное правило, мы имеем:

$$\begin{aligned} \frac{\partial}{\partial w_1} c &= \frac{\partial}{\partial a} c \frac{\partial}{\partial w_1} a + \frac{\partial}{\partial b} c \frac{\partial}{\partial w_1} b = 1 \times 2w_1 + 1 \times 0 = 2w_1; \\ \frac{\partial}{\partial w_2} c &= \frac{\partial}{\partial a} c \frac{\partial}{\partial w_2} a + \frac{\partial}{\partial b} c \frac{\partial}{\partial w_2} b = 1 \times 0 + 1 \times 2w_2 = 2w_2. \end{aligned} \tag{П2.35}$$

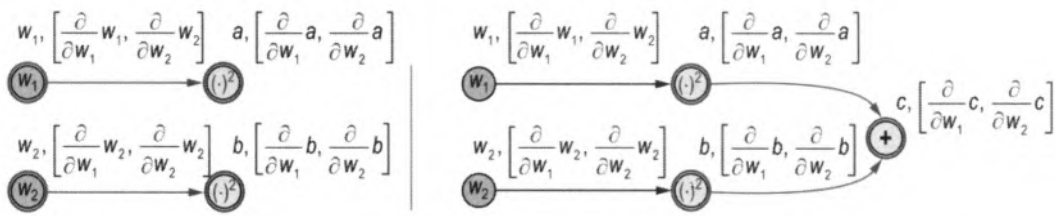


Рис. П2.7. Иллюстрация примера П2.6. Вычисление градиента многовходной функции $g(w_1, w_2)$, определенной в уравнении (П2.25), в прямом режиме

Рассматриваемый здесь прямой дифференциатор дает не алгебраическое описание производной функции, а программную функцию (вычислительный граф), которая может быть использована для вычисления функции и ее производной в любом наборе входных точек. И наоборот, можно построить алгоритм, который задействует базовые правила производных для получения алгебраической производной, но для этого требуется имплементация компьютерной алгебраической системы. Такой калькулятор производных, который работает с производными, используя символические вычисления (т. е. алгебру на компьютере), называется *символическим дифференциатором*. Однако при алгебраическом выражении уравнений они могут становиться довольно громоздкими. Например, довольно сложная на вид функция

$$g(w) = \frac{w^2 \sin(w^2 + w)\cos(w^2 + 1)}{\log(w + 1)} \tag{П2.36}$$

имеет обширную алгебраическую производную

$$\frac{d}{dw}g(w) = \frac{(2w+1)w^2\cos(w^2+1)\cos(w^2+w)}{\log(w+1)} - \frac{w^2\sin(w^2+w)\cos(w^2+1)}{(w+1)\log^2(w+1)} + \\ + \frac{2w\sin(w^2+w)\cos(w^2+1)}{\log(w+1)} - \frac{2w^3\sin(w^2+1)\sin(w^2+w)}{\log(w+1)}. \quad (П2.37)$$

Эта проблема экспоненциально усугубляется вплоть до того, что становится значительной вычислительной нагрузкой при работе с многовходными функциями (с которыми мы обычно имеем дело в машинном обучении). Автоматический (прямой) дифференциатор, который производится вычислительным графом производной вместо алгебраической формы, не имеет этой проблемы.

Наконец, обратите внимание на то, что поскольку вычисления, участвующие в прямом режиме автоматического дифференцирования, выполняются с использованием вычислительного графа заданной функции g, один из инженерных вариантов решения, который необходимо принять, состоит в том, чтобы определиться с конструкцией этого графа и принципами его управления. По существу, у нас есть два варианта: мы можем либо построить вычислительный граф *неявно*, тщательно имплементируя правила элементарной производной, либо разобрать входную функцию g и построить ее вычислительный граф *явно*, как мы это сделали при объяснении метода в этом разделе. Преимущество неявного построения графа состоит в том, что соответствующий калькулятор является облегченным (т. к. нет необходимости хранить граф) и строится просто. С другой стороны, реализация калькулятора, который строит вычислительные графы явно, требует дополнительных инструментов (например, лексического анализатора, или парсера), но позволяет упростить вычисление производных более высокого порядка. Это происходит потому, что в последнем случае дифференциатор берет функцию для дифференцирования и трактует ее как вычислительный граф, а также часто выводит вычислительный граф ее производной, который затем может быть подключен обратно в тот же алгоритм для создания производных второго порядка и т. д.

П2.7. Обратный режим автоматического дифференцирования

В то время как прямой режим автоматического дифференцирования, введенный в предыдущем разделе, обеспечивает прекрасный программный способ вычисления производных, он бывает неэффективным для многих видов многовходных функций (в особенности в машинном обучении, включая *полносвязные сети*). Это обусловлено тем, что в то время как большинство узлов в вычислительном графе многовходной функции могут принимать только несколько входов, мы вычисляем *полный градиент* по отношению *ко всем входам* в каждом узле. В результате вычислительные ресурсы значительно растрачиваются, т. к. мы знаем, что частная производная любого узла по отношению к исходному входу функции, которую она не принимает, всегда будет равна нулю.

Указанное очевидное расточительство является мотивацией для того, что называется *обратным режимом автоматической дифференциации* или, на жаргоне машинного обучения, *алгоритмом обратного распространения*. В обратном режиме мы начинаем с обхода вычислительного графа функции в прямом направлении (начиная с входных узлов и двигаясь слева направо), вычисляя форму только тех частных производных, которые необходимы в каждом узле вычислительного графа (игнорируя частные производные, которые всегда будут равны нулю). После того как этот прямой проход завершен, мы затем (начиная с конечного узла (узлов) в графе) двигаемся в *обратном* направлении и перемещаемся по графу *назад*, собирая и комбинируя ранее вычисленные частные производные, чтобы соответствующим образом построить градиент. Хотя это означает, что мы должны явно построить вычислительный граф (и его сохранить), компромисс с потраченными впустую вычислениями часто стоит того, когда речь идет о функциях машинного обучения. Это делает *обратный режим* более популярным вариантом выбора (чем прямой режим, подробно описанный в предыдущем разделе) в приложениях машинного обучения и, в частности, является именно тем автоматическим дифференциатором, который был реализован в библиотеке autograd (автоматическом дифференциаторе на основе Python, который мы рекомендуем использовать с этой книгой).

Пример П2.7. Обратное дфференцирование многовходной функции

В примере П2.6 предыдущего раздела мы описали способ вычисления градиента функции $g(w_1, w_2) = w_1^2 + w_2^2$ с помощью прямого автоматического дифференцирования. Обратите внимание на то, как при вычислении полного градиента в каждом узле вычислительного графа этой функции мы выполняли несколько расточительных вычислений: всякий раз, когда частные производные брались относительно входных данных, не принимаемых узлом (через его сложную сеть родительско-дочерних связей, ведущих обратно к первоначальному входу функции), мы по умолчанию знали, что эта частная производная всегда будет равна нулю. Например, частная производная $\frac{\partial}{\partial w_2} a = 0$, т. к. a не является функцией первоначального входа w_2.

На рис. П2.8 мы перерисовали вычислительный граф этой квадратичной функции. При этом градиент выражен в терминах частных производных в каждом узле и все нулевые частные производные отмечены соответствующим образом. Рассматривая этот граф, мы видим, что значительная часть частных производных тривиально равна нулю. Тривиальные нули такого рода приводят к расточительству вычислительных ресурсов, когда мы их формируем, проходя *вперед* через граф.

Эта проблема становится гораздо более серьезной с многовходными функциями, которые принимают большее число входных переменных. Например, на рис. П2.9 приведен вычислительный граф аналогичной квадратичной функции, принимающей четыре входа: $g(w_1, w_2, w_3, w_4) = w_1^2 + w_2^2 + w_3^2 + w_4^2$. В этом случае более *половины* всех записей градиента в узлах этого графа равны нулю вследствие того, что

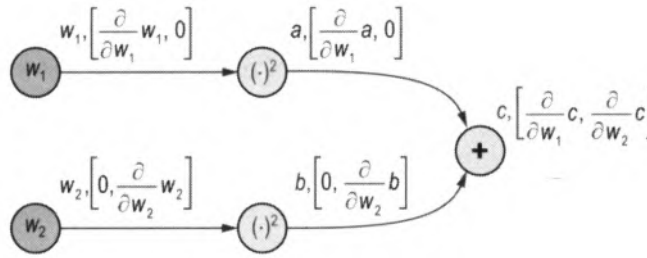


Рис. П2.8. Иллюстрация примера П2.7. Вычислительный граф квадратичной функции, описанный в тексте, в котором нулевые частные производные отмечены соответствующим образом

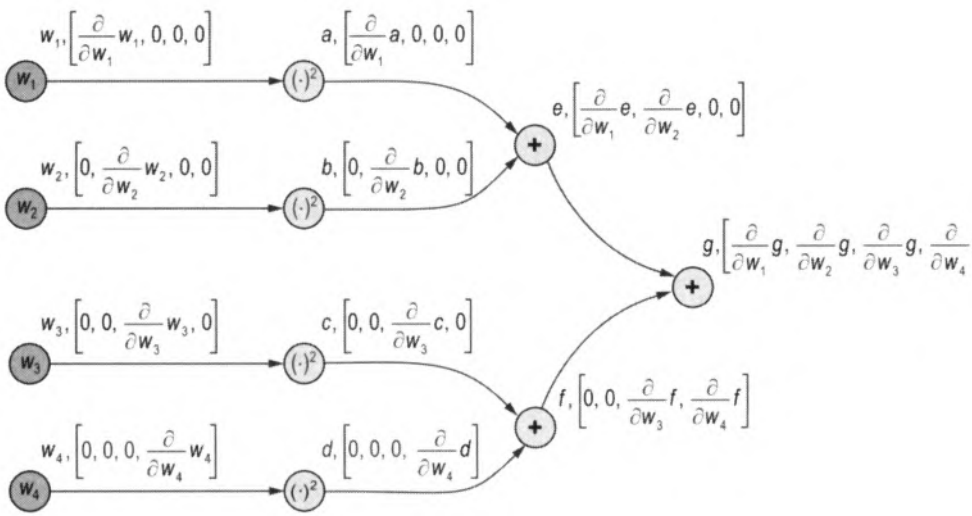


Рис. П2.9. Иллюстрация примера П2.7. Вычислительный граф простой четырехвходной квадратичной функции, градиент которой выражен в каждом узле в терминах частных производных. Здесь более половины вычисленных частных производных являются тривиальными нулями

некоторые узлы не являются функциями определенных входов, а, следовательно, их частные производные всегда равны нулю.

В целях устранения этой неэффективности автоматическое дифференцирование может также выполняться в обратном режиме, который состоит из *прямого* и *обратного* (или обратнонаправленного) *проходов* через вычислительный граф функции. В прямом проходе обратного режима мы пересекаем вычислительный граф в прямом направлении (слева направо) рекурсивно, как и в прямом режиме, только в каждом узле мы вычисляем частные производные каждого дочернего узла *лишь* по отношению к его родителю (родителям), а *не* полный градиент по отношению к входному значению функции.

Это показано для квадратичной функции $g(w_1, w_2) = w_1^2 + w_2^2$ на рис. П2.10, *сверху*. В левой части приведено вычисление частных производных для дочерних узлов a и b (окрашенных красным цветом), которые берутся только по отношению к их ро-

дителям (которые здесь соответственно равны w_1 и w_2 и окрашены синим цветом). В правой верхней части рисунка проиллюстрировано следующее вычисление в прямом проходе, частные производные, вычисленные в дочернем узле c (окрашены красным цветом) по отношению к его родителям a и b (окрашены синим цветом).

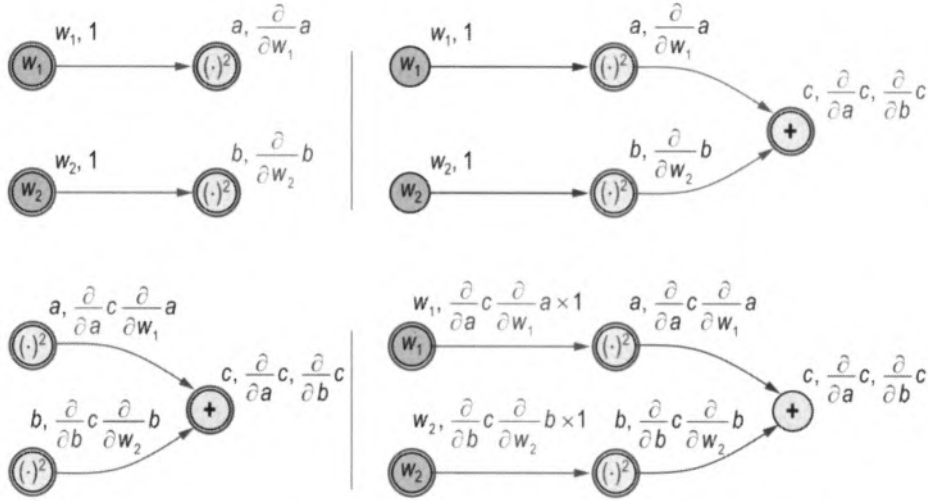


Рис. П2.10. Иллюстрация примера П2.7. Прямой (*сверху*) и обратный (*снизу*) проходы для простой двухвходной квадратичной функции

После того как прямой проход завершен, мы меняем курс и проходим через вычислительный граф в *обратном* направлении, начиная с финального узла и рекурсивно продвигаясь назад справа налево до тех пор, пока не достигнем входных узлов. На каждом этапе этого процесса мы обновляем частную производную каждого родителя, умножая ее на частную производную его дочернего узла по отношению к этому родителю (когда родительский узел имеет несколько потомков, аккумулированные частные производные должны быть сложены, а не умножены, поскольку это то, что требует цепное правило). Когда обратный проход будет завершен, мы рекурсивно построим градиент функции по отношению ко всем ее входам.

Обратный проход показан для двухвходной квадратичной функции на рис. П2.10, *снизу*. Начиная с финального узла в нашем прямом проходе, т. е. узла c, мы видим, что c имеет двух родителей: a и b. Поэтому мы обновляем производную в точке a путем (левого) умножения $\frac{\partial}{\partial a}c$, получив $\frac{\partial}{\partial a}c\frac{\partial}{\partial w_1}a$, и схожим образом обновляем производную в b путем (левого) умножения $\frac{\partial}{\partial b}c$, получив $\frac{\partial}{\partial b}c\frac{\partial}{\partial w_2}b$. Затем мы повторяем эту процедуру рекурсивно с детьми узлов a и b, закончив частными производными $\frac{\partial}{\partial a}c\frac{\partial}{\partial w_1}a\frac{\partial}{\partial w_1}w_1 = \frac{\partial}{\partial a}c\frac{\partial}{\partial w_1}a$ и $\frac{\partial}{\partial b}c\frac{\partial}{\partial w_2}b\frac{\partial}{\partial w_2}w_2 = \frac{\partial}{\partial b}c\frac{\partial}{\partial w_2}b$. Это как раз

те две частные производные полного градиента квадратичной функции по отношению к ее входам w_1 и w_2.

П2.8. Производные более высокого порядка

В предыдущих разделах мы видели, как можно эффективно вычислить производную функций, состоящих из элементарных строительных блоков, и что эти производные сами являются функциями, построенными из элементарных строительных блоков. По этой причине мы можем также вычислять производные от производных, обычно именуемых *производными более высокого порядка*, которые являются предметом этого раздела.

П2.8.1. Производные более высокого порядка одновходных функций

Разведаем понятие производных более высокого порядка одновходных функций, обратившись к простому примеру.

Пример П2.8. Производные более высокого порядка

Для вычисления производной второго порядка функции

$$g(w) = w^4 \tag{П2.38}$$

мы сначала найдем ее производную первого порядка как

$$\frac{d}{dw} g(w) = 4w^3, \tag{П2.39}$$

а потом дифференцируем ее еще раз, чтобы получить

$$\frac{d}{dw}\left(\frac{d}{dw} g(w)\right) = 12w^2. \tag{П2.40}$$

Взятие производной результирующей функции еще раз дает производную третьего порядка функции g:

$$\frac{d}{dw}\left(\frac{d}{dw}\left(\frac{d}{dw} g(w)\right)\right) = 24w. \tag{П2.41}$$

Схожим образом, мы можем вычислить первые три производные функции

$$g(w) = \cos(3w) + w^2 + w^3 \tag{П2.42}$$

явно как:

$$\begin{aligned} \frac{d}{dw} g(w) &= -3\sin(3w) + 2w + 3w^2; \\ \frac{d}{dw}\left(\frac{d}{dw} g(w)\right) &= -9\cos(3w) + 2 + 6w; \\ \frac{d}{dw}\left(\frac{d}{dw}\left(\frac{d}{dw} g(w)\right)\right) &= 27\sin(3w) + 6. \end{aligned} \tag{П2.43}$$

Производные более высокого порядка также часто выражаются с использованием более компактной нотации, чем приведенная в примере П2.8. Например, вторая производная очень часто обозначается компактнее с помощью следующей нотации:

$$\frac{d^{(2)}}{dw^{(2)}}g(w) = \frac{d}{dw}\left(\frac{d}{dw}g(w)\right). \tag{П2.44}$$

Аналогично третья производная часто обозначается компактнее как

$$\frac{d^{(3)}}{dw^{(3)}}g(w) = \frac{d}{dw}\left(\frac{d}{dw}\left(\frac{d}{dw}g(w)\right)\right), \tag{П2.45}$$

и в общем случае n-я производная записывается как

$$\frac{d^{(n)}}{dw^{(n)}}g(w). \tag{П2.46}$$

П2.8.2. Производные более высокого порядка многовходных функций

Мы видели, как градиент многовходной функции представляет собой коллекцию частных производных

$$\nabla g(w_1, w_2, ..., w_N) = \begin{bmatrix} \frac{\partial}{\partial w_1}g(w_1, w_2, ..., w_N) \\ \frac{\partial}{\partial w_2}g(w_1, w_2, ..., w_N) \\ \vdots \\ \frac{\partial}{\partial w_N}g(w_1, w_2, ..., w_N) \end{bmatrix}, \tag{П2.47}$$

где градиент содержит n-ю частную производную $\frac{\partial}{\partial w_n}g(w_1, w_2, ..., w_N)$ в качестве своей n-й записи. Эта частная производная (как и сама исходная функция) является функцией, принимающей N входов, сокращенно **w**, которые мы можем дифференцировать вдоль каждой входной оси. Например, мы можем взять m-ю частную производную $\frac{\partial}{\partial w_n}g(w_1, w_2, ..., w_N)$ как

$$\frac{\partial}{\partial w_m}\frac{\partial}{\partial w_n}g(w_1, w_2, ..., w_N). \tag{П2.48}$$

Эта форма представляет собой производную *второго порядка*. Сколько из них имеет функция g? Поскольку каждая из N производных первого порядка функции g (каждая из которых является функцией N входов) имеет N частных производных, $g(\mathbf{w})$ имеет в общей сложности N^2 производных второго порядка.

Как и в случае с понятием градиента, это большое множество производных второго порядка в типичной ситуации организовано конкретным образом, чтобы их было легче коммуницировать и вычислять. *Гессиан*, который нотационно записывается как $\nabla^2 g(\mathbf{w})$, — это $N \times N$ -матрица производных второго порядка, (m, n)-й элемент которой равен $\frac{\partial}{\partial w_m}\frac{\partial}{\partial w_n} g(\mathbf{w})$, или $\frac{\partial}{\partial w_m}\frac{\partial}{\partial w_n} g$ для краткости. Полная матрица Гессе записывается как

$$\nabla^2 g(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1}\frac{\partial}{\partial w_1} g & \frac{\partial}{\partial w_1}\frac{\partial}{\partial w_2} g & \cdots & \frac{\partial}{\partial w_1}\frac{\partial}{\partial w_N} g \\ \frac{\partial}{\partial w_2}\frac{\partial}{\partial w_1} g & \frac{\partial}{\partial w_2}\frac{\partial}{\partial w_2} g & \cdots & \frac{\partial}{\partial w_2}\frac{\partial}{\partial w_N} g \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial w_N}\frac{\partial}{\partial w_1} g & \frac{\partial}{\partial w_N}\frac{\partial}{\partial w_2} g & \cdots & \frac{\partial}{\partial w_N}\frac{\partial}{\partial w_N} g \end{bmatrix}. \quad \text{(П2.49)}$$

Кроме того, поскольку практически всегда бывает, что $\frac{\partial}{\partial w_m}\frac{\partial}{\partial w_n} g = \frac{\partial}{\partial w_n}\frac{\partial}{\partial w_m} g$, в особенности с такого рода функциями, используемыми в машинном обучении, гессиан всегда является *симметричной* матрицей.

Число частных производных многовходной функции растет *экспоненциально* вместе с порядком. Мы только что увидели, что функция, принимающая N входов, имеет N^2 производных второго порядка. В общем случае такая функция имеет N^D частных производных порядка D.

П2.9. Ряд Тейлора

В этом разделе мы опишем *ряд Тейлора* — фундаментальный инструмент дифференциального исчисления, который критически важен для локальной оптимизации первого и второго порядков (как подробно описано в *главах 3* и *4*). Мы начнем с того, что выведем это важнейшее понятие для одновходных функций, а затем обобщим его на случай многовходных функций.

П2.9.1. Линейная аппроксимация — это только начало

Мы начали наше обсуждение производных в *разд. П2.2*, определив производную в точке как наклон касательной к заданной входной функции. Для функции $g(w)$ мы формально описали касательную линию в точке w^0 как

$$h(w) = g\left(w^0\right) + \frac{d}{dw} g\left(w^0\right)\left(w - w^0\right), \quad \text{(П2.50)}$$

причем наклон здесь задается производной $\frac{d}{dw}g\left(w^0\right)$. Обоснование для того, чтобы с самого начала рассматривать касательную линию, является довольно простым: локально (близко к точке w^0) касательная линия выглядит невероятно похожей на функцию, и поэтому если мы хотим лучше понять g вблизи w^0, то можем с тем же успехом посмотреть там на касательную. Это намного упрощает нашу жизнь, потому что линия (по сравнению с обобщенной функцией g) всегда является сравнительно простым объектом, и поэтому понять касательную всегда просто.

Если мы изучим форму нашей касательной $h(w)$ внимательно, то сможем математически точно определить, как она совпадает с функцией g. Прежде всего, обратите внимание на то, что касательная принимает то же значение, что и функция g в точке w^0, т. е.

$$h\left(w^0\right)=g\left(w^0\right)+\frac{d}{dw}g\left(w^0\right)\left(w^0-w^0\right)=g\left(w^0\right). \qquad (П2.51)$$

Далее, обратите внимание, что значения первой производной этих двух функций также совпадают. То есть если мы возьмем первую производную h по отношению к w, то увидим, что

$$\frac{d}{dw}h\left(w^0\right)=\frac{d}{dw}g\left(w^0\right). \qquad (П2.52)$$

Одним словом, когда касательная линия h совпадает с g так, что в w^0 значения функции *и* производной равны, мы можем записать:

$$\begin{aligned} h\left(w^0\right)&=g\left(w^0\right);\\ \frac{d}{dw}h\left(w^0\right)&=\frac{d}{dw}g\left(w^0\right). \end{aligned} \qquad (П2.53)$$

А что, если мы перевернем эту идею и попытаемся найти линию, удовлетворяющую этим двум свойствам? Другими словами, мы начинаем с общей линии

$$h(w)=a_0+a_1\left(w-w^0\right) \qquad (П2.54)$$

с неизвестными коэффициентами a_0 и a_1 и мы хотим определить правильное значение для этих коэффициентов так, чтобы линия удовлетворяла двум критериям в уравнении (П2.53). Оба критерия образуют *систему уравнений*, которую мы можем решить для правильных значений a_0 и a_1. Вычислив левую часть каждого из них, где h — это наша общая линия в уравнении (П2.54), мы в итоге получаем тривиальную систему уравнений для решения обоих неизвестных одновременно:

$$\begin{aligned} h\left(w^0\right)&=a_0=g\left(w^0\right);\\ \frac{d}{dw}h\left(w^0\right)&=a_1=\frac{d}{dw}g\left(w^0\right), \end{aligned} \qquad (П2.55)$$

и, как и следовало ожидать, коэффициенты в точности совпадают с коэффициентами касательной линии.

П2.9.2. От касательной линии к касательной квадратичной функции

Учитывая, что функциональные и производные значения касательной линии совпадают со значениями ее опорной функции, можем ли мы добиться лучшего? Можем ли мы найти простую функцию, которая совпадает со значением функции, значением первой производной и значением второй производной в точке w^0? Другими словами, можно ли определить простую функцию h, удовлетворяющую следующим *трем* условиям?

$$\begin{aligned} h\left(w^0\right) &= g\left(w^0\right); \\ \frac{d}{dw} h\left(w^0\right) &= \frac{d}{dw} g\left(w^0\right); \\ \frac{d^{(2)}}{dw^{(2)}} h\left(w^0\right) &= \frac{d^{(2)}}{dw^{(2)}} g\left(w^0\right). \end{aligned} \tag{П2.56}$$

Обратите внимание, что (касательная) линия h может удовлетворять только первым двум из этих свойств и никогда третьему, т. к. она является многочленом первой степени и $\frac{d^2}{dw^2} h(w) = 0$ для всех w. Из этого факта следует, что нам нужен по крайней мере многочлен второй степени, чтобы удовлетворить всем трем критериям. Начиная с общего многочлена второй степени

$$h(w) = a_0 + a_1\left(w - w^0\right) + a_2\left(w - w^0\right)^2 \tag{П2.57}$$

с неизвестными коэффициентами a_0, a_1 и a_2, мы можем оценить левую часть каждого критерия в уравнении (П2.56), образовав систему из трех уравнений, и найти решение для этих коэффициентов.

$$\begin{aligned} h\left(w^0\right) &= a_0 = g\left(w^0\right); \\ \frac{d}{dw} h\left(w^0\right) &= a_1 = \frac{d}{dw} g\left(w^0\right); \\ \frac{d^{(2)}}{dw^{(2)}} h\left(w^0\right) &= 2a_2 = \frac{d^{(2)}}{dw^{(2)}} g\left(w^0\right). \end{aligned} \tag{П2.58}$$

После определения всех наших коэффициентов (в терминах производных функции g в w^0) мы имеем многочлен второй степени, удовлетворяющий трем желаемым критериям:

$$h(w) = g\left(w^0\right) + \frac{d}{dw} g\left(w^0\right)\left(w - w^0\right) + \frac{1}{2}\frac{d^2}{dw^2} g\left(w^0\right)\left(w - w^0\right)^2. \tag{П2.59}$$

Это представляет собой один шаг за пределы касательной линии (аппроксимирующей квадратичной функции), но обратите внимание, что первые два члена действительно являются касательной линией как таковой. Такая квадратичная аппроксимация совпадает с обобщенной функцией g вблизи точки w^0 гораздо точнее, чем касательная линия, как показано на рис. П2.11.

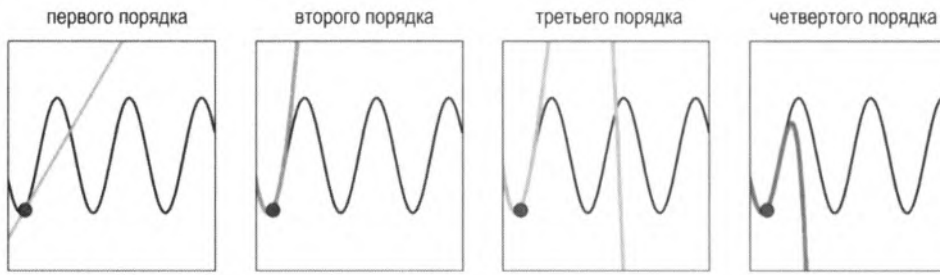


Рис. П2.11. Слева направо аппроксимация с использованием ряда Тейлора первого, второго, третьего и четвертого порядков функции $g(w) = \sin(w)$, нарисованная разными цветами и вычисленная в одной и той же входной точке

П2.9.3. Построение более качественных локальных аппроксимаций

Выведя эту квадратичную функцию на основе наших размышлений о касательной линии, можно, естественно, подумать о том, чтобы сделать еще один шаг вперед, т. е. найти простую функцию h, удовлетворяющую еще одному условию в дополнение к тем, которым удовлетворяет квадратичная функция.

$$\begin{aligned} h\left(w^0\right) &= g\left(w^0\right); \\ \frac{d}{dw} h\left(w^0\right) &= \frac{d}{dw} g\left(w^0\right); \\ \frac{d^{(2)}}{dw^{(2)}} h\left(w^0\right) &= \frac{d^{(2)}}{dw^{(2)}} g\left(w^0\right); \\ \frac{d^{(3)}}{dw^{(3)}} h\left(w^0\right) &= \frac{d^{(3)}}{dw^{(3)}} g\left(w^0\right). \end{aligned} \tag{П2.60}$$

Обратив внимание на то, что ни один многочлен второй степени не может удовлетворить этому последнему условию, т. к. его третья производная всегда равна нулю, мы могли бы найти многочлен третьей степени. Используя тот же анализ, что и выполненный ранее, задав соответствующую систему уравнений на основе обобщенного многочлена третьей степени, мы приходим к выводу, что следующее равенство действительно удовлетворяет всем критериям уравнения (П2.60):

$$\begin{aligned} h(w) = g\left(w^0\right) + \frac{d}{dw} g\left(w^0\right)\left(w - w^0\right) + \\ + \frac{1}{2}\frac{d^2}{dw^2} g\left(w^0\right)\left(w - w^0\right)^2 + \frac{1}{6}\frac{d^3}{dw^3} g\left(w^0\right)\left(w - w^0\right)^3. \end{aligned} \tag{П2.61}$$

Этот многочлен еще лучше аппроксимирует g вблизи точки w^0, чем квадратичная функция, как показано на конкретном примере на рис. П2.11. Рассматривая этот рисунок, мы ясно видим, что аппроксимация становится все лучше и лучше по мере увеличения степени аппроксимации. Это имеет смысл, поскольку каждый многочлен содержит больше информации о производной опорной функции по мере увеличения степени. Однако мы никак не ожидаем, что он будет совпадать со всей

функцией везде: мы строим каждый многочлен, чтобы он совпадал с g только в одной точке, поэтому независимо от степени мы можем ожидать, что он будет совпадать только с опорной функцией вблизи точки w^0.

Задание множества из $N+1$ критериев, первый из которых требует, чтобы $h(w^0)=g(w^0)$, а остальные N требуют, чтобы первые N производных функции h совпадали с производными функции g в w^0, приводит к построению многочлена степени N:

$$h(w)=g(w^0)+\sum_{n=1}^{N}\frac{1}{n!}\frac{d^{(n)}}{dw^{(n)}}g(w^0)(w-w^0)^n. \qquad (П2.62)$$

Этот общий многочлен степени N называется *рядом Тейлора* функции g в точке w^0.

П2.9.4. Многовходной ряд Тейлора

В случае с одновходной функцией мы увидели, как общая аппроксимация с использованием ряда Тейлора может рассматриваться в качестве естественного продолжения касательной линии для более высокостепенных многочленных аппроксимаций. История с многовходными функциями является в точности аналогичной.

Если бы мы спросили, для какого многочлена $h(\mathbf{w})$ первой степени наблюдается совпадение как с функцией $g(\mathbf{w})$ в точке $\mathbf{w}^0$, так с градиентами функции в этой же точке, т. е.

$$\begin{aligned} h(\mathbf{w}^0)&=g(\mathbf{w}^0);\\ \nabla h(\mathbf{w}^0)&=\nabla g(\mathbf{w}^0), \end{aligned} \qquad (П2.63)$$

то могли бы создать систему уравнений (зеркально отражающую ту, которую мы образовали, задав аналогичный вопрос для одновходных функций) и восстановить касательную гиперплоскости (нашу аппроксимацию с использованием ряда Тейлора первого порядка), которую мы видели еще в *разд. П2.4*

$$h(\mathbf{w})=g(\mathbf{w}^0)+\nabla g(\mathbf{w}^0)^T(\mathbf{w}-\mathbf{w}^0). \qquad (П2.64)$$

Обратите внимание, что эта запись представляет собой точный аналог аппроксимации первого порядка для одновходных функций и сводится к ней (касательной линии) при $N=1$.

Схожим образом, задавшись вопросом о том, какая (квадратичная) функция h второй степени может совпадать с g в точке $\mathbf{w}^0$ в терминах значения, которое она принимает, а также значений, которые принимают ее первая и вторая производные, т. е.

$$\begin{aligned} h(\mathbf{w}^0)&=g(\mathbf{w}^0);\\ \nabla h(\mathbf{w}^0)&=\nabla g(\mathbf{w}^0);\\ \nabla^2 h(\mathbf{w}^0)&=\nabla^2 g(\mathbf{w}^0), \end{aligned} \qquad (П2.65)$$

мы также вывели бы (как мы явно сделали с одновходным случаем) аппроксимацию рядом Тейлора второго порядка:

$$h(\mathbf{w}) = g(\mathbf{w}^0) + \nabla g(\mathbf{w}^0)^T (\mathbf{w} - \mathbf{w}^0) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^0)^T \nabla^2 g(\mathbf{w}^0)(\mathbf{w} - \mathbf{w}^0). \quad (П2.66)$$

Обратите внимание еще раз на то, что эта форма является точным аналогом аппроксимации второго порядка для одновходных функций и сводится к ней при $N = 1$.

На рис. П2.12 построены графики аппроксимации рядом Тейлора первого и второго порядков (показанные соответственно зеленым и бирюзовым цветом) функции $g(w_1, w_2) = \sin(w_1)$ в точке вблизи начала координат. Как и в случае с одновходными функциями, аппроксимация второго порядка является гораздо более качественным локальным аппроксиматором, чем апроксимация первого порядка, поскольку она содержит больше информации о производной.

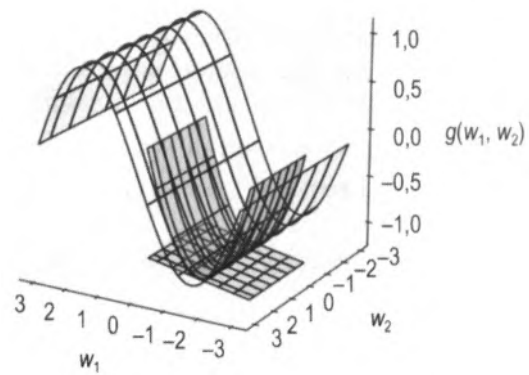


Рис. П2.12. Аппроксимации с использованием ряда Тейлора первого порядка (окрашено в светло-зеленый цвет) и второго порядка (окрашено в бирюзовый цвет) функции $g(w_1, w_2) = \sin(w_1)$ в точке вблизи начала координат

Аппроксимации более высокого порядка с использованием рядов Тейлора могут быть определены точно так же, как и в одновходном случае. Главное отличие многовходных функций состоит в том, что аппроксимации более высокого порядка, начиная с третьей производной, требуют серьезной манипуляции тензорами частных производных.

И хотя сложные аппроксимации легко определить, на практике все-таки используются только аппроксимации второго порядка. Это обусловлено тем, что (как мы видели в *разд. П2.8.2*) число частных производных растет экспоненциально согласно порядку производной. Отсюда, даже если мы получаем более качественный (локальный) аппроксиматор по мере увеличения порядка ряда Тейлора, серьезная ловушка вычисления/хранения экспоненциально многочисленных частных производных сводит это преимущество на нет.

П2.10. Использование библиотеки autograd

В этом разделе мы покажем, как использовать простой, но мощный *автоматический дифференциатор*, написанный на Python, под названием autograd [10, 11] — инструмент, который мы широко используем во всей книге.

П2.10.1. Установка библиотеки autograd

Библиотека autograd — это профессиональный градиентный калькулятор с открытым исходным кодом, или *автоматический дифференциатор*, по умолчанию использующий обратный режим, описанный в предыдущем разделе. Он позволяет автоматически вычислять производную произвольно сложных функций, построенных с использованием опорных функций языка Python и библиотеки NumPy.

Он также очень прост в установке: просто откройте Терминал и наберите

```
pip install autograd
```

Вы также можете посетить репозиторий GitHub библиотеки autograd по адресу **https://github.com/HIPS/autograd**, чтобы скачать тот же комплект файлов на свой компьютер.

Еще один инструмент под названием JAX, построенный тем же сообществом, что и расширение библиотеки autograd для использования на графических и тензорных процессорах, может использоваться очень похожим образом, как описано здесь. Скачать его можно, посетив веб-сайт **https://github.com/google/jax**.

Наряду с библиотекой autograd мы также настоятельно рекомендуем дистрибутив Anaconda Python 3, который можно установить, посетив **https://anaconda.com**.

Этот стандартный дистрибутив языка Python включает в себя ряд полезных библиотек, в том числе библиотеки NumPy, matplotlib и блокноты Jupyter.

П2.10.2. Использование autograd

В этом разделе мы представим ряд примеров, освещающих базовое использование автоматического дифференциатора autograd. С помощью простых модулей мы можем легко вычислять производные одновходных функций, а также частные производные и полные градиенты многовходных функций, имплементированных на языке Python и с использованием библиотеки NumPy.

Пример П2.9. Вычисление производных одновходных функций

Поскольку библиотека autograd специально разработана для автоматического вычисления производных в коде NumPy, он поставляется со своей собственной оберткой вокруг базовой библиотеки NumPy. Именно здесь определяются правила дифференцирования (применительно конкретно к функциональности NumPy). Вы можете использовать библиотеку NumPy в версии библиотеки autograd точно так же, как и стандартную версию, т. к. в пользовательском интерфейсе практически ниче-

го не изменилось. Для импорта этой обернутой в autograd версии библиотеки NumPy наберите

```
# инструкция импорта для библиотеки NumPy, обернутой в autograd
import autograd.numpy as np
```

Мы начнем демонстрацию применения библиотеки autograd с использования простой функции

$$g(w) = \tanh(w), \tag{П2.67}$$

чья производная, записанная алгебраически, такова:

$$\frac{d}{dw} g(w) = 1 - \tanh^2(w). \tag{П2.68}$$

В языке Python существуют два распространенных способа определения функций. Во-первых, это стандартное объявление функции Python (как показано ниже).

```
# именованная функция Python
def g(w):
    return np.tanh(w)
```

На языке Python можно также создавать *анонимные* функции (функции, которые можно определить в одной строке кода) с помощью команды `lambda`.

```
# функция, определенная посредством lambda
g = lambda w: np.tanh(w)
```

Независимо от того, как мы определяем нашу функцию на языке Python, математически/вычислительно она все равно сводится к одному и тому же.

Для того чтобы вычислить производную нашей функции, мы должны сначала импортировать градиентный калькулятор, удобно именуемый `grad`.

```
# импортировать базовый автоматический дифференциатор библиотеки autograd
from autograd import grad
```

Для того чтобы применить градиентный калькулятор `grad`, мы просто передаем ему функцию, которую хотим дифференцировать. Метод `grad` работает, *явно* вычисляя вычислительный граф нашего входа, возвращая его производную, которую мы затем можем вычислить везде, где захотим. `grad` предоставляет не *алгебраическую функцию*, а *Python-функцию*. Здесь мы назвали производную функцию нашего входа именем `dgdw`.

```
# создать градиент от g
dgdw = grad(g)
```

Мы можем вычислить производные более высокого порядка нашей входной функции, используя ту же самую функциональность библиотеки autograd рекурсивно, т. е. подключив производную функцию `dgdw` в метод `grad` библиотеки autograd. Сделав это один раз, мы получим вторую производную Python-функции, которую мы назвали `d2gdw2`.

```
# вычислить вторую производную от g
d2gdw2 = grad(dgdw)
```

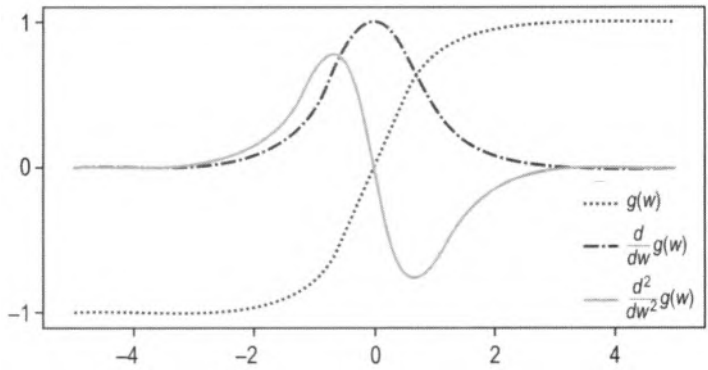


Рис. П2.13. Иллюстрация примера П2.9

Построим график входной функции, а также ее первой и второй производных на рис. П2.13.

Пример П2.10. Вычисление функции и градиента

Как уже упоминалось в предыдущем разделе, когда мы используем автоматический дифференциатор для вычисления градиента функции, в этом процессе мы вычисляем *саму функцию*. Другими словами, всякий раз, когда мы вычисляем градиент, мы получаем вычисление функции бесплатно.

Однако метод `grad` в предыдущем примере возвращал только одно значение: результат вычисления производной. Функция действительно вычисляется "под капотом" метода `grad` и просто не возвращается для удобства использования.

В библиотеке autograd существует еще один метод под названием `value_and_grad`, который возвращает все, что вычисляется "под капотом", включая производную и результат вычисления функции. Ниже мы используем эту функциональность библиотеки autograd для воспроизведения расчетов первой производной из предыдущего примера.

```
# импортировать автоматический дифференциатор библиотеки autograd
from autograd import value_and_grad

# создать градиент функции g
dgdw = value_and_grad(g)

# вычислить g и ее градиент в w=0
w = 0.0
g_val, grad_val = dgdw(w)
```

Пример П2.11. Вычисление аппроксимаций с использованием рядов Тейлора

Используя autograd, мы можем легко вычислить аппроксимации с применением ряда Тейлора (*см. разд. П2.9*) любой одновходной функции. Возьмем, например,

функцию $g(w)=\tanh(w)$ вместе с ее аппроксимацией рядом Тейлора первого порядка

$$h(w)=g\left(w^0\right)+\frac{d}{dw}g\left(w^0\right)\left(w-w^0\right), \qquad (П2.69)$$

центрированной в точке $w^0=1$. Сначала мы производим эту функцию и ее первопорядковую аппроксимацию с помощью ряда Тейлора на Python следующим образом:

```
# создать функцию g и ее первую производную
g = lambda w: np.tanh(w)
dgdw = grad(g)

# создать аппроксимацию с использованием ряда Тейлора первого порядка
first_order = lambda w0, w: g(w0) + dgdw(w0)*(w - w0)
```

Далее мы вычисляем функцию и строим ее график (черным цветом) и ее аппроксимации первого порядка (зеленым цветом) на рис. П2.14. Так же легко вычислить аппроксимацию с использованием ряда Тейлора второго порядка, формула которой задается как

$$q(w)=g\left(w^0\right)+\frac{d}{dw}g\left(w^0\right)\left(w-w^0\right)+\frac{1}{2}\frac{d^2}{dw^2}g\left(w^0\right)\left(w-w^0\right)^2. \qquad (П2.70)$$

```
# создать вторую производную g
d2gdw2 = grad(dgdw)

# создать аппроксимацию с использованием ряда Тейлора второго порядка
second_order = lambda w0, w: g(w0) + dgdw(w0)*(w - w0) + 0.5*d2gdw2(w0)*(w - w0)**2
```

Аппроксимация с использованием ряда Тейлора второго порядка показана (пунктирная кривая) на рис. П2.14, а точка расширения/касания — красным кругом.

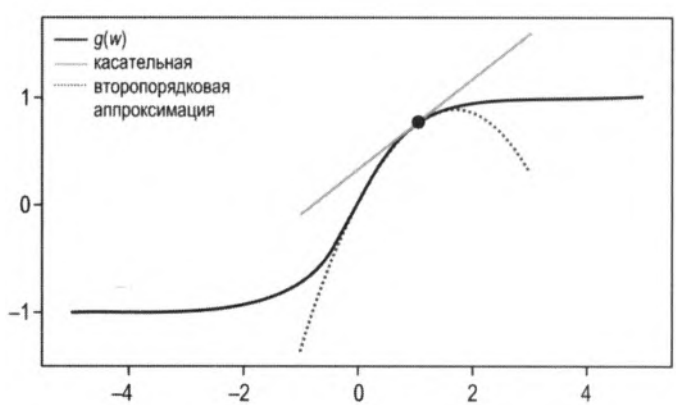


Рис. П2.14. Иллюстрация примера П2.11

Пример П2.12. Вычисление индивидуальных частных производных

Существует несколько способов использования библиотеки autograd для вычисления частных производных многовходной функции. Сначала давайте рассмотрим, как использовать autograd для вычисления частных производных по отдельности или по одному, начиная с функции

$$g(w_1, w_2) = \tanh(w_1 w_2). \qquad (П2.71)$$

Переведем эту функцию на язык Python:

```
# простая многовходная функция
def g(w_1, w_2):
    return np.tanh(w_1*w_2)
```

Принимая два входа, эта функция располагается в трех размерностях, как показано на рис. П2.15, *слева*.

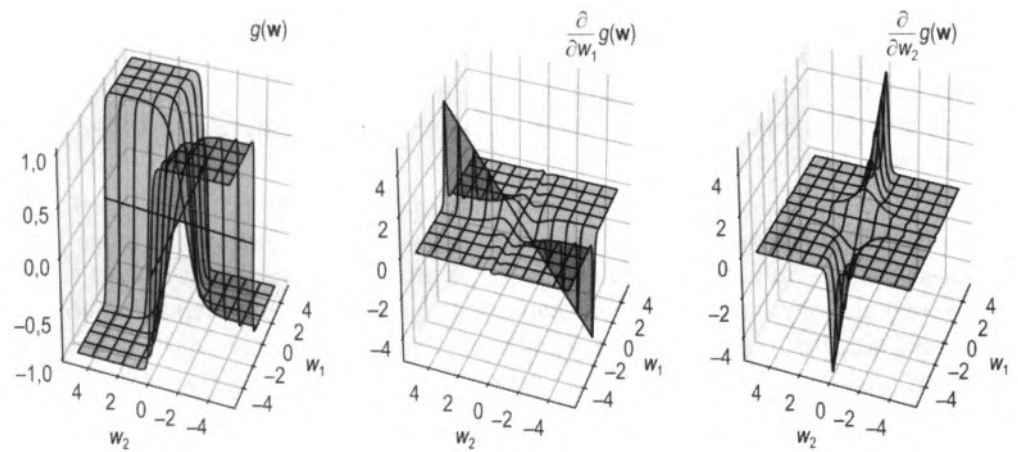


Рис. П2.15. Иллюстрация примера П2.12. Многовходная функция (*слева*) и ее частная производная в первом (*в центре*) и втором входах (*справа*)

Если мы применим точно такой же вызов, который мы использовали в предыдущих примерах, и напишем

```
grad(g)
```

поскольку наша функция принимает два входных значения, указанная инструкция вернет первую частную производную $\frac{\partial}{\partial w_1} g(w_1, w_2)$. В пакете autograd эта настройка принята *по умолчанию* для каждого метода автоматического дифференцирования.

В качестве альтернативы для вычисления этой частной производной как функции мы можем явно передать в `grad` второй аргумент либо любой другой метод библиотеки autograd, который представляет собой простой индекс, обозначающий, какая частная производная нам нужна. Для создания одинаковой (первой) частной произ-

водной таким образом мы передаем индекс `0` (т. к. индексирование в Python начинается с 0) вот так:

```
grad(g, 0)
```

Аналогично для того, чтобы построить вторую частную производную, мы передаем индекс `1`:

```
grad(g, 1)
```

В более общем случае, если g принимает N входов (с w_1 по w_N), то мы можем построить ее N-ю частную производную, используя тот же шаблон следующим образом:

```
grad(g, n-1)
```

Графики двух частных производных приведены на рис. П2.15, *в центре* и *справа*.

Пример П2.13. Вычисление нескольких производных или полного градиента

Развивая идею предыдущего примера, здесь мы рассмотрим, как использовать autograd для построения нескольких частных производных как функций сразу или всего градиента многовходной функции. Сделаем это, задействовав ту же самую функцию, что и в предыдущем примере.

Есть два способа сделать это с помощью autograd. Первый способ состоит в том, чтобы просто индексировать все частные производные, используя тот же самый введенный ранее вид нотации. Например, если мы хотим построить полный градиент двухвходной функции, задействованной в предыдущем примере, то сообщаем библиотеке autograd об этом желании, подавая на вход два индекса (0, 1), как показано ниже:

```
grad(g, (0,1))
```

В более общем случае для функции, принимающей N входов, для построения любого подмножества частных производных сразу мы используем тот же самый вид индексной нотации. Обратите внимание, что этот подход применим ко всем методам в библиотеке автоматического дифференцирования autograd.

Второй способ построить несколько производных одновременно с помощью autograd — это написать функцию в NumPy, где все желаемые переменные, которые мы хотим дифференцировать, вводятся в функцию в качестве одного аргумента. Например, вместо того чтобы написать нашу функцию на Python как

```
1| # простая многовходная функция, определенная на Python
2| def g(w_1, w_2):
3|     return np.tanh(w_1*w_2)
```

где и `w_1`, и `w_2` подаются по одной за раз, если мы запишем это эквивалентным образом, используя векторную нотацию как

```
1| def g(w):
2|     return np.tanh(w[0]*w[1])
```

затем вызовем

```
grad(g)
```

либо эквивалентным образом

```
grad(g, 0)
```

то мы породим производные g относительно ее первого аргумента, что здесь даст нам полный градиент g.

Этот индексный формат также соблюдается в более общем случае, т. е. инструкция

```
grad(g, n-1)
```

вычисляет производные функции по отношению к n-му входу (будь то одна переменная или несколько переменных).

П2.10.3. Выравнивание математических функций с помощью autograd

Математические функции бывают любых форм и размеров, и более того, мы часто можем выражать отдельные функции алгебраически разными способами. В этом коротком разделе обсуждается *выравнивание функций*, которое представляет собой удобный способ *стандартизации* функций, имплементированных на Python, чтобы, например, можно было быстрее применять локальную оптимизацию (в коде) без необходимости перебирать веса особенно сложной функции.

Пример П2.14. Выравнивание многовходной функции

Рассмотрим функцию

$$g(a, \mathbf{b}, \mathbf{C}) = \left(a + \mathbf{r}^T\mathbf{b} + \mathbf{z}^T\mathbf{C}\mathbf{z}\right)^2, \qquad (П2.72)$$

где входная переменная a — это скаляр; $\mathbf{b}$ — 2×1-вектор; $\mathbf{C}$ — 2×2-матрица, непеременные векторы $\mathbf{r}$ и $\mathbf{z}$ фиксированы соответственно в $\mathbf{r} = \begin{bmatrix}1 & 2\end{bmatrix}^T$ и $\mathbf{z} = \begin{bmatrix}1 & 3\end{bmatrix}^T$. Эта функция не написана в стандартной форме $g(\mathbf{w})$, в которой мы обсуждаем локальную оптимизацию в *главах 2–4*. Хотя, конечно, все принципы и алгоритмы, приведенные там, по-прежнему применимы к этой функции, реализовать любой из этих методов, естественно, будет сложнее, поскольку каждый шаг должен быть явно записан во всех трех входах a, $\mathbf{b}$ и $\mathbf{C}$, что написать (и имплементировать) будет обременительнее. Это раздражение значительно усиливается при работе с функциями многочисленных более явных входных переменных, с которыми мы регулярно сталкиваемся в машинном обучении. Для таких функций, чтобы сделать один шаг спуска с помощью некоторого локального метода, мы должны *перебирать* их многочисленные разные входные переменные.

К счастью, каждую математическую функцию можно выразить так, что *все* ее входные переменные будут представлены в виде одного сплошного вектора $\mathbf{w}$, что снижает это раздражение.

Например, определив один-единственный вектор

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \end{bmatrix} = \begin{bmatrix} a \\ b_1 \\ b_2 \\ c_{11} \\ c_{12} \\ c_{21} \\ c_{22} \end{bmatrix}, \tag{П2.73}$$

исходную функцию в уравнении (П2.72) можно эквивалентным образом записать как

$$g(\mathbf{w}) = \left(\mathbf{s}^T \mathbf{w}\right)^2, \tag{П2. 74}$$

где

$$\mathbf{s} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 1 \\ 3 \\ 3 \\ 9 \end{bmatrix}. \tag{П2.75}$$

Опять же отметим, что в действительности здесь мы лишь *переиндексировали* записи входных векторов подряд, благодаря чему мы можем имплементировать схемы локальной оптимизации менее громоздким способом одной алгебраической строкой или в одной строке кода autograd, вместо того чтобы требовать цикла над каждой входной переменной. Эта схема переиндексации переменных называется *выравниванием* (flattening) *функции* и наглядно показана на рис. П2.16.



Рис. П2.16. Иллюстрация примера П2.14. Фигуральная иллюстрация выравнивания функции, приведенной в уравнении (П2.72)

Хотя переиндексирование, необходимое для выравнивания функции, важно, оно (как и само вычисление производных) является повторяющейся и трудоемкой операцией для людей, чтобы выполнять ее вручную. К счастью, библиотека autograd

имеет встроенный модуль выравнивания функций, который можно импортировать, как показано ниже:

```
# импортировать модуль выравнивания функций из autograd
from autograd.misc.flatten import flatten_func
```

Определим Python-версию функции из уравнения (П2.72):

```
# Python-имплементация функции g
r = np.array([[1],[2]])
z = np.array([[1],[3]])
def g(input_weights):
    a = input_weights[0]
    b = input_weights[1]
    C = input_weights[2]
    return (((a + np.dot(r.T, b) + np.dot(np.dot(z.T, C), z)))**2)
        [0][0]
```

Для того чтобы выровнять g, мы просто вызываем

```
# сгладить входную функцию g
g_flat, unflatten_func, w = flatten_func(g, input_weights)
```

Здесь в `input_weights` находится список инициализаций для входных переменных, подаваемых в функцию g. Выходы `g_flat`, `unflatten_func` и `w` представляют собой соответственно выровненную версию функции g, модуль для снятия выравнивания входных весов и выровненную версию начальных весов.

3 ПРИЛОЖЕНИЕ
Линейная алгебра

П3.1. Введение

В этом приложении мы кратко рассмотрим базовые идеи линейной алгебры, которые являются основополагающими для понимания машинного обучения. К ним относятся векторная и матричная алгебра, векторные и матричные нормы и разложение на собственные числа. Читателю настоятельно рекомендуется ознакомиться со всеми концепциями, упомянутыми в этом приложении, прежде чем переходить к остальной части книги.

П3.2. Векторы и векторные операции

Мы начнем с рассмотрения фундаментального понятия вектора, а также векторной алгебры.

П3.2.1. Вектор

Вектор — это еще одно пончтие для упорядоченного перечисления чисел. Например,

$$\begin{bmatrix} -3 & 4 & 1 \end{bmatrix} \tag{П3.1}$$

есть вектор из трех элементов или записей, также именуемый *вектором размера* или *размерности* 3. В общем случае вектор может иметь произвольное число элементов и содержать числа, переменные или и то и другое. Например,

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix} \tag{П3.2}$$

есть вектор из четырех переменных. Когда числа или переменные перечислены по горизонтали (или в строке), мы называем результирующий вектор *вектор-строкой*. Однако мы также можем перечислить их вертикально (или в столбце), и в этом случае мы будем ссылаться на результирующий вектор как на *вектор-столбец*. Например,

$$\begin{bmatrix} -3 \\ 4 \\ 1 \end{bmatrix} \tag{П3.3}$$

теперь есть вектор-столбец размера 3. Мы можем переключаться между строковым и столбцовым представлениями вектора, транспонируя каждое из них. *Транспони-*

рование обычно обозначается надстрочной буквой T, расположенной справа и выше буквы вектора или скобок, и просто превращает вектор-строку в эквивалентный вектор-столбец, и наоборот. Например,

$$\begin{bmatrix} -3 \\ 4 \\ 1 \end{bmatrix}^T = \begin{bmatrix} -3 & 4 & 1 \end{bmatrix} \text{ и } \begin{bmatrix} -3 & 4 & 1 \end{bmatrix}^T = \begin{bmatrix} -3 \\ 4 \\ 1 \end{bmatrix}. \quad \text{(П3.4)}$$

В алгебраической нотации вектор записывают строчной полужирной латинской буквой, например $\mathbf{x}$. Тогда транспонирование $\mathbf{x}$ обозначается как $\mathbf{x}^T$. В этой нотации не конкретизируется, является вектор строкой или столбцом, а также сколько элементов он содержит. Поэтому такая информация должна быть представлена в явном виде. В этой книге, если не указано иное, мы исходим из того, что все векторы по умолчанию являются *вектор-столбцами.*

Векторы длины 2 (или 3) легко воспринимаются интуитивно, т. к. они располагаются в двумерном (или трехмерном) пространствах, интуитивно понятных нам. Например, двумерный вектор

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \text{(П3.5)}$$

может быть нарисован в двумерной плоскости в виде стрелки, начинающейся от начала координат и заканчивающейся в точке, горизонтальная и вертикальная координаты которой равны соответственно 1 и 2 (рис. П3.1, *слева*). Однако $\mathbf{x}$ можно альтернативно нарисовать (и представить) как одну-единственную точку, т. е. конечную точку стрелки (рис. П3.1, *справа*). При построении графика низкоразмерного набора данных машинного обучения (т. е. просто множества векторов) мы часто задействуем последний визуальный стиль.

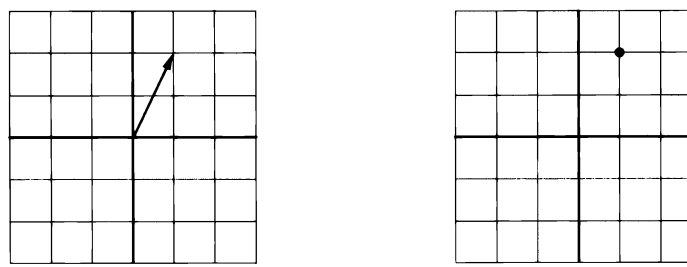


Рис. П3.1. Двумерный вектор, визуализированный в виде стрелки, идущей от начала координат (*слева*), или, что эквивалентно, в виде одной точки на двумерной плоскости (*справа*)

П3.2.2. Сложение векторов

Мы складываем (и вычитаем) два вектора поэлементно, учитывая, что два вектора должны иметь одинаковое число элементов (или одинаковый размер) и оба должны быть вектор-строками или вектор-столбцами.

Например, векторы

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \text{ и } \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{П3.6}$$

можно сложить поэлементно, получив в результате

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_N + y_N \end{bmatrix}. \tag{П3.7}$$

Вычитание **y** из **x** выполняется схожим образом:

$$\mathbf{x} - \mathbf{y} = \begin{bmatrix} x_1 - y_1 \\ x_2 - y_2 \\ \vdots \\ x_N - y_N \end{bmatrix}. \tag{П3.8}$$

Если вернуться к представлению векторов как стрелок, выходящих из начала координат, то сумму двух векторов можно определить как вектор, представляющий дальний угол параллелограмма, образованного двумя векторами. Это называется *правилом параллелограмма*. Пример приведен на рис. П3.2 для двух входных векторов в виде стрелок черного цвета, а их сумма показана вектором в виде стрелки серого цвета. Штриховые линии здесь являются просто визуальными направляющими, помогающими очертить параллелограмм, лежащий в основе суммы.

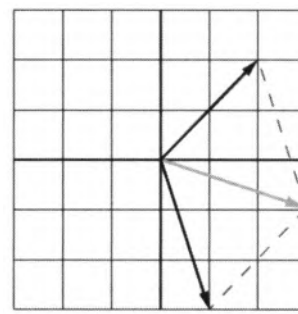


Рис. П3.2. Иллюстрация правила параллелограмма

П3.2.3. Умножение векторов

В отличие от сложения, существует несколько способов определения векторного умножения. Мы рассмотрим умножение вектора на скаляр, поэлементное умножение двух векторов, а также внутреннее и внешнее произведение двух векторов.

Умножение вектора на скаляр

Мы можем умножить любой вектор **x** на скаляр c, выполняя умножение поэлементно так:

$$c\mathbf{x} = \begin{bmatrix} cx_1 \\ cx_2 \\ \vdots \\ cx_N \end{bmatrix}. \tag{П3.9}$$

Поэлементное произведение двух векторов

Поэлементное произведение, иногда именуемое *произведением Адамара*, работает именно так, как оно звучит: мы умножаем два вектора элемент за элементом. Обратите внимание, что оба вектора должны иметь одинаковый размер. Нотационно поэлементное произведение двух векторов **x** и **y** записывается так:

$$\mathbf{x} \circ \mathbf{y} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ \vdots \\ x_N y_N \end{bmatrix}. \tag{П3.10}$$

Внутреннее произведение двух векторов

Внутреннее произведение (также именуемое точечным произведением) — это еще один способ умножить два вектора одинаковой размерности. В отличие от поэлементного произведения, внутреннее произведение двух векторов дает *скалярный результат*. Для того чтобы выполнить внутреннее произведение двух векторов, мы сначала умножаем их между собой поэлементно, а затем просто складываем элементы. Внутреннее произведение векторов **x** и **y** записывается как

$$\mathbf{x}^T \mathbf{y} = x_1 y_1 + x_2 y_2 + \ldots + x_N y_N = \sum_{i=1}^{N} x_i y_i. \tag{П3.11}$$

Длина или норма вектора

Хорошо известная теорема Пифагора предоставляет полезный способ измерения длины вектора в двух размерностях. Используя теорему Пифагора, мы можем трактовать длину двумерного вектора

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{П3.12}$$

как гипотенузу прямоугольного треугольника и записать

$$\text{длина вектора } \mathbf{x} = \sqrt{x_1^2 + x_2^2}. \tag{П3.13}$$

Обратите внимание, что мы также можем выразить длину вектора $\mathbf{x}$ в терминах внутреннего произведения вектора $\mathbf{x}$ с самим собой, как

$$\text{длина вектора } \mathbf{x} = \sqrt{\mathbf{x}^T\mathbf{x}}, \tag{П3.14}$$

и это распространяется на векторы любого размера. Используя нотацию $\|\mathbf{x}\|_2$ для обозначения длины N-мерного вектора $\mathbf{x}$, мы имеем

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^T\mathbf{x}} = \sqrt{\sum_{i=1}^{N} x_i^2}. \tag{П3.15}$$

Геометрическая интерпретация внутреннего произведения

Внутреннее произведение двух векторов $\mathbf{x}$ и $\mathbf{y}$

$$\mathbf{x}^T\mathbf{y} = \sum_{i=1}^{N} x_i y_i \tag{П3.16}$$

можно выразить в терминах длин векторов $\mathbf{x}$ и $\mathbf{y}$, используя так называемое *правило внутреннего произведения*, как

$$\mathbf{x}^T\mathbf{y} = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos\theta, \tag{П3.17}$$

где θ — это угол между $\mathbf{x}$ и $\mathbf{y}$.

Указанное правило, возможно, лучше всего интуитивно понять после небольшой перестановки его членов:

$$\left(\frac{\mathbf{x}}{\|\mathbf{x}\|_2}\right)^T \left(\frac{\mathbf{y}}{\|\mathbf{y}\|_2}\right) = \cos\theta, \tag{П3.18}$$

где векторы $\mathbf{x}/\|\mathbf{x}\|_2$ и $\mathbf{y}/\|\mathbf{y}\|_2$ по-прежнему указывают в том же направлении, что $\mathbf{x}$ и $\mathbf{y}$ соответственно, но оба были нормализованы, чтобы иметь единичную длину, поскольку

$$\left\|\frac{\mathbf{x}}{\|\mathbf{x}\|_2}\right\|_2 = \left\|\frac{\mathbf{y}}{\|\mathbf{y}\|_2}\right\|_2 = 1. \tag{П3.19}$$

Обратите внимание, что поскольку значение косинуса всегда находится между –1 и 1, то же самое происходит и с внутренним произведением любых двух векторов единичной длины. Когда они указывают в одном и том же направлении, $\theta = 0$, и их внутреннее произведение является максимальным (т. е. 1). Когда два вектора указывают друг на друга, θ увеличивается и внутреннее произведение начинает уменьшаться. Когда два вектора *перпендикулярны* друг другу, их внутреннее произведение равно нулю. Внутреннее произведение достигает своего минимального значения (т. е. –1), когда $\theta = \pi$, т. е. векторы указывают в противоположные направления (рис. П3.3).

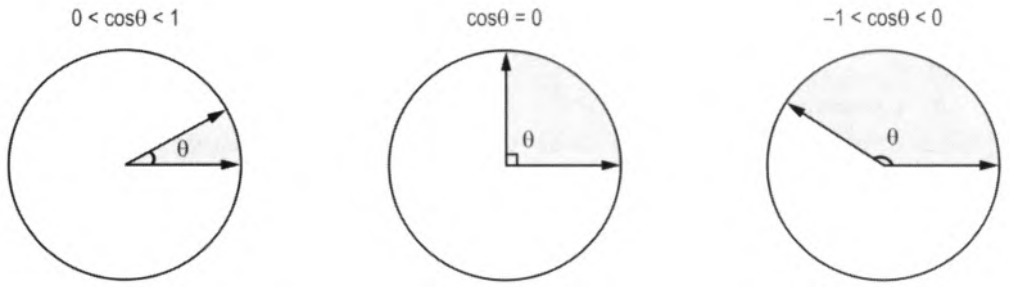


Рис. П3.3. Внутреннее произведение двух векторов единичной длины равно косинусу угла θ, создаваемого между ними

Внешнее произведение двух векторов

Внешнее произведение — это еще один способ определить умножение двух векторов. Для двух вектор-столбцов (не обязательно одинакового размера)

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \text{ и } \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} \tag{П3.20}$$

их внешнее произведение записывается как $\mathbf{x}\mathbf{y}^T$ и определяется как

$$\mathbf{x}\mathbf{y}^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \cdots & y_M \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \cdots & x_1 y_M \\ x_2 y_1 & x_2 y_2 & \cdots & x_2 y_M \\ \vdots & \vdots & \ddots & \vdots \\ x_N y_1 & x_N y_2 & \cdots & x_N y_M \end{bmatrix}. \tag{П3.21}$$

В результате получается $(N \times M)$-матрица, которую можно рассматривать как коллекцию из M вектор-столбцов длины N, уложенных бок о бок (или как коллекцию из N вектор-строк длины M, уложенных друг на друга). Мы вернемся к матрицам и обсудим их в следующем разделе.

П3.2.4. Линейная комбинация векторов

Линейная комбинация — это операция, которая обобщает простое сложение двух векторов путем совмещения сложения и умножения на скаляр. Пусть имеются два вектора $\mathbf{x}_1$ и $\mathbf{x}_2$ одинакового размера. Тогда их линейная комбинация получается путем умножения каждого из них сначала на скаляр, а затем сложения результата:

$$\alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2, \tag{П3.22}$$

где α_1 и α_2 — это действительные числа. Обратите внимание, что для данной пары значений (α_1, α_2) линейная комбинация является вектором с той же размерностью, что и $\mathbf{x}_1$ и $\mathbf{x}_2$. На рис. П3.4 показана линейная комбинация векторов:

$$\mathbf{x}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \text{ и } \mathbf{x}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \tag{П3.23}$$

для трех вариантов пар (α_1, α_2). Множество всех таких векторов, созданных путем взятия линейной комбинации векторов $\mathbf{x}_1$ и $\mathbf{x}_2$, называется *охватом* (или *линейной оболочкой*) $\mathbf{x}_1$ и $\mathbf{x}_2$ и записывается как

$$\text{охват } \mathbf{x}_1 \text{ и } \mathbf{x}_2 = \left\{ \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 \middle| (\alpha_1, \alpha_2) \in \mathbb{R}^2 \right\}. \tag{П3.24}$$

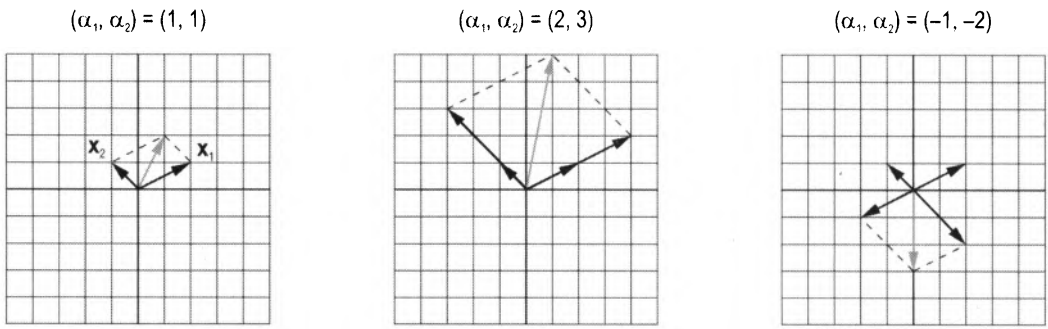


Рис. П3.4. Линейная комбинация (обозначена стрелкой серого цвета) векторов $\mathbf{x}_1$ и $\mathbf{x}_2$, определенных в уравнении (П3.24) для трех вариантов пар (α_1, α_2). Как вы видите, изменяя значения α_1 и α_2 в $\alpha_1\mathbf{x}_1 + \alpha_2\mathbf{x}_2$, мы каждый раз получаем новый вектор. Множество всех таких векторов называется охватом $\mathbf{x}_1$ и $\mathbf{x}_2$, который в данном случае является двумерной плоскостью

Для векторов $\mathbf{x}_1$ и $\mathbf{x}_2$ в уравнении (П3.23) охват представляет собой всеобъемлющую двумерную плоскость. Но это не всегда верно для любой пары векторов $\mathbf{x}_1$ и $\mathbf{x}_2$. Возьмем, например,

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ и } \mathbf{x}_2 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}. \tag{П3.25}$$

Поскольку эти два вектора указывают в одном и том же направлении (один скалярно кратен другому), любая линейная комбинация этих двух векторов будет иметь одинаковое направление. В этом случае линейный охват $\mathbf{x}_1$ и $\mathbf{x}_2$ — это уже не всеобъемлющая двумерная плоскость, а одномерная линия, которую можно проследить с помощью скалярных кратных любого из двух векторов. Другими словами, при наличии одного из векторов, $\mathbf{x}_1$ и $\mathbf{x}_2$, другой становится избыточным (с точки зрения отыскания их охвата). В терминах линейной алгебры такие векторы называются *линейно зависимыми*.

Понятие линейной комбинации векторов может быть распространено в общем случае на множество k векторов $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k\}$ (все одного размера):

$$\sum_{i=1}^{k} \alpha_i \mathbf{x}_i = \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + ... + \alpha_k \mathbf{x}_k. \tag{П3.26}$$

Если эти векторы охватывают k-мерное пространство, они называются *линейно независимыми*. В противном случае существует по крайней мере один вектор в множестве, который может быть записан как линейная комбинация остальных.

П3.3. Матрицы и матричные операции

В этом разделе мы рассмотрим понятие матрицы, а также базовые операции, которые можно выполнять с одной матрицей или парами матриц. Эти операции аналогичны операциям над векторами, описанными в предыдущем разделе, включая операцию транспонирования, сложение/вычитание и несколько операций умножения. Из-за близкого сходства матриц с векторами этот раздел гораздо более лаконичен, чем предыдущий.

П3.3.1. Матрица

Если мы возьмем множество из N вектор-строк, каждый из которых имеет размер M:

$$\begin{aligned}
\mathbf{x}_1 &= [x_{11} \quad x_{12} \quad \ldots \quad x_{1M}]; \\
\mathbf{x}_2 &= [x_{21} \quad x_{22} \quad \ldots \quad x_{2M}]; \\
&\vdots \\
\mathbf{x}_N &= [x_{N1} \quad x_{N2} \quad \ldots \quad x_{NM}],
\end{aligned}$$

и уложим их друг на друга, то мы сформируем объект, именуемый *матрицей*

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1M} \\ x_{21} & x_{22} & \ldots & x_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \ldots & x_{NM} \end{bmatrix} \qquad \text{(П3.27)}$$

размера $N \times M$, где N — это число строк в матрице; M — число столбцов. Для описания матрицы в книге мы используем нотацию в виде жирной прописной буквы, например $\mathbf{X}$. Как и в векторной нотации, здесь ничего о размере матрицы не сообщается, и он должен быть указан явным образом.

Операцию транспонирования, которую мы первоначально ввели для векторов, можно расширить для матриц. При выполнении операции транспонирования вся матрица переворачивается: каждый столбец превращается в строку, а затем эти строки укладываются одна на другую, образуя $(M \times N)$-матрицу

$$\mathbf{X}^T = \begin{bmatrix} x_{11} & x_{21} & \ldots & x_{N1} \\ x_{12} & x_{22} & \ldots & x_{N2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{1M} & x_{2M} & \ldots & x_{NM} \end{bmatrix}. \qquad \text{(П3.28)}$$

П3.3.2. Сложение матриц

Как и в случае с векторами, сложение (и вычитание) выполняется поэлементно на матрицах одинаковых размеров. Например, для двух $(N \times M)$-матриц

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1M} \\ x_{21} & x_{22} & \dots & x_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{NM} \end{bmatrix} \text{ и } \mathbf{Y} = \begin{bmatrix} y_{11} & y_{12} & \dots & y_{1M} \\ y_{21} & y_{22} & \dots & y_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ y_{N1} & y_{N2} & \dots & y_{NM} \end{bmatrix} \quad \text{(П3.29)}$$

их сумма определяется как

$$\mathbf{X} + \mathbf{Y} = \begin{bmatrix} x_{11} + y_{11} & x_{12} + y_{12} & \dots & x_{1M} + y_{1M} \\ x_{21} + y_{21} & x_{22} + y_{22} & \dots & x_{2M} + y_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} + y_{N1} & x_{N2} + y_{N2} & \dots & x_{NM} + y_{NM} \end{bmatrix}. \quad \text{(П3.30)}$$

П3.3.3. Умножение матриц

Как и в случае с векторами, существуют разные способы умножения матриц, которые мы рассмотрим здесь.

Умножение матрицы на скаляр

Мы можем умножить любую матрицу $\mathbf{X}$ на скаляр c, и эта операция выполняется элемент за элементом так:

$$c\mathbf{X} = \begin{bmatrix} cx_{11} & cx_{12} & \dots & cx_{1M} \\ cx_{21} & cx_{22} & \dots & x_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ cx_{N1} & cx_{N2} & \dots & x_{NM} \end{bmatrix}. \quad \text{(П3.31)}$$

Умножение матрицы на вектор

Вообще говоря, есть два способа умножения $(N \times M)$-матрицы $\mathbf{X}$ на вектор $\mathbf{y}$. Первый, именуемый *левым умножением*, предусматривает умножение на N-мерный вектор-строку. Эта операция, записанная как $\mathbf{yX}$, в результате дает вектор-строку размера M, m-й элемент которого является внутренним произведением $\mathbf{y}$ с m-м столбцом матрицы $\mathbf{X}$:

$$\mathbf{yX} = \begin{bmatrix} \sum_{i=1}^{N} y_i x_{i1} & \sum_{i=1}^{N} y_i x_{i2} & \dots & \sum_{i=1}^{N} y_i x_{iM} \end{bmatrix}. \quad \text{(П3.32)}$$

Правое умножение — это умножение матрицы $\mathbf{X}$ справа на M-мерный вектор-столбец. Записанное как $\mathbf{Xy}$, правое умножение приводит к N-мерному вектор-

столбцу, n-й элемент которого является внутренним произведением $\mathbf{y}$ с n-й строкой матрицы $\mathbf{X}$:

$$\mathbf{Xy} = \begin{bmatrix} \sum_{j=1}^{M} y_j x_{1j} \\ \sum_{j=1}^{M} y_j x_{2j} \\ \vdots \\ \sum_{j=1}^{M} y_j x_{Nj} \end{bmatrix}. \qquad \text{(П3.33)}$$

Поэлементное умножение двух матриц

Как и в случае с векторами, мы можем определить поэлементное перемножение двух матриц одинакового размера. Поэлементное произведение двух $(N \times M)$-матриц $\mathbf{X}$ и $\mathbf{Y}$ записывается так:

$$\mathbf{X} \circ \mathbf{Y} = \begin{bmatrix} x_{11}y_{11} & x_{12}y_{12} & \cdots & x_{1M}y_{1M} \\ x_{21}y_{21} & x_{22}y_{22} & \cdots & x_{2M}y_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1}y_{N1} & x_{N2}y_{N2} & \cdots & x_{NM}y_{NM} \end{bmatrix}. \qquad \text{(П3.34)}$$

Общее умножение двух матриц

Общее произведение (или просто произведение) двух матриц $\mathbf{X}$ и $\mathbf{Y}$ может быть определено на основе операции векторного внешнего произведения при условии, что число столбцов в матрице $\mathbf{X}$ совпадает с числом строк в матрице $\mathbf{Y}$. То есть $\mathbf{X}$ и $\mathbf{Y}$ должны иметь размеры соответственно $N \times M$ и $M \times P$, чтобы можно было вычислить произведение матриц:

$$\mathbf{XY} = \sum_{m=1}^{M} \mathbf{x}_m \mathbf{y}_m^T, \qquad \text{(П3.35)}$$

где $\mathbf{x}_m$ — m-й столбец матрицы $\mathbf{X}$; $\mathbf{y}_m^T$ — транспонированный m-й столбец матрицы $\mathbf{Y}^T$ (или, что эквивалентно, m-я строка матрицы $\mathbf{Y}$). Обратите внимание, что каждое слагаемое в уравнении (П3.35) само по себе является матрицей размера $N \times P$, и такой же является финальная матрица $\mathbf{XY}$.

Общее умножение матрицы также можно определить поэлементно, используя внутренние произведения векторов, где запись в n-й строке и p-м столбце матрицы $\mathbf{XY}$ отыскивается как внутреннее произведение n-й строки матрицы $\mathbf{X}$ и p-го столбца матрицы $\mathbf{Y}$.

П3.4. Собственные значения и собственные векторы

В этом разделе мы проведем обзор общих линейных функций и выясним их связь с матрицами. Мы особо сосредоточимся на частном случае квадратной матрицы, в отношении которой обсудим важные темы собственных векторов и собственных чисел.

П3.4.1. Линейные функции и матричное умножение

Как мы обсуждали в предыдущем разделе, произведение матрицы размера $N \times M$

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1M} \\ x_{21} & x_{22} & \dots & x_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{NM} \end{bmatrix} \tag{П3.36}$$

на M-мерный вектор-столбец

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix} \tag{П3.37}$$

есть N-мерный вектор-столбец, записываемый как $\mathbf{Xw}$. Если трактовать вектор $\mathbf{w}$ как вход, то $\mathbf{Xw}$ определяет функцию g, записываемую формально как

$$g(\mathbf{w}) = \mathbf{Xw}. \tag{П3.38}$$

Запишем $g(\mathbf{w})$ явным образом:

$$g(\mathbf{w}) = \begin{bmatrix} x_{11}w_1 + x_{12}w_2 + \dots + x_{1M}w_M \\ x_{21}w_1 + x_{22}w_2 + \dots + x_{2M}w_M \\ \vdots \\ x_{N1}w_1 + x_{N2}w_2 + \dots + x_{NM}w_M \end{bmatrix}. \tag{П3.39}$$

Тогда становится ясно, что каждый ее элемент является линейной функцией в переменных с w_1 по w_M, и поэтому g называется *линейной функцией*.

П3.4.2. Линейные функции и квадратные матрицы

Когда число строк в матрице $\mathbf{X}$ совпадает с числом столбцов в ней, т. е. $N = M$, такая матрица называется *квадратной матрицей*. Когда $N = M = 2$, мы можем визуально исследовать эффект линейной функции $g(\mathbf{w}) = \mathbf{Xw}$, наблюдая, как двумерные точки $\mathbf{w}$ преобразуются посредством функции g.

На рис. П3.5 мы приводим именно такую визуализацию, используя матрицу $\mathbf{X}$ размера 2×2, записи которой были заданы случайно:

$$\mathbf{X} = \begin{bmatrix} 0,726 & -1,059 \\ -0,200 & -0,947 \end{bmatrix}. \qquad \text{(П3.40)}$$

На рис. П3.5, *слева* показана грубая координатная сетка. Цель этой сетки состоит в том, чтобы помочь визуализировать, как каждая точка, составляющая линии сетки (и, следовательно, все пространство), преобразуется с помощью матрицы $\mathbf{X}$ из формулы (П3.40). Для визуализации в верхней части сетки нарисована окружность радиуса 2, которая преобразуется вместе с ней. На рис. П3.5, *справа* показано, как пространство, показанное на рисунке слева, искажается функцией $g(\mathbf{w}) = \mathbf{X}\mathbf{w}$.

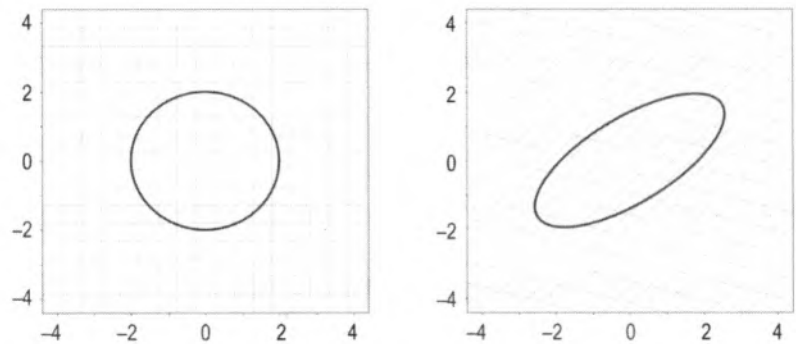


Рис. П3.5. Входное (*слева*) и выходное (*справа*) пространства линейной функции $g(\mathbf{w}) = \mathbf{X}\mathbf{w}$ с матрицей $\mathbf{X}$, определенной в формуле (П3.40)

П3.4.3. Собственные числа и собственные векторы

Предыдущая визуализация на рис. П3.5 интересна сама по себе, но при внимательном рассмотрении может быть использована и для того, чтобы сформулировать понятие *собственных векторов*. Это несколько направлений, которые, в отличие от большинства других, искаженных и скрученных умножением на данную матрицу, шкалируются только функцией. Другими словами, собственные векторы — это те особые векторы во входном пространстве, которые сохраняют свое направление после прохождения линейного преобразования g. На рис. П3.6 снова показано преобразование, обеспечиваемое случайной матрицей $\mathbf{X}$ из формулы (П3.40). На этот раз, однако, выделены два собственных вектора (черные стрелки). Сравнивая рисунки слева и справа, обратите внимание, что ни одно из направлений не искажается и не скручивается трансформацией: они только шкалируются.

То, что мы видели с линейными функциями, основанными на квадратных матрицах размера 2×2, имеет более общее значение и для более высоких размеров: линейная функция, основанная на матрице размера $N \times N$, влияет не более чем на N линейно независимых направлений, просто шкалируя их. Для матрицы $\mathbf{X}$ каждое такое направление $\mathbf{v} \neq \mathbf{0}_{N\times 1}$, удовлетворяющее

$$\mathbf{X}\mathbf{v} = \lambda\mathbf{v}, \qquad \text{(П3.41)}$$

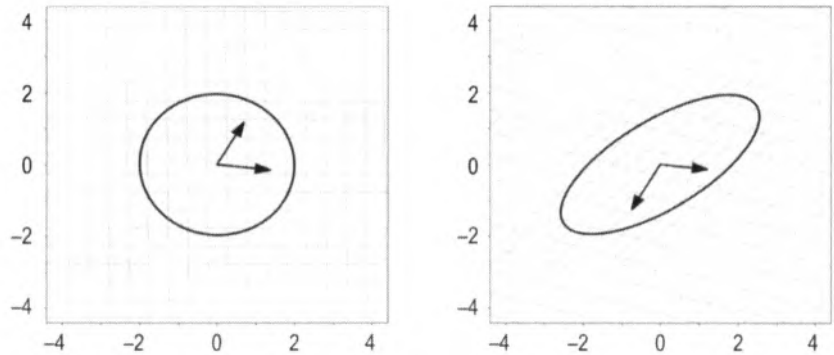


Рис. П3.6. Перерисовка рис. П3.5, где два собственных вектора матрицы **X** добавлены в виде двух черных стрелок как во входном, так и в выходном пространстве

называется *собственным вектором*. Здесь значение λ является именно той величиной, на которую **X** шкалирует **v**, и называется *собственным значением*. В общем случае λ может принимать действительные или компле́ксные значения.

П3.4.4. Частный случай симметричной матрицы

Симметричная матрица, т. е. квадратная матрица **X**, такая что $\mathbf{X} = \mathbf{X}^T$, является важным частным случаем квадратной матрицы, возникающей в широком диапазоне контекстов (например, матрицы Гессе, матрицы ковариаций и т. д.). Одно из главных преимуществ таких матриц перед просто квадратными заключается в следующем: их собственные векторы всегда перпендикулярны друг другу, а их собственные значения всегда являются действительными числами [74–76]. Этот факт имеет значительные последствия при анализе таких матриц, поскольку мы можем *диагонализировать* их следующим образом.

Уложив все собственные векторы матрицы **X** по столбцам в матрицу **V** и поместив соответствующие собственные значения вдоль диагонали матрицы **D**, мы можем записать выражение (П3.41) одновременно для всех собственных векторов/значений, как

$$\mathbf{XV} = \mathbf{VD}. \qquad (П3.42)$$

Когда все собственные векторы перпендикулярны друг другу, **V** есть *ортонормальная матрица*[1], и мы имеем $\mathbf{VV}^T = \mathbf{I}$. Отсюда, умножив обе части уравнения (П3.42) на $\mathbf{V}^T$ (справа), мы можем полностью выразить матрицу **X** в терминах ее собственных векторов/значений как

$$\mathbf{X} = \mathbf{VDV}^T. \qquad (П3.43)$$

[1] Здесь мы приняли допущение, что каждый собственный вектор **v**, удовлетворяющий уравнению (П3.41), имеет единичную длину, т. е. $\|\mathbf{v}\|_2 = 1$. Если нет, то мы всегда можем заменить **v** на $\mathbf{v} / \|\mathbf{v}\|_2$, и уравнение (П3.41) будет по-прежнему соблюдаться.

П3.5. Векторные и матричные нормы

В этом разделе мы обсудим популярные векторные и матричные нормы, которые часто будут возникать в нашем исследовании машинного обучения, в особенности при обсуждении регуляризации. Норма — это своего рода функция, которая измеряет длину векторов и матриц, содержащих действительные значения. Понятие длины является чрезвычайно полезным, поскольку оно позволяет нам определять расстояние (или подобие) между любыми двумя векторами (или матрицами), располагающимися в одном и том же пространстве.

П3.5.1. Векторные нормы

ℓ_2-норма

Мы начнем с наиболее широко используемой векторной нормы в машинном обучении, ℓ_2-нормы, определенной для N-мерного вектора $\mathbf{x}$ как

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{n=1}^{N} x_n^2}. \tag{П3.44}$$

Используя ℓ_2-норму, мы можем измерить расстояние между любыми двумя точками $\mathbf{x}$ и $\mathbf{y}$ посредством $\|\mathbf{x}-\mathbf{y}\|_2$, которое является просто длиной вектора, соединяющего $\mathbf{x}$ и $\mathbf{y}$. Например, расстояние между

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \text{ и } \mathbf{y} = \begin{bmatrix} 9 \\ 8 \end{bmatrix} \tag{П3.45}$$

рассчитывается как $\sqrt{(1-9)^2+(2-8)^2} = 10$, как показано наглядно (пунктирной линией) на рис. П3.7.

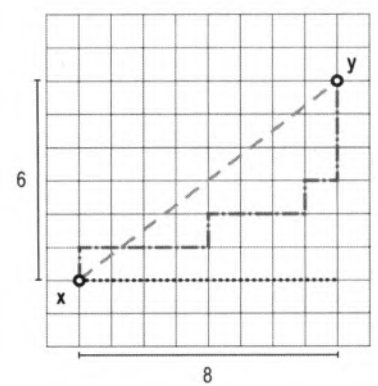


Рис. П3.7. Расстояния на основе ℓ_1 (серая штрихпунктирная), ℓ_2 (серая пунктирная) и ℓ_∞ (черная пунктирная линия) между точками $\mathbf{x}$ и $\mathbf{y}$, определенными в выражении (П3.45)

ℓ_1-норма

ℓ_1-норма вектора $\mathbf{x}$ — это еще один способ измерения его длины, определяемый как сумма абсолютных значений его записей:

$$\|\mathbf{x}\|_1 = \sum_{n=1}^{N} |x_n|. \tag{П3.46}$$

В терминах ℓ_1-нормы расстояние между векторами $\mathbf{x}$ и $\mathbf{y}$ задается выражением $\|\mathbf{x}-\mathbf{y}\|_1$, которое обеспечивает измерение расстояния, отличного от ℓ_2-нормы. Как показано на рис. П3.7, расстояние, определяемое ℓ_1-нормой, представляет собой длину пути, состоящего из перпендикулярных отрезков (показаных черной сплошной линией). Поскольку эти пути в некоторой степени подобны тому, как автомобиль мог бы перемещаться из $\mathbf{x}$ в $\mathbf{y}$, если бы они были двумя точками в городе с организацией улиц в виде сетки, пересекая перпендикулярные городские кварталы один за другим, ℓ_1-норма иногда именуется *таксомоторной нормой*, а расстояние, измеряемое посредством ℓ_1-нормы, — *манхэттенским расстоянием*. Для $\mathbf{x}$ и $\mathbf{y}$ из формулы (П3.45) манхэттенское расстояние вычисляется как $|1-9|+|2-8|=14$.

ℓ_∞-норма

ℓ_∞-норма вектора $\mathbf{x}$ равна его наибольшей записи (в терминах абсолютного значения) и определяется математически как

$$\|\mathbf{x}\|_\infty = \max_n |x_n| \tag{П3.47}$$

Например, расстояние между $\mathbf{x}$ и $\mathbf{y}$ из формулы (П3.45) в терминах ℓ_∞-нормы определяется как $\max\left(|1-9|,\ |2-8|\right)=8$ (см. штрихпунктирную черную линию на рис. П3.7).

П3.5.2. Общие свойства векторных норм

ℓ_2-, ℓ_1- и ℓ_∞-нормы обладают рядом полезных свойств, которые мы подробно рассмотрим ниже. Поскольку эти свойства в общем случае справедливы для любой векторной нормы, мы на мгновение отбрасываем индекс и представляем общую норму вектора $\mathbf{x}$ просто через $\|\mathbf{x}\|$.

1. Нормы всегда неотрицательны, т. е. $\|\mathbf{x}\| \geq 0$ для любого $\mathbf{x}$. Кроме того, равенство соблюдается, если и только если $\mathbf{x}=0$, из чего вытекает, что норма любого ненулевого вектора всегда больше нуля.
2. Норма $\alpha\mathbf{x}$, т. е. скалярного кратного вектора $\mathbf{x}$, может быть записана в терминах нормы вектора $\mathbf{x}$ как $\|\alpha\mathbf{x}\| = |\alpha| \cdot \|\mathbf{x}\|$. Например, при $\alpha=-1$ мы имеем, что $\|-\mathbf{x}\| = \|\mathbf{x}\|$.
3. Нормы также удовлетворяют так называемому *неравенству треугольника*, где для любых трех векторов $\mathbf{x}$, $\mathbf{y}$ и $\mathbf{z}$ мы имеем $\|\mathbf{x}-\mathbf{z}\|+\|\mathbf{z}-\mathbf{y}\| \geq \|\mathbf{x}-\mathbf{y}\|$. Как показано

на рис. П3.8 для ℓ_2-нормы (*слева*), ℓ_1-нормы (*в центре*) и ℓ_∞-нормы (*справа*), неравенство треугольника просто утверждает, что расстояние между **x** и **y** всегда меньше (или равно) расстоянию между **x** и **z** и расстоянию между **z** и **y**, вместе взятыми. Другими словами, если бы кто-то захотел путешествовать из данной точки **x** в данную точку **y**, то всегда было бы лучше путешествовать непосредственно из **x** в **y**, чем путешествовать сначала в третью точку **z**, а затем в **y**. При замене переменных $\mathbf{u} = \mathbf{x} - \mathbf{z}$ и $\mathbf{v} = \mathbf{z} - \mathbf{y}$ неравенство треугольника иногда записывается в более простой форме $\|\mathbf{u}\| + \|\mathbf{v}\| \geq \|\mathbf{u} + \mathbf{v}\|$ для всех векторов **u** и **v**.

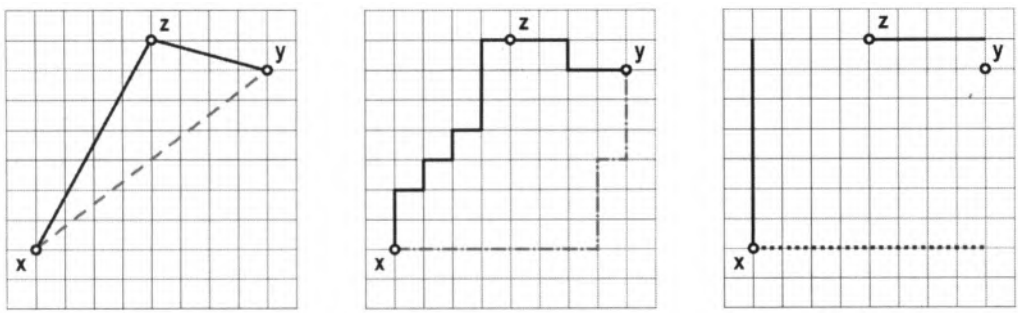


Рис. П3.8. Иллюстрация неравенства треугольника для ℓ_2-нормы (*слева*), ℓ_1-нормы (*в центре*) и ℓ_∞-нормы (*справа*)

В дополнение к общим свойствам, упомянутым выше и соблюдаемым любой нормой, ℓ_2-, ℓ_1- и ℓ_∞-нормы имеют более сильную связь, которая связывает их вместе: все они являются членами семейства ℓ_p-норм. ℓ_p-норма обычно определяется как

$$\|\mathbf{x}\|_p = \left(\sum_{n=1}^{N} |x_n|^p \right)^{\frac{1}{p}} \qquad \text{(П3.48)}$$

для $p \geq 1$. Можно легко проверить, что при $p = 1$, $p = 2$ и по мере того, как $p \to \infty$, ℓ_p-норма сводится соответственно к ℓ_1-, ℓ_2- и ℓ_∞-нормам.

Шары ℓ_p-нормы

Шар нормы — это множество всех векторов **x** с одинаковым значением нормы, т. е. всех **x** таких, что $\|\mathbf{x}\| = c$ для некоторой константы $c > 0$. При $c = 1$ это множество называется *единичным шаром нормы* или просто *единичным шаром*. Единичные ℓ_1-, ℓ_2- и ℓ_∞-шары показаны на рис. П3.9.

ℓ_0-норма

ℓ_0-норма — это еще один способ определения длины вектора как

$$\|\mathbf{x}\|_0 = \text{число ненулевых записей вектора } \mathbf{x} \qquad \text{(П3.49)}$$

Именовать ℓ_0-норму нормой технически неправильно, поскольку она не обладает свойством шкалируемости, присущим всем векторным нормам. То есть, как прави-

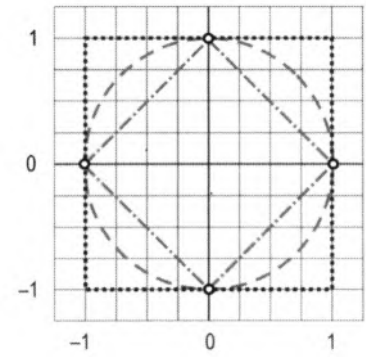


Рис. П3.9. Иллюстрация ℓ_1- (серая штрихпунктирная линия), ℓ_2- (серая пунктирная линия), ℓ_∞- (черная пунктирная линия) единичных шаров

ло, $\|\alpha\mathbf{x}\|_0 \neq |\alpha| \cdot \|\mathbf{x}\|_0$. Тем не менее ℓ_0-норма часто возникает при моделировании векторов с большим числом нулей (так называемых *разреженных векторов*).

П3.5.3. Матричные нормы

Норма Фробениуса

Напомним, что ℓ_2-норма вектора определяется как квадратный корень из суммы квадратов его элементов. *Норма Фробениуса* — это интуитивное расширение ℓ_2-нормы для векторов на матрицы, определяемое схожим образом как квадратный корень из суммы квадратов всех элементов матрицы и записываемое для матрицы $\mathbf{X}$ размера $N \times M$ как

$$\|\mathbf{X}\|_F = \sqrt{\sum_{n=1}^{N}\sum_{m-1}^{M} x_{nm}^2}\,. \qquad (П3.50)$$

Например, норма Фробениуса матрицы $\mathbf{X} = \begin{bmatrix} -1 & 2 \\ 0 & 5 \end{bmatrix}$ вычисляется как

$$\sqrt{(-1)^2 + 2^2 + 0^2 + 5^2} = \sqrt{30}\,.$$

Связь между ℓ_2-нормой и нормой Фробениуса глубже: собрав все сингулярные значения матрицы $\mathbf{X}$ в вектор $\mathbf{s}$, мы имеем

$$\|\mathbf{X}\|_F = \|\mathbf{s}\|_2\,. \qquad (П3.51)$$

Спектральные и ядерные нормы

Наблюдение, что ℓ_2-норма вектора сингулярных значений матрицы идентична ее норме Фробениуса, мотивирует использование других ℓ_p-норм на векторе $\mathbf{s}$. В частности, ℓ_1-норма вектора $\mathbf{s}$ определяет ядерную норму матрицы $\mathbf{X}$, обозначаемую через $\|\mathbf{X}\|_*$:

$$\|\mathbf{X}\|_* = \|\mathbf{s}\|_1\,, \qquad (П3.52)$$

а ℓ_∞-норма вектора $\mathbf{s}$ определяет спектральную норму матрицы $\mathbf{X}$, обозначаемую через $\|\mathbf{X}\|_2$

$$\|\mathbf{X}\|_2 = \|\mathbf{s}\|_\infty . \qquad (П3.53)$$

Поскольку сингулярные значения действительных (вещественных) матриц всегда неотрицательны, спектральная норма и ядерная норма такой матрицы являются соответственно просто ее наибольшим значением и суммой всех ее сингулярных значений.

Список литературы[1]

1. Elson J., Douceur J. R., Howell J., Saul J. Asirra: a CAPTCHA that exploits interest-aligned manual image categorization // Proceedings of ACM Conference on Computer and Communications Security. — 2007. — P. 366–374.

   Asirra: капча, которая использует ручную классификацию изображений, выровненную по интересам.

2. Lee D., Van der Klaauw W., Haughwout A., Brown M., Scally J. Measuring student debt and its performance // FRB of New York StaffReport. — 2014. — № 668.

   Измерение студенческой кредитной задолженности и его результативность.

3. Panaligan R., Chen A. Quantifying movie magic with google search. — Google Whitepaper, 2013.

   Квантификация магии фильмов с помощью поиска в Google.

4. Asur S., Huberman B. A. Predicting the future with social media // Proceedings of IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). — 2010. — Vol. 1. — P. 492–499.

   Предсказание будущего с помощью социальных медиа.

5. Dalal N., Triggs B. Histograms of oriented gradients for human detection // Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition. — 2005. — Vol. 1. — P. 886–893.

   Гистограммы ориентированных градиентов для обнаружения человека.

6. Enzweiler M., Gavrila D. M. Monocular pedestrian detection: survey and experiments // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2009. — Vol. 31, № 12. — P. 2179–2195.

   Монокулярное обнаружение пешеходов: обзор и эксперименты.

7. Maldonado-Bascon S., Lafuente-Arroyo S., Gil-Jimenez P., Gomez-Moreno H., Lopez-Ferreras F. Road-sign detection and recognition based on support vector machines // IEEE Transactions on Intelligent Transportation Systems. — 2007. — Vol. 8, № 2. — P. 264–278.

   Обнаружение и распознавание дорожных знаков на основе машин опорных векторов.

[1] Приведены оригинальные названия источников литературы и переводы названий. — *Прим. ред.*

8. Pang B., Lee L., Vaithyanathan S. Thumbs up?: sentiment classification using machine learning techniques // Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing. — 2002. — Vol. 10. — P. 79–86.

   Большие пальцы вверх? Классификация сентимента с использованием приемов машинного обучения.

9. Hammer R., Booth J. R., Borhani R., Katsaggelos A. K. Pattern analysis based on fMRI data collected while subjects perform working memory tasks allowing high-precision diagnosis of ADHD. — US Patent App., 2017. 15317724.

   Анализ шаблонов на основе данных фМРТ, собранных во время выполнения испытуемыми задач оперативной памяти, позволяет с высокой точностью диагностировать СДВГ.

10. Maclaurin D., Duvenaud D., Adams R. P. Autograd: reverse-mode diffrentiation of native Python // ICML Workshop on Automatic Machine Learning. — 2015.

    Autograd: обратный режим дифференцирования на нативном Python.

11. Johnson M., Frostig R., Leary C. Compiling machine learning programs via high-level tracing // Systems and Machine Learning (SysML). — 2018.

    Компиляция программ машинного обучения с помощью высокоуровневой трассировки.

12. Baydin A. G., Pearlmutter B. A., Radul A. A., Siskind J. M. Automatic differentiation in machine learning: a survey // Journal of Marchine Learning Research. — 2018. — Vol. 18. — P. 1–43.

    Автоматическое дифференцирование в машинном обучении: обзор.

13. Neidinger R. D. Introduction to automatic diffrentiation and MATLAB object-oriented programming // SIAM Review. — 2010. — Vol. 52, № 3. — P. 545–563.

    Введение в автоматическое дифференцирование и объектно-ориентированное программирование на MATLAB.

14. Luenberger D. G., Linear and Nonlinear Programming. — Springer, 2003.

    Линейное и нелинейное программирование.

15. Boyd S. P., Vandenberghe L. Convex Optimization. — Cambridge University Press, 2004.

    Выпуклая оптимизация.

16. Harrison Jr. D., Rubinfeld D. L. Hedonic housing prices and the demand for clean air // Journal of Environmental Economics and Management. — 1978. — Vol. 5, № 1. — P. 81–102.

    Гедонистические цены на жилье и спрос на чистый воздух.

17. Dua D., Graff C. Auto MPG dataset. UCI Machine Learning Repository. — 2017. — URL: https://archive.ics.uci.edu/ml/datasets/auto+mpg.

    Набор данных Auto MPG об автомобильном пробеге в милях на галлон израсходованного топлива.

18. Cortes C., Vapnik V. Support-vector networks // Machine Learning. — 1995. — Vol. 20, № 3. — P. 273–297.

Опорно-векторные сети.

19. Boyd S., Parikh N., Chu E., Peleato B., Eckstein J. Distributed optimization and statistical learning via the alternating direction method of multipliers // Foundations and Trends R in Machine Learning. — 2011. — Vol. 3, № 1. — P. 1–122.

Распределенная оптимизация и статистическое обучение посредством переменно-направленного метода множителей.

20. Scholkopf B., Smola A. J. Learning with kernels: support vector machines, regularization, optimization, and beyond. — MIT Press, 2002.

Обучение с помощью ядер.

21. Bottou L. Large-scale machine learning with stochastic gradient descent // Proceedings of COMPSTAT'2010. — 2010. — P. 177–186.

Крупномасштабное машинное обучение с помощью стохастического градиентного спуска.

22. Chapelle O. Training a support vector machine in the primal // Neural Computation. — 2007. — Vol. 19, № 5. — P. 1155–1178.

Тренировка машины опорных векторов в примали.

23. Dua D., Graff C. Spambase dataset. UCI Machine Learning Repository. — 2017. — URL: https://archive.ics.uci.edu/ml/datasets/spambase.

Набор данных Spambase для обнаружения спама.

24. Dua D., Graff C. Statlog dataset. UCI Machine Learning Repository. — 2017. — URL: https://archive.ics.uci.edu/ml/datasets/statlog+(german+credit+data).

Набор данных Statlog с кредитными данными немецкого банка.

25. Rifkin B. R., Klautau A. In defense of one-vs-all classification // Journal of Machine Learning Research. — 2004. — Vol. 5. — P. 101–141.

Защита классификации "один против всех".

26. Tang Y. Deep learning using support vector machines. — CoRR, 2013, abs/1306.0239.

Глубокое обучение с использованием опорно-векторных машин.

27. Dua D., Graff C. Iris dataset. UCI Machine Learning Repository. — 2017. — URL: https://archive.ics.uci.edu/ml/datasets/iris.

Набор данных Iris с данными о цветках ириса.

28. Bertsekas D. P. Incremental gradient, subgradient, and proximal methods for convex optimization: a survey // Optimization for Machine Learning. — 2011. — Vol. 2010. — P. 1–38.

Поступательный градиентный, субградиентный и проксимальный методы выпуклой оптимизации: обзор.

29. LeCun Y., Cortes C. MNIST handwritten digit database. — 2010. — URL: http://yann.lecun.com/exdb/mnist/.

MNIST: база данных рукописных цифр.

30. Olshausen B. A., Field D. J. Sparse coding with an overcomplete basis set: a strategy employed by V1? // Vision Research. — 1997. — Vol. 37, № 23. — P. 3311–3325.

Разреженное кодирование с чрезмерно полным базисным множеством: стратегия, используемая V1?

31. Lee D. D., Seung H. S. Algorithms for nonnegative matrix factorization // Advances in Neural Information Processing Systems. — 2001. — P. 556–562.

Алгоритмы неотрицательной матричной факторизации.

32. Manning C. D., Schutze H. Foundations of Statistical Natural Language Processing. — MIT Press, 1999.

Основы статистической обработки естественного языка.

33. Barlow H. The coding of sensory messages // Current Problems in Animal Behaviour. — 1961. — P. 331–360.

Кодирование сенсорных сообщений.

34. Barlow H. Redundancy reduction revisited // Network: Computation in Neural Systems. — 2001. — Vol. 12, № 3. — P. 241–253.

Редукция избыточности повторно.

35. Prince S. J. Computer Vision: Models, Learning, and Inference. — Cambridge University Press, 2012.

Компьютерное зрение: модели, обучение и логический вывод.

36. LeCun Y., Kavukcuoglu K., Farabet C. Convolutional networks and applications in vision // Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS). — 2010. — P. 253–256.

Сверточные сети и приложения в области компьютерного зрения.

37. LeCun Y., Bengio Y. Convolutional networks for images, speech, and time series // The Handbook of Brain Theory and Neural Networks. — 1995. — Vol. 3361, № 10.

Сверточные сети для изображений, речи и временных рядов.

38. Krizhevsky A., Sutskever I., Hinton G. E. Imagenet classification with deep convolutional neural networks // Advances in Neural Information Processing Systems. — 2012. — P. 1097–1105.

Классификация базы данных Imagenet с помощью глубоких сверточных нейронных сетей.

39. Marcelja S. Mathematical description of the responses of simple cortical cells // JOSA. — 1980. — Vol. 70, № 11. — P. 1297–1300.

Математическое описание реакций простых кортикальных клеток.

40. Jones J. P., Palmer L. A. An evaluation of the two-dimensional gabor filter model of simple receptive fields in cat striate cortex // Journal of neurophysiology. — 1987. — Vol. 58, № 6. — P. 1233–1258.

Оценивание двумерной модели на основе фильтра Габора простых рецептивных полей в полосатой коре головного мозга кошек.

41. Huang X., Acero A., Hon H. W. Spoken language processing: a guide to theory, algorithm and system development. — Prentice Hall, 2001.

Обработка устной речи: руководство по теории, алгоритмам и развитию системы.

42. Rabiner L. R., Juang B. H. Fundamentals of Speech Recognition. — Prentice Hall, 1993.

Основы распознавания речи.

43. Dua D., Graff C. Breast cancer Wisconsin (diagnostic) dataset. The University of California, Irvine (UCI) Machine Learning Repository. — 2017. — URL: https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic).

Висконсинский (диагностический) набор данных по раку молочной железы.

44. Galilei G. Dialogues Concerning Two New Sciences. — Dover, 1914.

Диалоги о двух новых науках.

45. Straulino S. Reconstruction of Galileo Galilei's experiment: the inclined plane // Physics Education. — 2008. — Vol. 43, № 3. — P. 316.

Реконструкция эксперимента Галилео Галилея: наклонная плоскость.

46. Lin J., Lee S. M., Lee H. J., Koo Y. M. Modeling of typical microbial cell growth in batch culture // Biotechnology and Bioprocess Engineering. — 2000. — Vol. 5, № 5. — P. 382–385.

Моделирование роста типичных микробных клеток в пакетной культуре.

47. Moore G. E. Cramming more components onto integrated circuits // Proceedings of the IEEE. — 1998. — Vol. 86, № 1. — P. 82–85.

Запихивание большего числа компонентов в интегральные схемы.

48. Mayer V., Varaksina E. Modern analogue of Ohm's historical experiment // Physics Education. — 2014. — Vol. 49, № 6. — P. 689.

Современный аналог исторического эксперимента Ома.

49. Rudin W. Principles of Mathematical Analysis. — McGraw-Hill New York, 1964.

Принципы математического анализа.

50. Cybenko G. Approximation by superpositions of a sigmoidal function // Mathematics of Control, Signals and Systems. — 1989. — Vol. 2, № 4. — P. 303–314.

Аппроксимация по суперпозициям сигмоидной функции.

51. Park J., Sandberg I. W. Universal approximation using radial-basis-function networks // Neural Computation. — 1991. — Vol. 3, № 2. — P. 246–257.

Универсальная аппроксимация с использованием сетей на основе радиальных базисных функций.

52. Hornik K., Stinchcombe M., White H. Multilayer feedforward networks are universal approximators // Neural Networks. — 1989. — Vol. 2, № 5. — P. 359–366.

Многослойные сети прямого распространения являются универсальными аппроксиматорами.

53. Rahimi A., Recht B. Uniform approximation of functions with random bases // Proceedings of the 46th Annual Allerton Conference on Communication, Control, and Computing. — 2008. — P. 555–561.

Равномерная аппроксимация функций со случайными базисами.

54. Rahimi A., Recht B. Random features for large-scale kernel machines // Advances in Neural Information Processing Systems. — 2008. — P. 1177–1184.

Случайные признаки для крупномасштабных ядерных машин.

55. Buhrman H., Wolf R. De Complexity measures and decision tree complexity: a survey // Theoretical Computer Science. — 2002. — Vol. 288, № 1. — P. 21–43.

Меры сложности и сложность дерева решений: обзор.

56. Osgood B. Lectures on the Fourier Transform and Its Applications. — American Mathematical Society, 2019.

Лекции по преобразованию Фурье и его применению.

57. MacKay D. J. Introduction to Gaussian processes // NATO ASI Series F Computer and Systems Sciences. — 1998. — Vol. 168. — P. 133–166.

Введение в гауссовы процессы.

58. Bishop C. M. Pattern Recognition and Machine Learning. — Springer, 2011.

Распознавание шаблонов и машинное обучение.

59. Rosenblatt F. The perceptron, a perceiving and recognizing automaton. — Cornell Aeronautical Laboratory, 1957.

Персептрон: воспринимающий и распознающий автомат.

60. Glorot X., Bordes A., Bengio Y. Deep sparse rectifier networks // Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. — 2001. — Vol. 15. — P. 315–323.

Глубокие разряженные сети на основе выпрямителя.

61. Goodfellow I. J., Warde-Farley D., Mirza M., Courville A., Bengio Y. Maxout networks // arXiv preprint arXiv. — 2013. — P. 1302.4389.

Сети на основе активациии maxout.

62. Ioffe S., Szegedy C. Batch normalization: accelerating deep network training by reducing internal covariate shift // arXiv preprint arXiv. — 2015. — P. 1502.03167.

Пакетная нормализация: ускорение тренировочного процесса глубокой сети за счет редукции внутреннего ковариатного сдвига.

63. Huang G., Li Y., Pleiss G., Liu Z., Hopcroft J. E., Weinberger K. Q. Snapshot ensembles: train 1, get M for free // arXiv preprint arXiv. — 2017. — 1704.00109.

Моментальные ансамбли: тренируешь 1, получаешь M бесплатно.

64. L Smith. N. Cyclical learning rates for training neural networks // Proceedings of 2017 IEEE Winter Conference on Applications of Computer Vision (WACV). — 2017. — P. 464–472.

Циклические скорости обучения для тренировки нейронных сетей.

65. Breiman L., Friedman J., Stone C. J., Olshen R. A. Classification and Regression Trees. — Chapman and Hall CRC, 1984.

Классификационные и регрессионные деревья.

66. Friedman J. H. Greedy function approximation: a gradient boosting machine // Annals of Statistics. — 2001. — P. 1189–1232.

Аппроксимация жадной функции: машина бустирования градиента.

67. Chen T., Guestrin C. XGBoost: a scalable tree boosting system // Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. — 2016. — P. 785–794.

XGBoost: масштабируемая система бустирования деревьев.

68. Liaw A., Wiener M. Classification and regression by randomForest // R News. — 2002. — Vol. 2, № 3. — P. 18–22.

Классификация и регрессия с помощью randomForest.

69. Kingma D. P., Ba J. Adam: a method for stochastic optimization // arXiv preprint arXiv. — 2014. — P. 1412.6980.

Adam: метод для стохастической оптимизации.

70. Tieleman T., Hinton G. Lecture 6a: Overview of Mini-Batch Gradient Descent // Neural Networks for Machine Learning (Coursera Lecture Slides). — 2012.

Лекция 6а: обзор мини-пакетного градиентного спуска. Нейронные сети для машинного обучения.

71. Bishop C. M. Neural Networks for Pattern Recognition. — Oxford University Press, 1995.

Нейронные сети для распознавания шаблонов.

72. Fletcher R. Practical Methods of Optimization. — John Wiley & Sons, 2013.

Практические методы для оптимизации.

73. Broyden C. G. The convergence of a class of double-rank minimization algorithms // Journal of the Institute of Mathematics and Its Applications. — 1970. — Vol. 6, № 3–4. — P. 76–90.

Сходимость семейства алгоритмов минимизации с двойным рангом.

74. Boyd S., Vandenberghe L. Introduction to applied linear algebra: vectors, matrices, and least squares. — Cambridge University Press, 2018.

Введение в прикладную линейную алгебру.

75. Lobo M., Vandenberghe L., Boyd S., Lebret H. Applications of second-order cone programming // Linear Algebra and Its Applications. — 1998. — Vol. 284, № 1. — P. 193–228.

Приложения программирования конусов второго порядка.

76. Trefethen L. N., Bau D. III. Numerical Linear Algebra. — Philadelphia: SIAM: Society for Industrial and Applied Mathematics, 1997. — 376 p.

Численная линейная алгебра.

Предметный указатель

A

Adam *См.* Оценивание импульса адаптивное
autograd 71, 79, 571

J

JAX 71, 571

K

K-средних 243

P

PCA-сферирование 280, 282, 283
Python 71, 79, 106, 121, 136, 151, 156, 210, 240, 307, 311, 316, 320, 450

R

RMSProp 461, 519

S

SVM:
◊ на основе логарифмической потери 176
◊ машина с маржой:
  ▫ жесткой 174
  ▫ мягкой 174

A

Автокодировщик:
◊ линейный 235, 236, 255
◊ нелинейный 321–323, 453, 454, 473
Агрегирование *См.* Бутстрап-агрегирование
Активация maxout 458
Алгоритм жадный 287
Анализ:
◊ главных компонент (PCA) 235, 237
◊ настроений 24
  См. также Анализ сентиментный
◊ сентиментный 24, 265
Ансамблирование 403, 477
Аппроксиматор универсальный:
◊ древесный 345
◊ нейросетевой 344
◊ фиксированно-контурный 343
Аппроксимация:
◊ универсальная 333, 341
◊ функции 339

Б

Базис:
◊ ортонормальный 232
◊ Фурье 415
Бустирование:
◊ отбор признаков 283
◊ перекрестная валидация 368
◊ градиента 374, 490
Бутстрап-агрегирование
◊ классификация 399
◊ регрессия 397

В

Валидация:
◊ модели 16
◊ перекрестная:
▫ K-блочная 404
▫ на основе бустирования 368
▫ на основе регуляризации 379
▫ наивная 362
• слабости 368
▫ с исключением по одному 406
Вектор:
◊ линейная комбинация векторов 585
◊ магнитуда 583
◊ нормы 593
◊ операции арифметические 581
◊ произведение:
▫ внешнее 585
▫ внутреннее 583
▫ поэлементное 583
◊ собственный 590, 591
◊ стандартный базисный 54
◊ умножение вектора на скаляр 583
Вес:
◊ наклона 116
◊ смещения 116
Взятие подвыборок из гессиана 536
Вменение значений пропущенным элементам *См.* Импутирование
Вменение среднего значения 279
Вогнутость 94
Выброс 125, 127, 129, 398, 399
Выделение основ слов 264
Выпуклость 66, 73, 94, 98, 108, 119, 532
Выравнивание математических функций 577
Вычисление градиента 72, 571

Г

Галилей 130, 305
◊ эксперимент с рампой 130
Геометрия квадратичная 94
Гиперплоскость 67
Градиент 549
Граница решения 143
Граф вычислительный 550
График:
◊ истории стоимостной функции 48, 77
◊ контурный 43
◊ крутого склона 248, 255

Д

Данные:
◊ несбалансированные 183, 187
◊ отсутствующие 278
Дерево 345, 474
◊ бустирование градиента 490
◊ глубое, созданное посредством:
▫ рекурсии 475
▫ сложения 477
◊ классификационное 484
◊ перекрестная валидация 496
◊ подрезка 498
◊ ранняя остановка 496
◊ регрессионное 477
◊ рекурсивно выстраиваемое 482
◊ случайные леса 494
Диагностика компьютерная патологических состояний здоровья 24
Диск наборный:
◊ емкостный 347
◊ оптимизационный 347, 357
◊ сложностный 332
Дифференцирование:
◊ автоматическое 79, 106, 308, 450, 458, 544, 553
◊ правило 71
◊ символическое 558
◊ численное 546
◊ обратный режим 458, 559
◊ прямой режим 553
Длина шага 523
◊ консервативная, правило 528
◊ корректируемая, правило 49
◊ параметр 44, 105, 523
◊ проблемы 44, 76, 78
◊ убывающая, правило 49, 51, 59, 76
◊ фиксированная, правило 45, 49, 76, 528

Е

Единица линейная выпрямленная 160, 457

З

Задача:
◊ матричной факторизации 249
◊ неотрицательной факторизации матриц 252
◊ отыскания нуля 105

Закон:
◊ Клейбера 21, 138
◊ Мура 325
◊ Ома 326
Заучивание признаков *См.* Усвоение признаков
Значение:
◊ собственное 91, 94, 95, 590–592
◊ корреляции 265

И

Имплементация на языке Python:
◊ анализ главных компонент (PCA) 240
◊ выравнивание функции 577
◊ градиентный спуск 79
◊ классификация:
  ▫ многоклассовая 210
  ▫ нелинейная двухклассовая 316
◊ метод Ньютона 106
◊ нейронная сеть 450
◊ регрессия:
  ▫ линейная 121
  ▫ многовыходная 136
  ▫ нелинейная 307
Импутирование 278
Инженерия признаков 313, 322
Интерпретируемость человеком 283, 377, 390, 396, 404, 405, 408, 477, 494
Исследование ассоциаций полногеномное (GWAS) 408

К

Калькулятор градиентный 71
Квантизация 130
Классификатор 17
◊ на основе максимальной маржи 172
Классификация:
◊ «один против всех» 212, 219, 227
◊ взвешенная 187
◊ двухклассовая:
  ▫ взвешенная 187
  ▫ линейная, нотация и моделирование 143
◊ метрики качества 179
◊ многоклассовая 22
◊ нелинейная 312, 316, 330, 434, 473
Кластеризация 27, 28
◊ на основе K-средних 243
Кодирование:
◊ разряженное 252
◊ с одним активным состоянием 177, 217
Константа Липшица 528
Конструирование признаков 14
Корень из среднеквадратической ошибки 129
Кривизна 91, 523

Л

Лес случайный 494
Линия:
◊ касательная 544
◊ секущая 544

М

Максимум глобальный 38
Маржа 169
Матрица 587
◊ арифметические операции 588
◊ Гесса 108, 152, 528
  ▫ положительная полуопределенная *См.* Полудефинит положительный
◊ диагональная 255, 282
◊ квадратная 590
◊ нормы 593
◊ путаницы 185
◊ симметричная 592
Машина опорных векторов жесткомаржевая 174
Медиана (статистика) 397
Мера сходства 427
Метод:
◊ бесгессианов 535
◊ Бройдена — Флетчера — Гольдфарба — Шанно (BFGS) 112, 542
◊ Дэвидона — Флетчера — Пауэлла (DFP) 541, 542
◊ квазиньютонов 539
  ▫ квазиньютонов с низким потреблением памяти 542
◊ Ньютона 532
  ▫ алгоритм 100
  ▫ иллюстрация 101
  ▫ имплементация на языке Python 106
  ▫ ограничения по масштабированию 108
  ▫ с точки зрения отыскания нуля 105
  ▫ связь с градиентным спуском 100
  ▫ сравнение с градиентным спуском 101

- схождение 101
- числовая стабильность 104

◊ опорных векторов 169
◊ секущей 537
Метрика:
◊ качества:
- классификация 179
- регрессия 127

◊ ошибка среднеквадратическая 129
◊ точность 220
Мешок слов (BoW) 264, 266
Минимум глобальный 38
Многообразие топологическое 27, 28, 321, 322, 327, 453, 454
Множество охватывающее 229
См.также Базис ортонормальный
Мода (статистика) 397, 485
Моделирование многочисленных волн 311
Модель ядрированная:
◊ классификация:
- двухклассовая 420
- многоклассовая 419

◊ регрессия 419

H

Набор данных:
◊ Auto-MPG, об автомобильном пробеге в милях на галлон израсходованного топлива 130
◊ Iris о цветках ириса 223
◊ валидационный 16
◊ данные:
- в поддержку закона Клейбера 21
- в поддержку закона Мура 325
- в поддержку закона Ома 326
- для задачи обнаружения спама 186, 493
- кредитные немецкого банка 186, 287, 292
- о росте численности популяции бактерий 324
- о студенческой кредитной задолженности 20
- о ценах на жилье в Бостоне 129, 140, 286, 292
- по раку молочной железы 193, 429

◊ рукописных изображений MNIST 225, 272, 294, 296, 461, 469
◊ тестовый 393
◊ тренировочный 14
Направление:
◊ подъема 68
◊ спуска 43
Невыпуклость 459
Недоподгонка 352, 361
Нейрон 449
Неравенство треугольника 594
Норма:
◊ векторная/матричная 593
◊ свойства 594
◊ спектральная 596
◊ Фробениуса 596
◊ ядерная 596
Нормализация:
◊ пакетная 448, 462
◊ стандартная 274, 282, 286, 294, 296

O

Обнаружение:
◊ лиц 23, 142
◊ объектов 23
◊ ребер (краев изображения) 269
◊ спама 25, 266
Обучение машинное 13
Один против всех:
◊ алгоритм 202
◊ нотация и моделирование 194
Оптимизация:
◊ бесгессианова:
- взятие подвыборок 536
- квазиньютонова 535
- метод секущей 537

◊ второпорядковая 91
◊ глобальная 38
◊ локальная 41
◊ математическая 29
◊ мини-пакетная 224, 520
◊ нуль-порядковая 35
◊ ограниченная 174
◊ первопорядковая 60
◊ стохастическая 522
Остановка ранняя 470
◊ бустирование 374
◊ регуляризация 381
Остаток 288, 378, 492
Отбеливание 280
Отбор:
◊ образцов:
- равномерный 41
- случайный 39

◊ признаков 129

Отклонение наименьшее абсолютное 125
Отношение Рэлея 63, 88, 258
Оценивание импульса адаптивное 518
Ошибка:
◊ валидационная 351, 361
◊ среднеквадратическая (MSE) 129
◊ тестовая 393

П

Параметр длины шага 44
Пень (дерева) 474
Переподгонка 352, 361
Персептрон 158
◊ маржевой 169
Подрезка 496, 498
Подъем самый крутой 67
Поиск:
◊ координатный 53
◊ линейный:
▫ по прямой 105
▫ с отслеживанием в обратном порядке 525
▫ точный 527
◊ случайный 45
▫ локальный 52
Полудефинит положительный 91
Правило:
◊ внутреннего произведения 69
◊ слияния 201, 317, 318
Предиктор *См.* Предсказатель
Предсказание:
◊ автомобильного пробега в милях на галлон израсходованного топлива 130, 140, 297
◊ цен на жилье 129, 140, 296
Предсказатель 286
Прецизионность машины 104
Признак 14
◊ гистограммный 262
▫ для аудиоданных 272
▫ для категориальных данных 262
▫ для текстовых данных 263
▫ для фотоснимковых данных 267
◊ инженерия 7, 14, 262, 301
◊ отбор 129, 283, 288
◊ усвоение 16
◊ шкалирование 274
Применение:
◊ задачи распознавания объектов 26
◊ классификация:
▫ домашних животных на кошек и собак 13
▫ рака 193, 412, 429
◊ компьютерная диагностика патологических состояний здоровья 24
◊ обнаружение:
▫ объектов 23
▫ спама 25, 186, 263, 266, 493
◊ оценивание уровня кредитного риска 186, 287, 292, 297
◊ полногеномные исследования ассоциаций 408
◊ предсказание:
▫ кассового успеха 21
▫ студенческой кредитной задолженности 20
◊ применения в бизнесе и промышленности 22
◊ распознавание:
▫ объектов 22, 267
▫ рукописных цифр 23, 26, 225, 228, 272, 470, 473
◊ сентиментный анализ 24, 142, 263, 265
◊ финансовое 26
Прирост информации 488
Проблема исчезающего градиента 456, 512
Прогнозирование выручки 21
Произведение Адамара 583
Производная 544
◊ более высокого порядка 563
Проклятие размерности 40
Псевдоинверсия 101

Р

Разделение линейное 158
Размер пакета 521
Распознавание:
◊ объектов 23, 26
◊ речи 26, 272
◊ рукописных цифр 22, 272
Распределение вероятностей дискретное 213
Распространение обратное 458, 559
Расстояние:
◊ знаковое 216
◊ манхэттенское 594
Регрессия:
◊ взвешенная 130

◊ линейная 117
  ▫ имплементация на языке Python 121
  ▫ корень из среднеквадратической ошибки 129
  ▫ нотация и моделирование 115
  ▫ работа с дубликатами 130
  ▫ стоимость:
    • наименьших абсолютных отклонений 125
    • наименьших квадратов 117
◊ логистическая 147
◊ метрики качества 127
◊ многовыходная 133
  ▫ имплементация на языке Python 136
  ▫ нелинейная 308
  ▫ нотация и моделирвание 133
  ▫ стоимостные функции 134
◊ нелинейная 301, 308, 330, 434, 474
Регуляризация 104, 533
◊ отбор признаков 288
◊ перекрестная валидация 379
Редукция размерности 27
Режим:
◊ обратный автоматического дифференцирования 559
◊ прямой автоматического дифференцирования 553
Решение большинством голосов 486
Ряд:
◊ временной 506
◊ гармонический 51
◊ Тейлора 4, 60, 67, 73, 91, 94, 96, 97, 102, 523, 533, 537, 544, 565, 569

C

Самообучение:
◊ контролируемое 19
  ▫ классификация 22
  ▫ регрессия 19
◊ неконтролируемое 27
◊ онлайновое 224
Сбор данных 14
Секущая 544
Сентимент-анализ 24
Сеть нейронная 344, 434
◊ активационные функции 455
◊ биологическая 448, 455
◊ валидация перекрестная посредством ранней остановки 470
◊ имплементация на языке Python 450
◊ невыпуклость 459
◊ нормализация пакетная 462
◊ оптимизация 459
◊ полносвязная 434
◊ представление:
  ▫ графовое 450
  ▫ компактное 438
◊ распространение обратное 458
◊ рекурсивно выстраиваемая 434
◊ с двумя скрытыми слоями 439
◊ с многочисленными скрытыми слоями 444
◊ с одним скрытым слоем 434
◊ точка зрения биологическая 448
Сжатие изображений 267
Синдром дефицита внимания и гиперактивности (СДВГ) 25, 442
Система:
◊ линейных уравнений симметричная 65
◊ рекомендательная 241
◊ уравнений первопорядковая 61
Скорость заучивания (самообучения) 45
Слой скрытый 437
Спектрограмма 273
Спуск:
◊ градиентный:
  ▫ алгоритм 72, 78, 102
  ▫ иллюстрация 73
  ▫ имплементация на языке Python 79
  ▫ импульсно-ускоренный 505
  ▫ критерии схождения 78
  ▫ мини-пакетный 223
  ▫ нормализация покомпонентная 514
  ▫ нормализованный 511
  ▫ поведение:
    • зигзагообразное 81, 84, 505
    • медленно ползающее 85, 511
  ▫ слабости 80
  ▫ ускорение импульса 505
◊ координатный 54, 65
◊ самый крутой 67
Среднее:
◊ значение экспоненциальное 505
◊ кумулятивное 506, 507
◊ статистика 484
◊ экспоненциальное 506
Стоимость:
◊ softmax *См.* Стоимость мягкого максимума
◊ мягкого максимума 152
◊ наименьшие:
  ▫ абсолютные отклонения 125
  ▫ квадраты 119

Стоимость *(прод.)*:
◊ персептрона 158
◊ петлевая 161
◊ энтропия перекрестная 150
▫ категориальная 176, 213
Стоп-слово 264
Суррогат квадратичный 523
Схождение 46, 78, 101, 530

Т

Теорема линейной алгебры фундаментальная 418
Точка:
◊ седловая 62, 78, 85, 93, 459, 511, 512
◊ стационарная 62, 63, 65, 78, 85, 87, 93, 98–100, 511, 517, 538
Точность 181
◊ сбалансированная 183
Тренировка модели 16
Трюк ядерный 417

У

Узел:
◊ дочерний 551
◊ родительский 551
Усвоение признаков 330, 409
Ускорение импульса 505
Условие оптимальности:
◊ второпорядковое 91
◊ нуль-порядковое 37
◊ первопорядковое 60

Ф

Факторизация матричная неотрицательная 252
фМРТ 25, 409
Функция:
◊ активационная 434, 450, 455
◊ гиперболического тангенса 154
◊ квадратичная 36, 39, 47, 50, 52, 55–57, 60, 65–67, 83, 88, 92
◊ радиально-базисная 424
◊ сигмоидная логистическая 147
◊ стоимостная 29
▫ наименьших квадратов 117
• минимизация 119
• подверженность к выбросам 125
◊ экспоненциальная нормализованная 214

Ч

Чистота 488

Э

Эпоха 521

Я

Ядро:
◊ валидация перекрестная 429
◊ как мера сходства 427
◊ многочленное 422
◊ оптимизация 428
◊ радиальной базисной функции (RBF) 424
◊ Фурье 423



Коды всех примеров и более 300 цветных иллюстраций можно скачать по ссылке ftp://ftp.bhv.ru/9785977567633.zip, а также со страницы книги на сайте **bhv.ru**

Благодаря интуитивно понятному, но строгому подходу к машинному обучению эта книга предоставляет фундаментальные знания и практические инструменты, необходимые для проведения исследований и разработки систем машинного обучения. В книге приведено более 100 углубленных упражнений на языке Python. Дано введение в машинное обучение и математическую оптимизацию, включая методы первого и второго порядков, градиентного спуска и Ньютона. Отдельно рассмотрены продвинутые методы оптимизации. Приведено полное описание обучения с учителем, включая линейную регрессию, двухклассовую и многоклассовую классификацию, а также обучение без учителя и фундаментальные методы генерации признаков. Дано введение в нелинейное обучение с учителем и без. Обсуждается тема автоматизированного отбора подходящих нелинейных моделей, включая перекрестную валидацию, бустирование, регуляризацию и ансамблирование. Рассмотрены фиксированно-контурные ядра, нейронные сети, деревья и другие универсальные аппроксиматоры.

КЛЮЧЕВЫЕ ОСОБЕННОСТИ

- Интуитивно понятные объяснения
- Доступный подход к современным методам численной оптимизации
- Комплексное введение в логистическую регрессию и машины опорных векторов
- Представление сложных тем через призму аппроксимации функций
- Уточненное описание глубоких нейронных сетей и методов ядра

bhv®

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-6763-3