O'REILLY®

2-е издание

Python для сложных задач

Наука о данных

SPRINT book

Джейк Вандер Плас

SECOND EDITION

Python Data Science Handbook

Essential Tools for Working with Data

Jake VanderPlas

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

2-е издание

Python для сложных задач

Наука о данных

Джейк Вандер Плас

SPRINT book

2024

ББК 32.973
УДК 004
П37

Плас Вандер Джейк

П37 Python для сложных задач: наука о данных. 2-е междунар. изд. — Астана: «Спринт Бук», 2024. — 592 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-601-08-3564-1

Python — первоклассный инструмент, и в первую очередь благодаря наличию множества библиотек для хранения, анализа и обработки данных. Отдельные части стека Python описываются во многих источниках, но только в новом издании «Python для сложных задач» вы найдете подробное описание IPython, NumPy, pandas, Matplotlib, Scikit-Learn и др.

Специалисты по обработке данных, знакомые с языком Python, найдут во втором издании решения таких повседневных задач, как обработка, преобразование и подготовка данных, визуализация различных типов данных, использование данных для построения статистических моделей и моделей машинного обучения. Проще говоря, эта книга является идеальным справочником по научным вычислениям в Python.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973
УДК 004

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1098121228 англ.

Authorized Russian translation of the English edition of Python Data Science Handbook, 2nd Edition ISBN 9781098121228 © 2023 Jake VanderPlas.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-601-08-3564-1

© Перевод на русский язык ТОО «Спринт Бук», 2023
© Издание на русском языке, оформление ТОО «Спринт Бук», 2023
© Серия «Бестселлеры O'Reilly», 2023

Краткое содержание

https://t.me/it_boooks/2

Предисловие 19

ЧАСТЬ I. JUPYTER: ЗА ПРЕДЕЛАМИ ОБЫЧНОГО PYTHON

Глава 1. Знакомство с IPython и Jupyter 27
Глава 2. Расширенные интерактивные возможности 38
Глава 3. Отладка и профилирование 48

ЧАСТЬ II. ВВЕДЕНИЕ В NUMPY

Глава 4. Типы данных в Python 64
Глава 5. Введение в массивы NumPy 72
Глава 6. Вычисления с массивами NumPy: универсальные функции 81
Глава 7. Агрегирование: минимум, максимум и все, что посередине 91
Глава 8. Операции над массивами. Транслирование 97
Глава 9. Сравнения, маски и булева логика 104
Глава 10. «Прихотливая» индексация 114
Глава 11. Сортировка массивов 123
Глава 12. Структурированные данные: структурированные массивы NumPy 130

ЧАСТЬ III. МАНИПУЛЯЦИИ НАД ДАННЫМИ С ПОМОЩЬЮ ПАКЕТА PANDAS

Глава 13. Знакомство с объектами библиотеки Pandas 138
Глава 14. Индексация и выборка данных 147
Глава 15. Операции над данными в библиотеке Pandas 155
Глава 16. Обработка отсутствующих данных 161
Глава 17. Иерархическая индексация 171
Глава 18. Объединение наборов данных: конкатенация и добавление в конец 185
Глава 19. Объединение наборов данных: слияние и соединение 191
Глава 20. Агрегирование и группировка 205
Глава 21. Сводные таблицы 218
Глава 22. Векторизованные операции над строками 227
Глава 23. Работа с временными рядами 237
Глава 24. Увеличение производительности библиотеки Pandas: eval() и query() 259

ЧАСТЬ IV. ВИЗУАЛИЗАЦИЯ С ПОМОЩЬЮ БИБЛИОТЕКИ MATPLOTLIB

Глава 25. Общие советы по библиотеке Matplotlib ... 269
Глава 26. Простые линейные графики ... 276
Глава 27. Простые диаграммы рассеяния ... 287
Глава 28. Графики плотности и контурные графики ... 297
Глава 29. Настройка легенд на графиках ... 308
Глава 30. Настройка цветовых шкал ... 315
Глава 31. Множественные субграфики ... 324
Глава 32. Текст и поясняющие надписи ... 332
Глава 33. Настройка делений на осях координат ... 340
Глава 34. Настройка Matplotlib: конфигурации и таблицы стилей ... 349
Глава 35. Построение трехмерных графиков в библиотеке Matplotlib ... 358
Глава 36. Визуализация с помощью библиотеки Seaborn ... 368

ЧАСТЬ V. МАШИННОЕ ОБУЧЕНИЕ

Глава 37. Что такое машинное обучение ... 389
Глава 38. Знакомство с библиотекой Scikit-Learn ... 401
Глава 39. Гиперпараметры и проверка модели ... 419
Глава 40. Проектирование признаков ... 437
Глава 41. Заглянем глубже: наивная байесовская классификация ... 445
Глава 42. Заглянем глубже: линейная регрессия ... 455
Глава 43. Заглянем глубже: метод опорных векторов ... 472
Глава 44. Заглянем глубже: деревья решений и случайные леса ... 489
Глава 45. Заглянем глубже: метод главных компонент ... 501
Глава 46. Заглянем глубже: обучение на базе многообразий ... 515
Глава 47. Заглянем глубже: кластеризация методом k средних ... 534
Глава 48. Заглянем глубже: смеси гауссовых распределений ... 549
Глава 49. Заглянем глубже: ядерная оценка плотности распределения ... 565
Глава 50. Прикладная задача: конвейер распознавания лиц ... 578

Об авторе ... 589
Иллюстрация на обложке ... 590

Оглавление

Отзывы ко второму изданию 18

Предисловие 19
Что такое наука о данных 19
Для кого предназначена эта книга 20
Почему Python 21
Общая структура книги 21
Вопросы установки 22
Условные обозначения 23
Использование примеров кода 24
Полноцветные иллюстрации 24
От издательства 24

ЧАСТЬ I. JUPYTER: ЗА ПРЕДЕЛАМИ ОБЫЧНОГО PYTHON

Глава 1. Знакомство с IPython и Jupyter 27
Запуск командной оболочки IPython 27
Запуск Jupyter Notebook 28
Справка и документация в IPython 28
Доступ к документации с помощью символа ? 29
Доступ к исходному коду с помощью символов ?? 31
Исследование содержимого модулей с помощью функции автодополнения 32
Горячие клавиши в командной оболочке IPython 34
Навигационные горячие клавиши 35
Горячие клавиши ввода текста 35
Горячие клавиши для истории команд 36
Прочие горячие клавиши 37

Глава 2. Расширенные интерактивные возможности 38
Магические команды IPython 38
Выполнение внешнего кода: %run 38
Измерение продолжительности выполнения кода: %timeit 39
Справка по «магическим» функциям: ?, %magic и %lsmagic 40
История ввода и вывода 40
Объекты In и Out оболочки IPython 40
Быстрый доступ к предыдущим выводам с помощью знака подчеркивания 42
Подавление вывода 42
Соответствующие «магические» команды 43
IPython и использование системного командного процессора 43
Краткое введение в использование командного процессора 44
Инструкции командного процессора в оболочке IPython 45
Передача значений в командный процессор и из него 45
«Магические» команды для командного процессора 46

Глава 3. Отладка и профилирование 48
Ошибки и отладка 48
Управление исключениями: %xmode....48
Отладка: что делать, если информации в трассировке недостаточно....50
Профилирование и хронометраж выполнения кода....53
Хронометраж выполнения фрагментов кода: %timeit и %time 53
Профилирование сценариев целиком: %prun....55
Пошаговое профилирование с помощью %lprun 56
Профилирование потребления памяти: %memit и %mprun....57
Дополнительные источники информации об оболочке IPython....59
Веб-ресурсы 59
Книги....59

ЧАСТЬ II. ВВЕДЕНИЕ В NUMPY

Глава 4. Типы данных в Python 64
Целое число в Python — больше, чем просто целое число....65
Список в Python — больше, чем просто список....66
Массивы фиксированного типа в Python 68
Создание массивов из списков 68
Создание массивов с нуля 69
Стандартные типы данных NumPy....70

Глава 5. Введение в массивы NumPy....72
Атрибуты массивов NumPy 72
Индексация массива: доступ к отдельным элементам 73
Срезы массивов: доступ к подмассивам 74
Одномерные подмассивы....75
Многомерные подмассивы 75
Подмассивы как представления....76
Создание копий массивов 77
Изменение формы массивов 77
Слияние и разбиение массивов....78
Слияние массивов 78
Разбиение массивов....80

Глава 6. Вычисления с массивами NumPy: универсальные функции 81
Медлительность циклов....81
Введение в универсальные функции....83
Обзор универсальных функций в библиотеке NumPy 84
Арифметические операции над массивами....84
Абсолютное значение....85
Тригонометрические функции 86
Показательные функции и логарифмы....86
Специализированные универсальные функции....87
Продвинутые возможности универсальных функций 88
Сохранение результатов в массиве....88

Сводные показатели ... 89
Векторные произведения ... 90
Универсальные функции: дополнительная информация ... 90

Глава 7. Агрегирование: минимум, максимум и все, что посередине ... 91
Суммирование значений в массиве ... 91
Минимум и максимум ... 92
Многомерные сводные показатели ... 93
Другие функции агрегирования ... 93
Пример: чему равен средний рост президентов США ... 94

Глава 8. Операции над массивами. Транслирование ... 97
Введение в транслирование ... 97
Правила транслирования ... 99
Транслирование. Пример 1 ... 99
Транслирование. Пример 2 ... 100
Транслирование. Пример 3 ... 101
Транслирование на практике ... 102
Центрирование массива ... 102
Построение графика двумерной функции ... 103

Глава 9. Сравнения, маски и булева логика ... 104
Пример: подсчет количества дождливых дней ... 104
Операторы сравнения как универсальные функции ... 106
Работа с булевыми массивами ... 107
Подсчет количества элементов ... 108
Булевы операторы ... 109
Булевы массивы как маски ... 110
Ключевые слова and/or и операторы &/| ... 111

Глава 10. «Прихотливая» индексация ... 114
Возможности «прихотливой» индексации ... 114
Комбинированная индексация ... 116
Пример: выборка случайных точек ... 116
Изменение значений с помощью «прихотливой» индексации ... 118
Пример: разбиение данных на интервалы ... 120

Глава 11. Сортировка массивов ... 123
Быстрая сортировка в библиотеке NumPy: функции np.sort и np.argsort ... 124
Сортировка по строкам и столбцам ... 124
Частичная сортировка: секционирование ... 125
Пример: k ближайших соседей ... 126

Глава 12. Структурированные данные: структурированные массивы NumPy ... 130
Создание структурированных массивов ... 132
Более продвинутые типы данных ... 133
Массивы записей: структурированные массивы с дополнительными возможностями ... 133
Вперед, к Pandas ... 134

ЧАСТЬ III. МАНИПУЛЯЦИИ НАД ДАННЫМИ С ПОМОЩЬЮ ПАКЕТА PANDAS

Глава 13. Знакомство с объектами библиотеки Pandas ... 138
Объект Series ... 138
Объект Series как обобщенный массив NumPy ... 139
Объект Series как специализированный словарь ... 140
Создание объектов Series ... 141
Объект DataFrame ... 142
DataFrame как обобщенный массив NumPy ... 142
Объект DataFrame как специализированный словарь ... 143
Создание объектов DataFrame ... 144
Объект Index ... 145
Объект Index как неизменяемый массив ... 146
Index как упорядоченное множество ... 146

Глава 14. Индексация и выборка данных ... 147
Выборка данных из объекта Series ... 147
Объект Series как словарь ... 147
Объект Series как одномерный массив ... 148
Индексаторы: loc и iloc ... 149
Выборка данных из объекта DataFrame ... 150
Объект DataFrame как словарь ... 150
Объект DataFrame как двумерный массив ... 152
Дополнительный синтаксис для индексации ... 154

Глава 15. Операции над данными в библиотеке Pandas ... 155
Универсальные функции: сохранение индекса ... 155
Универсальные функции: согласование индексов ... 156
Согласование индексов в объектах Series ... 156
Согласование индексов в объектах DataFrame ... 158
Универсальные функции: операции между объектами DataFrame и Series ... 159

Глава 16. Обработка отсутствующих данных ... 161
Компромиссы при обозначении отсутствующих данных ... 161
Отсутствующие данные в Pandas ... 162
None как значение-индикатор ... 163
NaN: отсутствующие числовые данные ... 164
Значения NaN и None в библиотеке Pandas ... 165
Типы данных с поддержкой пустых значений в Pandas ... 166
Операции над пустыми значениями ... 167
Выявление пустых значений ... 167
Удаление пустых значений ... 168
Заполнение пустых значений ... 169

Глава 17. Иерархическая индексация ... 171
Мультииндексированный объект Series ... 171
Плохой способ ... 172

Лучший способ: объект MultiIndex ... 173
Мультииндекс как дополнительное измерение ... 174
Методы создания объектов MultiIndex ... 175
Явные конструкторы MultiIndex ... 176
Названия уровней мультииндексов ... 177
Мультииндекс для столбцов ... 177
Индексация и срезы по мультииндексу ... 178
Мультииндексация объектов Series ... 178
Мультииндексация объектов DataFrame ... 180
Перегруппировка мультииндексов ... 181
Отсортированные и неотсортированные индексы ... 181
Выполнение операций stack и unstack над индексами ... 183
Создание и перестройка индексов ... 183

Глава 18. Объединение наборов данных: конкатенация и добавление в конец ... 185
Напоминание: конкатенация массивов NumPy ... 186
Простая конкатенация с помощью метода pd.concat ... 187
Дублирование индексов ... 188
Конкатенация с использованием соединений ... 189
Метод append() ... 190

Глава 19. Объединение наборов данных: слияние и соединение ... 191
Реляционная алгебра ... 192
Виды соединений ... 192
Соединения «один-к-одному» ... 192
Соединения «многие-к-одному» ... 193
Соединения «многие-ко-многим» ... 194
Задание ключа слияния ... 194
Именованный аргумент on ... 195
Именованные аргументы left_on и right_on ... 195
Именованные аргументы left_index и right_index ... 196
Применение операций над множествами для соединений ... 197
Пересекающиеся имена столбцов: именованный аргумент suffixes ... 199
Пример: данные по штатам США ... 200

Глава 20. Агрегирование и группировка ... 205
Данные о планетах ... 206
Простое агрегирование в библиотеке Pandas ... 206
groupby: разбиение, применение, объединение ... 208
Разбиение, применение и объединение ... 208
Объект GroupBy ... 211
Агрегирование, фильтрация, преобразование, применение ... 213
Задание ключа разбиения ... 215
Пример группировки ... 217

Глава 21. Сводные таблицы ... 218
Примеры для изучения приемов работы со сводными таблицами ... 218
Сводные таблицы «вручную» ... 219

Синтаксис сводных таблиц 220
Многоуровневые сводные таблицы 220
Дополнительные параметры сводных таблиц 221
Пример: данные о рождаемости 222

Глава 22. Векторизованные операции над строками 227
Знакомство со строковыми операциями в библиотеке Pandas 227
Таблица строковых методов в библиотеке Pandas 228
Методы, аналогичные строковым методам языка Python 228
Методы, использующие регулярные выражения 230
Прочие методы 231
Пример: база данных рецептов 233
Простая рекомендательная система для рецептов 235
Дальнейшая работа с рецептами 236

Глава 23. Работа с временными рядами 237
Дата и время в языке Python 238
Представление даты и времени в Python: пакеты datetime и dateutil 238
Типизированные массивы значений времени: тип datetime64 библиотеки NumPy 239
Даты и время в библиотеке Pandas: лучшее из обоих миров 241
Временные ряды библиотеки Pandas: индексация по времени 241
Структуры данных для временных рядов библиотеки Pandas 242
Регулярные последовательности: функция pd.date_range() 243
Периодичность и смещение дат 244
Передискретизация, временные сдвиги и окна 246
Передискретизация и изменение периодичности интервалов 248
Временные сдвиги 250
Скользящие окна 251
Пример: визуализация количества велосипедов в Сиэтле 252
Визуализация данных 253
Углубленное изучение данных 256

Глава 24. Увеличение производительности библиотеки Pandas: eval() и query() 259
Основания для использования функций query() и eval(): составные выражения 259
Использование функции pandas.eval() для эффективных операций 260
Использование метода DataFrame.eval() для выполнения операций по столбцам 262
Присваивание в методе DataFrame.eval() 263
Локальные переменные в методе DataFrame.eval() 264
Метод DataFrame.query() 264
Производительность: когда следует использовать эти функции 265
Дополнительные источники информации 266

ЧАСТЬ IV. ВИЗУАЛИЗАЦИЯ С ПОМОЩЬЮ БИБЛИОТЕКИ MATPLOTLIB

Глава 25. Общие советы по библиотеке Matplotlib 269
Импортирование matplotlib 269
Настройка стилей 269
Использовать или не использовать show()? Как отображать графики 270
Построение графиков в сценариях 270
Построение графиков из командной оболочки IPython 271
Построение графиков из блокнота Jupyter 271
Сохранение изображений в файлы 272
Два интерфейса по цене одного 273

Глава 26. Простые линейные графики 276
Настройка графика: цвета и стили линий 279
Настройка графика: пределы осей координат 281
Метки на графиках 284
Нюансы использования Matplotlib 285

Глава 27. Простые диаграммы рассеяния 287
Построение диаграмм рассеяния с помощью plt.plot 287
Построение диаграмм рассеяния с помощью plt.scatter 290
plot и scatter: примечание относительно производительности 292
Визуализация погрешностей 293
Простые планки погрешностей 293
Непрерывные погрешности 295

Глава 28. Графики плотности и контурные графики 297
Визуализация трехмерной функции 297
Гистограммы, разбиения по интервалам и плотность 301
Двумерные гистограммы и разбиение по интервалам 304
Функция plt.hist2d: двумерная гистограмма 304
Функция plt.hexbin: гексагональное разбиение по интервалам 305
Ядерная оценка плотности распределения 305

Глава 29. Настройка легенд на графиках 308
Выбор элементов для легенды 310
Задание легенды для точек разного размера 312
Отображение нескольких легенд 313

Глава 30. Настройка цветовых шкал 315
Настройка цветовой шкалы 315
Выбор карты цветов 317
Ограничение и расширение карты цветов 319
Дискретные цветовые шкалы 320
Пример: рукописные цифры 321

Глава 31. Множественные субграфики ... 324
plt.axes: создание субграфиков вручную ... 324
plt.subplot: простые сетки субграфиков ... 326
plt.subplots: создание всей сетки за один раз ... 328
plt.GridSpec: более сложные конфигурации ... 329

Глава 32. Текст и поясняющие надписи ... 332
Преобразования и координаты текста ... 334
Стрелки и поясняющие надписи ... 336

Глава 33. Настройка делений на осях координат ... 340
Основные и промежуточные деления осей координат ... 340
Сокрытие делений и/или меток ... 342
Уменьшение или увеличение количества делений ... 344
Экзотические форматы делений ... 345
Краткая сводка локаторов и форматеров ... 348

Глава 34. Настройка Matplotlib: конфигурации и таблицы стилей ... 349
Настройка графиков вручную ... 349
Изменение значений по умолчанию: rcParams ... 351
Таблицы стилей ... 353
Стиль по умолчанию default ... 354
Стиль FiveThirtyEight ... 354
Стиль ggplot ... 355
Стиль «байесовские методы для хакеров» ... 355
Стиль с темным фоном ... 356
Оттенки серого ... 356
Стиль Seaborn ... 357

Глава 35. Построение трехмерных графиков в библиотеке Matplotlib ... 358
Трехмерные точки и линии ... 359
Трехмерные контурные графики ... 360
Каркасы и поверхностные графики ... 362
Триангуляция поверхностей ... 364
Пример: визуализация ленты Мёбиуса ... 365

Глава 36. Визуализация с помощью библиотеки Seaborn ... 368
Анализируем графики Seaborn ... 369
Гистограммы, KDE и плотности ... 369
Графики пар ... 371
Фасетные гистограммы ... 372
Графики факторов ... 374
Совместные распределения ... 375
Столбиковые диаграммы ... 376
Пример: время прохождения марафона ... 377
Дополнительные источники информации ... 385
Другие графические библиотеки для Python ... 386

ЧАСТЬ V. МАШИННОЕ ОБУЧЕНИЕ

Глава 37. Что такое машинное обучение 389
Категории машинного обучения 390
Качественные примеры прикладных задач машинного обучения 390
Классификация: предсказание дискретных меток 391
Регрессия: предсказание непрерывных меток 393
Кластеризация: определение меток для немаркированных данных 396
Понижение размерности: определение структуры немаркированных данных 398
Резюме 400

Глава 38. Знакомство с библиотекой Scikit-Learn 401
Представление данных в Scikit-Learn 401
Матрица признаков 402
Целевой массив 402
API статистического оценивания в Scikit-Learn 404
Основы API статистического оценивания 405
Пример обучения с учителем: простая линейная регрессия 406
Пример обучения с учителем: классификация набора данных Iris 409
Пример обучения без учителя: понижение размерности набора данных Iris 410
Обучение без учителя: кластеризация набора данных Iris 412
Прикладная задача: анализ рукописных цифр 413
Загрузка и визуализация цифр 413
Обучение без учителя: понижение размерности 415
Классификация цифр 416
Резюме 418

Глава 39. Гиперпараметры и проверка модели 419
Соображения относительно проверки модели 419
Плохой способ проверки модели 420
Хороший способ проверки модели: отложенные данные 420
Перекрестная проверка модели 421
Выбор оптимальной модели 424
Компромисс между систематической ошибкой и дисперсией 424
Кривые проверки в библиотеке Scikit-Learn 427
Кривые обучения 430
Проверка на практике: поиск по сетке 435
Резюме 436

Глава 40. Проектирование признаков 437
Категориальные признаки 437
Текстовые признаки 439
Признаки для изображений 440
Производные признаки 440
Подстановка отсутствующих данных 443
Конвейеры признаков 444

Глава 41. Заглянем глубже: наивная байесовская классификация ... 445
Байесовская классификация ... 445
Гауссов наивный байесовский классификатор ... 446
Полиномиальный наивный байесовский классификатор ... 449
Пример: классификация текста ... 450
Когда имеет смысл использовать наивный байесовский классификатор ... 453

Глава 42. Заглянем глубже: линейная регрессия ... 455
Простая линейная регрессия ... 455
Регрессия по комбинации базисных функций ... 458
Полиномиальные базисные функции ... 458
Гауссовы базисные функции ... 460
Регуляризация ... 462
Гребневая регрессия (L_2-регуляризация) ... 464
Лассо-регрессия (L_1-регуляризация) ... 465
Пример: предсказание велосипедного трафика ... 466

Глава 43. Заглянем глубже: метод опорных векторов ... 472
Основания для использования метода опорных векторов ... 472
Метод опорных векторов: максимизация отступа ... 474
Аппроксимация методом опорных векторов ... 475
За границами линейности: SVM-ядро ... 478
Настройка SVM: размытие отступов ... 481
Пример: распознавание лиц ... 483
Резюме ... 487

Глава 44. Заглянем глубже: деревья решений и случайные леса ... 489
Движущая сила случайных лесов: деревья принятия решений ... 489
Создание дерева принятия решений ... 490
Деревья принятия решений и переобучение ... 493
Ансамбли моделей: случайные леса ... 494
Регрессия с помощью случайных лесов ... 496
Пример: использование случайного леса для классификации цифр ... 497
Резюме ... 500

Глава 45. Заглянем глубже: метод главных компонент ... 501
Знакомство с методом главных компонент ... 501
PCA как метод понижения размерности ... 504
Использование метода PCA для визуализации: рукописные цифры ... 505
Что означают компоненты? ... 507
Выбор количества компонент ... 508
Использование метода PCA для фильтрации шума ... 509
Пример: метод Eigenfaces ... 511
Резюме ... 514

Глава 46. Заглянем глубже: обучение на базе многообразий ... 515
Обучение на базе многообразий: «HELLO» ... 516

Многомерное масштабирование (MDS) 517
MDS как обучение на базе многообразий 520
Нелинейные вложения: там, где MDS не работает 521
Нелинейные многообразия: локально линейное вложение 523
Некоторые соображения относительно методов обучения на базе многообразий 525
Пример: использование Isomap для распознавания лиц 526
Пример: визуализация структуры цифр 530

Глава 47. Заглянем глубже: кластеризация методом k средних 534
Знакомство с методом k средних 534
Максимизация математического ожидания 536
Примеры 542
Пример 1: применение метода k средних для распознавания рукописных цифр 542
Пример 2: использование метода k средних для сжатия цветов 545

Глава 48. Заглянем глубже: смеси гауссовых распределений 549
Причины появления GMM: недостатки метода k средних 549
Обобщение EM-модели: смеси гауссовых распределений 553
Выбор типа ковариации 557
GMM как метод оценки плотности распределения 557
Пример: использование метода GMM для генерации новых данных 561

Глава 49. Заглянем глубже: ядерная оценка плотности распределения 565
Обоснование метода KDE: гистограммы 565
Ядерная оценка плотности распределения на практике 570
Выбор ширины ядра путем перекрестной проверки 571
Пример: не столь наивный байес 572
Внутреннее устройство пользовательской модели 574
Использование пользовательской модели 576

Глава 50. Прикладная задача: конвейер распознавания лиц 578
Признаки HOG 579
Метод HOG в действии: простой детектор лиц 580
1. Получаем набор положительных обучающих образцов 580
2. Получаем набор отрицательных обучающих образцов 581
3. Объединяем наборы и выделяем HOG-признаки 582
4. Обучаем метод опорных векторов 583
5. Выполняем поиск лиц в новом изображении 583
Предостережения и дальнейшие усовершенствования 585
Дополнительные источники информации по машинному обучению 587

Об авторе 589
Иллюстрация на обложке 590

Отзывы ко второму изданию

К настоящему времени опубликовано большое количество книг, посвященных науке о данных, но, как мне кажется, Джейку ВандерПласу удалось написать исключительную книгу. Он взял очень сложную и обширную тему, разбил ее на части и описал ее таким простым и понятным языком, сопроводив множеством примеров, что вы без труда освоите ее.

Селеста Стингер (Celeste Stinger), инженер по надежности сайта

Опыт Джейка ВандерПласа и его страсть к обмену знаниями несомненны. В этом обновленном издании вы найдете ясные и простые примеры, которые помогут вам настроить и использовать основные инструменты обработки данных и машинного обучения. Если вы готовы погрузиться в обработку данных и использовать инструменты Python для извлечения реальной информации из ваших данных, то эта книга для вас!

Энн Боннер (Anne Bonner), основатель и генеральный директор Content Simplicity

В течение многих лет я неизменно рекомендовал книгу «Python для сложных задач» студентам, изучающим курс науки о данных. Второе издание этой великолепной книги получилось еще лучше благодаря включению привлекательных блокнотов Jupyter, которые могут заняться обработкой ваших данных, пока вы читаете.

Ной Гифт (Noah Gift), исполнительный директор и основатель Pragmatic AI Labs

Это обновленное издание является отличным введением в библиотеки, делающие Python одним из лучших языков для обработки данных и научных вычислений. Книга написана доступным языком и наполнена отличными примерами.

Аллен Дауни (Allen Downey), автор книг Think Python[1] и Think Bayes[2]

«Python для сложных задач» — отличное руководство для изучающих приемы обработки данных на Python. Благодаря практическим примерам, написанным в доступной форме, читатель сможет научиться эффективно хранить наборы данных, манипулировать ими и извлекать полезную информацию.

Уильям Джамир Сильва (William Jamir Silva),
старший инженер-программист, Adjust GmbH

Джейк ВандерПлас имеет богатый опыт объяснения основных концепций и инструментов Python тем, кто изучает науку о данных. И в этом втором издании «Python для сложных задач» он снова подтвердил свои умения. В этой книге вы найдете обзор всех инструментов, которые могут пригодиться на начальном этапе, а также доступные объяснения, почему то или иное реализуется именно так, а не иначе.

Джеки Казил (Jackie Kazil), создатель библиотеки Mesa
и обладатель звания Data Science Leader

---

[1] *Дауни А.* Основы Python. Научитесь мыслить как программист.

[2] *Дауни А.* Байесовские модели. Байесовская статистика на языке Python.

Предисловие

Что такое наука о данных

Эта книга посвящена исследованию данных с помощью языка программирования Python. Сразу же возникает вопрос: что же такое *наука о данных* (data science)? Ответ на него дать непросто — настолько этот термин многозначен. Долгое время активные критики отказывали термину «наука о данных» в праве на существование либо по причине его избыточности (в конце концов, какая наука не имеет дела с данными?), либо расценивая этот термин как «модное словечко» для придания красоты резюме и привлечения внимания агентов по найму кадров.

На мой взгляд, в подобных высказываниях критики упускали нечто очень важное. Название «наука о данных», несмотря на поднятую вокруг него шумиху, пожалуй, лучше всего подходит для обозначения междисциплинарного набора навыков, становящихся все более востребованными в промышленности и науке. Междисциплинарность — ключ к ее пониманию. Лучшее из возможных определений науки о данных приведено в диаграмме Венна в науке о данных, впервые опубликованной Дрю Конвеем в его блоге в сентябре 2010 года (рис. 0.1).

Хотя некоторые названия немного ироничны, эта диаграмма ясно передает суть того, что, как мне кажется, понимается под наукой о данных: она является междисциплинарным предметом. Наука о данных охватывает три отдельные, но пересекающиеся сферы:

- навыки *специалиста по математической статистике*, умеющего моделировать наборы данных и извлекать из них основное;
- навыки *специалиста в области информатики*, умеющего проектировать и использовать алгоритмы для эффективного хранения, обработки и визуализации этих данных;
- *экспертные знания предметной области*, полученные в ходе традиционного изучения предмета, — умение формулировать правильные вопросы и рассматривать ответы на них в соответствующем контексте.

С учетом этого я рекомендовал бы рассматривать науку о данных не как новую область знаний, которую нужно изучить, а как новый набор навыков, который вы можете использовать в рамках хорошо знакомой вам предметной области. Извещаете ли вы о результатах выборов, прогнозируете ли прибыльность ценных бумаг,

занимаетесь ли оптимизацией контекстной рекламы в Интернете или распознаванием микроорганизмов на фотографиях, сделанных с помощью микроскопа, ищете ли новые классы астрономических объектов или же работаете с данными в любой другой сфере, цель этой книги — научить задавать новые вопросы о вашей предметной области и отвечать на них.

Рис. 0.1. Диаграмма Венна в науке о данных, опубликованная Дрю Конвееем (источник: https://oreil.ly/PkOOw)

Для кого предназначена эта книга

«Как именно следует изучать Python?» — один из наиболее часто задаваемых мне вопросов студентами Вашингтонского университета и на различных технологических конференциях и встречах. Задают его заинтересованные в технологиях студенты, разработчики или исследователи, часто уже со значительным опытом написания кода и использования вычислительного и цифрового инструментария. Большинству из них не нужен язык программирования Python в чистом виде, они хотели бы изучать его, чтобы применять в качестве инструмента для решения задач, требующих вычислений с обработкой больших объемов данных. Несмотря на большое разнообразие видеороликов, статей в блогах и учебных пособий, ориентированных на эту аудиторию, вы не найдете в Интернете единственного хорошего ответа на данный вопрос, именно поэтому я взялся за эту книгу.

Эта книга не планировалась как введение в язык Python или в программирование вообще. Я предполагаю, что читатель знаком с языком Python и знает, как опре-

делять функции, присваивать значения переменным, вызывать методы объектов, управлять потоком выполнения программы и решать другие простейшие задачи. Она должна помочь пользователям языка Python научиться применять инструменты исследования данных, имеющиеся в языке Python, — библиотеки, упоминающиеся в следующем разделе, и сопутствующие инструменты — для эффективного хранения и анализа данных и извлечения из них полезной информации.

Почему Python

За последние несколько десятилетий язык программирования Python превратился в первоклассный инструмент для научных вычислений, включая анализ и визуализацию больших наборов данных. Это может удивить давних поклонников Python: сам по себе этот язык не был создан в расчете на анализ данных или научные вычисления. Возможность применения языка программирования Python в науке о данных появилась в основном благодаря большой и активно развивающейся экосистеме пакетов, созданных сторонними разработчиками, в число которых входят:

- *NumPy* — библиотека для работы с массивами однородных данных;
- *Pandas* — библиотека для работы с разнородными и маркированными данными;
- *SciPy* — библиотека для общих научных вычислительных задач;
- *Matplotlib* — библиотека для создания визуализаций типографского качества;
- *IPython* — оболочка для интерактивного выполнения и распространения кода;
- *Scikit-Learn* — библиотека для машинного обучения.

И множество других инструментов, которые будут упомянуты в дальнейшем.

Если вы ищете руководство по самому языку программирования Python, то рекомендую обратить ваше внимание на проект *A Whirlwind Tour of Python* («Краткая экскурсия по языку программирования Python») (https://oreil.ly/jFtWj). Он знакомит с важнейшими возможностями языка Python и рассчитан на исследователей данных, уже знакомых с одним или несколькими языками программирования.

Общая структура книги

Каждая часть книги посвящена конкретному пакету или инструменту, составляющему существенную часть инструментария Python для исследования данных, и разбита на более короткие независимые главы, обсуждающие отдельные понятия.

- *Jupyter: за пределами обычного Python (часть I)* — знакомит с IPython и Jupyter. Эти пакеты образуют вычислительное окружение, в котором работают многие исследователи данных, использующие Python.

- *Введение в NumPy (часть II)* — представляет библиотеку NumPy, в состав которой входит объект `ndarray` для эффективного хранения и работы с плотными массивами данных в Python.
- *Манипуляции над данными с помощью пакета Pandas (часть III)* — представляет библиотеку Pandas, в состав которой входит объект `DataFrame` для эффективного хранения и работы с маркированными/табличными данными в Python.
- *Визуализация с помощью библиотеки Matplotlib (часть IV)* — представляет библиотеку Matplotlib, реализующую богатые возможности для разнообразной гибкой визуализации данных в Python.
- *Машинное обучение (часть V)* — представляет библиотеку Scikit-Learn, включающую эффективные реализации на Python многих важных и широко известных алгоритмов машинного обучения.

Мир PyData гораздо шире представленных пакетов, и он растет день ото дня. С учетом этого я использую каждую возможность в книге, чтобы упомянуть другие интересные работы, проекты и пакеты, расширяющие пределы возможностей языка Python. Тем не менее пакеты, перечисленные выше, являются основой для применения языка программирования Python в исследовании данных. Я полагаю, что они сохранят свое значение и при росте окружающей их экосистемы.

Вопросы установки

Установка Python и набора библиотек поддержки научных вычислений не представляет сложности. В данном разделе будут рассмотрены особенности, которые следует принимать во внимание при настройке.

Существует множество вариантов установки Python, но я предложил бы воспользоваться дистрибутивом Anaconda, одинаково работающим в операционных системах Windows, Linux и macOS. Дистрибутив Anaconda существует в двух вариантах.

- *Miniconda* (https://oreil.ly/dH7wJ) содержит сам интерпретатор Python, а также утилиту командной строки *conda* — кросс-платформенную систему управления пакетами, ориентированную на работу с пакетами Python и аналогичную по духу утилитам apt и yum, хорошо знакомым пользователям операционной системы Linux.
- *Anaconda* (https://oreil.ly/ndxjm) включает интерпретатор Python и утилиту conda, а также набор предустановленных пакетов для научных вычислений. Приготовьтесь к тому, что для установки этого дистрибутива потребуется несколько гигабайтов дискового пространства.

Все пакеты, входящие в состав Anaconda, можно также установить вручную после установки Miniconda, именно поэтому я рекомендую начать с Miniconda.

Для начала скачайте и установите пакет Miniconda (не забудьте выбрать версию с языком Python 3), после чего установите базовые пакеты, используемые в данной книге:

```
[~]$ conda install numpy pandas scikit-learn matplotlib seaborn jupyter
```

На протяжении всей книги мы будем применять и другие, более специализированные утилиты из научной экосистемы Python, установка которых сводится к выполнению команды `conda install` *`название_пакета`*. Если вдруг вам доведется столкнуться с пакетами, недоступными в conda по умолчанию, то обязательно загляните в *conda-forge* (https://oreil.ly/CCvwQ) — обширный репозиторий пакетов conda.

За дополнительной информацией об утилите conda, включая информацию о создании и использовании сред разработки conda (которые я *настоятельно* рекомендую), обратитесь к онлайн-документации утилиты conda (https://oreil.ly/MkqPw).

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины.

`Моноширинный шрифт`

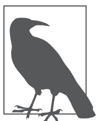
Используется для листингов программ, а также внутри текста для выделения элементов программ, таких как имена переменных, функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Полужирный моноширинный шрифт`

Выделяет команды или другой текст, который пользователь должен ввести самостоятельно.

`Курсивный моноширинный шрифт`

Выделяет текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.



Так обозначаются примечания общего характера.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. п.) доступны для скачивания по адресу https://github.com/jakevdp/PythonDataScienceHandbook.

В общем случае все примеры кода из книги вы можете использовать в своих программах и документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Полноцветные иллюстрации

Поскольку эта книга напечатана в черно-белом исполнении, мы подготовили онлайн-приложение с полноцветными и полноразмерными изображениями, которое вы найдете по адресу https://oreil.ly/PDSH_GitHub.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ I

Jupyter: за пределами обычного Python

Меня часто спрашивают, какой из множества вариантов среды разработки для Python я использую в своей работе. Мой ответ иногда удивляет спрашивающих: моя излюбленная среда — IPython (http://ipython.org/) плюс текстовый редактор (в моем случае редактор Emacs или VSCode в зависимости от настроения). Jupyter начинался как оболочка IPython (сокращение от «интерактивный Python»), созданная в 2001 году Фернандо Пересом и задумывавшаяся как продвинутый интерпретатор Python, и с тех пор вырос в проект, призванный обеспечить, по словам Переса, «инструменты для всего жизненного цикла исследовательских расчетов». Если язык Python — механизм решения нашей задачи в области науки о данных, то Jupyter можно рассматривать как интерактивную панель управления.

Jupyter как полезный интерактивный интерфейс для языка Python поддерживает несколько удобных синтаксических дополнений к нему. Самый известный, пожалуй, интерфейс, предоставляемый проектом Jupyter, — это Jupyter Notebook, веб-среда для разработки, сотрудничества, совместного использования и даже публикации результатов. Не стоит далеко ходить за примером, показывающим удобства формата блокнотов Jupyter, им служит страница, которую вы сейчас читаете: вся рукопись данной книги написана как набор блокнотов Jupyter.

Эта часть книги начнется знакомством с некоторыми особенностями Jupyter и IPython, которые могут пригодиться при исследовании данных. Основное свое внимание мы сосредоточим на синтаксических возможностях, выходящих за пределы стандартных возможностей языка Python. Немного углубимся в *«магические» команды*, позволяющие ускорить выполнение стандартных задач при создании и использовании кода, предназначенного для исследования данных. И наконец, затронем возможности Jupyter Notebook, помогающие понять смысл данных и обмениваться результатами.

ГЛАВА 1

Знакомство с IPython и Jupyter

https://t.me/it_boooks/2

Разрабатывая код на Python для исследования данных, я обычно переключаюсь между тремя режимами работы: использую оболочку IPython для тестирования коротких последовательностей команд, Jupyter Notebook — для более продолжительного интерактивного анализа и обмена результатами исследований с другими, а также интерактивные среды разработки (Interactive Development Environments, IDE), такие как Emacs или VSCode, для создания многократно используемых пакетов Python. В этой главе основное внимание уделяется первым двум режимам: оболочке IPython и Jupyter Notebook. Интерактивная среда разработки (IDE) — очень важный третий инструмент в арсенале любого специалиста по обработке и анализу данных, но здесь мы не будем обращаться к нему напрямую.

Запуск командной оболочки IPython

Данная глава, как и бóльшая часть книги, не предназначена для пассивного чтения. Я рекомендую вам экспериментировать с описываемыми инструментами и синтаксисом: формируемая при этом мышечная память принесет намного больше пользы, чем простое чтение. Начнем с запуска интерпретатора оболочки IPython путем ввода команды `ipython` в командной строке. Если вы установили один из таких дистрибутивов, как Anaconda или EPD, то у вас, возможно, уже есть средство запуска для вашей операционной системы.

После этого вы должны увидеть приглашение к вводу:

```
Python 3.9.2 (v3.9.2:1a79785e3e, Feb 19 2021, 09:06:10)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.21.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Теперь вы можете активно следовать за примерами в книге.

Запуск Jupyter Notebook

Jupyter Notebook — браузерный графический интерфейс для командной оболочки Ipython, предлагающий богатый набор возможностей динамической визуализации. Помимо выполнения операторов Python/IPython, блокноты позволяют пользователям вставлять форматированный текст, статические и динамические диаграммы, математические уравнения, виджеты JavaScript и многое другое. Более того, эти документы можно сохранять, благодаря чему другие смогут открывать и выполнять их в своих системах.

Хотя просмотр и редактирование блокнотов Jupyter осуществляется в окне браузера, они должны подключаться к запущенному процессу Python для выполнения кода. Для запуска этого процесса (называемого «ядром») выполните следующую команду в командной строке вашей операционной системы:

```
$ jupyter lab
```

Эта команда запустит локальный веб-сервер, доступный браузеру, и сразу же начнет журналировать выполняемые действия. Журнал будет выглядеть следующим образом:

```
$ jupyter lab
[ServerApp] Serving notebooks from local directory: /Users/jakevdp/ \
PythonDataScienceHandbook
[ServerApp] Jupyter Server 1.4.1 is running at:
[ServerApp] http://localhost:8888/lab?token=dd852649
[ServerApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
```

Эта команда должна автоматически запустить браузер по умолчанию и открыть в нем указанный в выводе локальный URL; точный адрес зависит от вашей системы. Если браузер не запускается автоматически, то запустите его вручную и перейдите по указанному в выводе адресу (в примере это http://localhost:8888/lab).

Справка и документация в IPython

Если вы не читали другие разделы в данной главе, прочитайте хотя бы этот. Обсуждаемые здесь утилиты (из IPython) внесли наибольший вклад в мой ежедневный процесс разработки.

Когда человека с техническим складом ума просят помочь другу, родственнику или коллеге решить проблему с компьютером, чаще всего речь идет об умении быстро найти неизвестное решение. В науке о данных все точно так же: веб-ресурсы с поддержкой поиска, такие как онлайн-документация, дискуссии в почтовых рассылках и ответы на сайте Stack Overflow, содержат массу информации, даже

если речь идет о теме, информацию по которой вы уже искали. Уметь эффективно исследовать данные означает скорее не запоминание утилит или команд, которые нужно использовать в каждой из возможных ситуаций, а знание того, как эффективно искать неизвестную пока информацию: посредством поиска в Интернете или с помощью других средств.

Одна из самых полезных возможностей IPython/Jupyter заключается в сокращении разрыва между пользователями и типом документации и поиска, что должно помочь им эффективнее выполнять свою работу. Хотя поиск в Интернете все еще играет важную роль в ответе на сложные вопросы, большое количество информации можно найти, используя саму оболочку IPython. Вот несколько примеров вопросов, на которые IPython может помочь ответить буквально с помощью нескольких нажатий клавиш.

- Как вызвать эту функцию? Какие параметры она имеет?
- Как выглядит исходный код этого объекта Python?
- Что имеется в импортированном мной пакете?
- Какие атрибуты или методы есть у этого объекта?

Далее мы обсудим инструменты IPython для быстрого доступа к этой информации, а именно символ `?` для просмотра документации, символы `??` для просмотра исходного кода и клавишу Tab для автодополнения.

Доступ к документации с помощью символа ?

Язык программирования Python и его экосистема для исследования данных ориентированы на потребности клиента, и в значительной степени это проявляется в доступе к документации. Каждый объект Python содержит ссылку на строку, именуемую *docstring* (сокращение от *documentation string* — «строка документации»), которая в большинстве случаев будет содержать краткое описание объекта и способ его использования. В языке Python имеется встроенная функция `help`, позволяющая обращаться к этой информации и выводить результат. Например, вот как можно посмотреть документацию по встроенной функции `len`:

```
In [1]: help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container[1].
```

В зависимости от интерпретатора информация будет отображена в виде встраиваемого текста или в отдельном всплывающем окне.

[1] Возвращает количество элементов в контейнере. — *Здесь и далее примеч. пер.*

Поскольку поиск справочной информации по объекту — очень распространенное действие, оболочка IPython предоставляет символ `?` для быстрого доступа к документации и другой соответствующей информации:

```
In [2]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container[1].
Type:      builtin_function_or_method
```

Данная нотация подходит практически для чего угодно, включая методы объектов:

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Signature: L.insert(index, object, /)
Docstring: Insert object before index[2].
Type:      builtin_function_or_method
```

или даже сами объекты с описанием их типов:

```
In [5]: L?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified[3].
```

Эта возможность поддерживается даже для созданных пользователем функций и других объектов! В следующем фрагменте кода мы опишем маленькую функцию с docstring:

```
In [6]: def square(a):
  ....:     """Return the square of a."""[4]
  ....:     return a ** 2
  ....:
```

Обратите внимание: чтобы создать docstring с описанием нашей функции, мы просто вставили в начало определения строковый литерал. Поскольку docstring обычно занимает несколько строк, в соответствии с соглашениями мы использовали тройные кавычки — нотацию языка Python для многострочных docstring.

[1] Возвращает количество элементов в контейнере.

[2] Вставляет object перед index.

[3] При вызове без аргументов конструктор создает новый пустой список. Передаваемый аргумент должен быть итерируемым объектом.

[4] Возвращает квадрат числа a.

Теперь воспользуемся знаком ? для поиска этой строки docstring:

```
In [7]: square?
Signature: square(a)
Docstring: Return the square of a.
File:      <ipython-input-6>
Type:      function
```

Быстрый доступ к документации через строки docstring — одна из причин, почему желательно приучить себя добавлять подобную встроенную документацию в создаваемый код!

Доступ к исходному коду с помощью символов ??

Поскольку код на языке Python читается очень легко, простота доступа к исходному коду интересующего вас объекта может обеспечить более глубокое его понимание. Оболочка IPython предоставляет сокращенную форму получения исходного кода — двойной знак вопроса (??):

```
In [8]: square??
Signature: square(a)
Source:
def square(a):
    """Return the square of a."""
    return a ** 2
File:      <ipython-input-6>
Type:      function
```

Для подобных простых функций двойной знак вопроса позволяет быстро вникнуть в особенности внутренней реализации.

Немного поэкспериментировав, вы можете заметить, что иногда добавление ?? в конце не приводит к отображению исходного кода: обычно это объясняется тем, что рассматриваемый объект реализован не на языке Python, а на C или каком-либо другом компилируемом языке. В подобных случаях добавление ?? приводит к такому же результату, что и добавление ?. Вы столкнетесь с этим при исследовании многих встроенных объектов и типов Python, например упомянутой выше функции len:

```
In [9]: len??
Signature: len(obj, /)
Docstring: Return the number of items in a container¹.
Type:      builtin_function_or_method
```

Использование ? и/или ?? — простой способ для быстрого поиска информации о работе любой функции или модуля языка Python.

[1] Возвращает количество элементов в контейнере.

Исследование содержимого модулей с помощью функции автодополнения

Другой удобный интерфейс оболочки IPython — клавиша Tab и вызываемая ею функция автодополнения, позволяющая исследовать содержимое объектов, модулей и пространств имен. В следующих примерах мы будем применять обозначение `<TAB>` там, где необходимо нажать клавишу Tab.

Использование автодополнения для исследования содержимого объектов

Каждый объект в Python имеет множество различных атрибутов и методов. Помимо упоминавшейся выше функции `help`, в языке Python есть встроенная функция `dir`, возвращающая их список, но на практике интерфейс автодополнения по клавише Tab гораздо удобнее. Чтобы получить список всех доступных атрибутов объекта, необходимо набрать имя объекта, символ точки (`.`) и нажать клавишу Tab:

```
In [10]: L.<TAB>
            append() count    insert   reverse
            clear    extend   pop      sort
            copy     index    remove
```

Чтобы сократить список, можно набрать первый символ или несколько символов нужного имени и нажать клавишу Tab, после чего будут отображены соответствующие атрибуты и методы:

```
In [10]: L.c<TAB>
             clear() count()
             copy()

In [10]: L.co<TAB>
              copy()  count()
```

Если имеется только один вариант, нажатие клавиши Tab приведет к автодополнению строки. Например, эта последовательность символов будет немедленно заменена на `L.count`:

```
In [10]: L.cou<TAB>
```

В языке Python отсутствует четкое разграничение между открытыми/внешними и закрытыми/внутренними атрибутами, поэтому по соглашениям для обозначения закрытых методов их имена принято начинать с символа подчеркивания. По умолчанию закрытые, а также специальные методы исключаются из списка вариантов автодополнения, но их можно вывести, набрав знак подчеркивания:

```
In [10]: L._<TAB>
           __add__              __delattr__     __eq__
           __class__            __delitem__     __format__()
           __class_getitem__() __dir__()        __ge__            >
           __contains__         __doc__         __getattribute__
```

Для краткости я показал только несколько первых строк вывода. Большинство этих методов имеют специальное значение в языке Python, и их имена начинаются с двойного подчеркивания (на сленге называются dunder[1]-методами).

Автодополнение в инструкциях импортирования

Функцию автодополнения удобно использовать также в инструкциях импорта для импортирования объектов из пакетов. Воспользуемся этой возможностью для поиска всех элементов с именами, начинающимися с `co`, доступных для импорта из пакета `itertools`:

```
In [10]: from itertools import co<TAB>
         combinations()                  compress()
         combinations_with_replacement() count()
```

Точно так же можно использовать функцию автодополнения по Tab для просмотра пакетов, доступных в системе для импорта (у вас результат может отличаться от приведенного ниже в зависимости от того, какие сторонние сценарии и модули являются видимыми в данном сеансе Python):

```
In [10]: import <TAB>
            abc                 anyio
            activate_this       appdirs
            aifc                appnope        >
            antigravity         argon2

In [10]: import h<TAB>
            hashlib html
            heapq   http
            hmac
```

Помимо автодополнения: поиск с использованием шаблонного символа

Автодополнение по клавише Tab удобно, когда известны первые несколько символов в имени искомого объекта или атрибута. Однако эта функция малопригодна, когда нужно найти соответствие по символам, находящимся в середине или конце имени. На этот случай оболочка IPython предлагает возможность поиска соответствий с использованием шаблонного символа `*`.

[1] Игра слов: одновременно сокращение от double underscore — «двойное подчеркивание» и dunderhead — «тупица», «болван».

Например, вот как можно получить список всех объектов с именами, оканчивающимися на `Warning`:

```
In [10]: *Warning?
BytesWarning                  RuntimeWarning
DeprecationWarning            SyntaxWarning
FutureWarning                 UnicodeWarning
ImportWarning                 UserWarning
PendingDeprecationWarning     Warning
ResourceWarning
```

Обратите внимание, что символ `*` соответствует любой строке, включая пустую.

Аналогично предположим, что мы ищем строковый метод, содержащий где-то в имени слово `find`. Отыскать его можно так:

```
In [11]: str.*find*?
str.find
str.rfind
```

Я обнаружил, что подобный гибкий поиск с помощью шаблонных символов очень удобен для поиска нужной команды при знакомстве с новым пакетом или обращении после перерыва к уже знакомому.

Горячие клавиши в командной оболочке IPython

Вероятно, все, кто проводит время за компьютером, используют в своей работе горячие клавиши. Наиболее известные — Cmd+C и Cmd+V (или Ctrl+C и Ctrl+V), применяемые для копирования и вставки в различных программах и системах. Опытные пользователи выбирают популярные текстовые редакторы, такие как Emacs, Vim и другие, позволяющие выполнять множество операций посредством замысловатых сочетаний клавиш.

В командной оболочке IPython также имеются горячие клавиши для быстрой навигации при наборе команд. Хотя некоторые из них работают в блокноте для браузера, данный раздел в основном касается горячих клавиш именно в командной оболочке IPython.

Привыкнув к сочетаниям горячих клавиш, вы сможете использовать их для быстрого выполнения команд без изменения исходного положения рук на клавиатуре. Если вы пользователь Emacs или имеете опыт работы с Linux-подобными командными оболочками, некоторые сочетания горячих клавиш покажутся вам знакомыми. Мы сгруппируем их в несколько категорий: *навигационные горячие клавиши*, *горячие клавиши ввода текста*, *горячие клавиши для истории команд* и *прочие горячие клавиши*.

Навигационные горячие клавиши

Использовать стрелки «влево» (←) и «вправо» (→) для перемещения назад и вперед по строке вполне естественно, но есть и другие возможности, не требующие изменения исходного положения рук на клавиатуре (табл. 1.1).

Таблица 1.1. Навигационные горячие клавиши

Комбинация клавиш	Действие
Ctrl+A	Перемещает курсор в начало строки
Ctrl+E	Перемещает курсор в конец строки
Ctrl+B (или стрелка «влево»)	Перемещает курсор назад на один символ
Ctrl+F (или стрелка «вправо»)	Перемещает курсор вперед на один символ

Горячие клавиши ввода текста

Для удаления предыдущего символа привычно использовать клавишу Backspace, несмотря на то что требуется небольшая гимнастика для пальцев, чтобы до нее дотянуться. Эта клавиша удаляет только один символ за раз. В оболочке IPython имеется несколько сочетаний горячих клавиш для удаления различных частей набираемого текста. Наиболее полезные из них — команды для удаления сразу целых строк текста (табл. 1.2). Вы поймете, что привыкли к ним, когда поймаете себя на использовании сочетания Ctrl+B и Ctrl+D вместо Backspace для удаления предыдущего символа!

Таблица 1.2. Горячие клавиши для ввода текста

Комбинация клавиш	Действие
Backspace	Удаляет предыдущий символ в строке
Ctrl+D	Удаляет следующий символ в строке
Ctrl+K	Вырезает текст, начиная от курсора и до конца строки
Ctrl+U	Вырезает текст с начала строки до курсора
Ctrl+Y	Вставляет предварительно вырезанный текст
Ctrl+T	Меняет местами предыдущие два символа

Горячие клавиши для истории команд

Вероятно, наиболее важные из обсуждаемых здесь горячих клавиш в IPython — сочетания для навигации по истории команд. Данная история команд распространяется за пределы текущего сеанса оболочки IPython: полная история команд хранится в базе данных SQLite в каталоге с профилем IPython.

Простейший способ получить к ним доступ — с помощью стрелок «вверх» (↑) и «вниз» (↓) для пошагового перемещения по истории, но есть и другие варианты (табл. 1.3).

Таблица 1.3. Горячие клавиши для истории команд

Комбинация клавиш	Действие
Ctrl+P (или стрелка «вверх»)	Доступ к предыдущей команде в истории
Ctrl+N (или стрелка «вниз»)	Доступ к следующей команде в истории
Ctrl+R	Поиск в обратном направлении по истории команд

Особенно полезным может оказаться поиск в обратном направлении. Как вы помните, в предыдущем разделе мы определили функцию `square`. Выполним поиск в обратном направлении по нашей истории команд Python в новом окне оболочки IPython и найдем это описание снова. После нажатия Ctrl+R в терминале IPython вы должны увидеть следующее приглашение командной строки:

```
In [1]:
(reverse-i-search)`':
```

Если начать вводить символы в этом приглашении, IPython автоматически будет дополнять их, подставляя недавние команды, соответствующие этим символам, если такие существуют:

```
In [1]:
(reverse-i-search)`sqa': square??
```

Вы можете в любой момент добавить символы для уточнения поискового запроса или снова нажать Ctrl+R, чтобы отыскать следующую команду, соответствующую запросу. Если в процессе чтения предыдущего раздела вы выполняли все описанные там действия, то нажмите Ctrl+R еще два раза, и вы получите:

```
In [1]:
(reverse-i-search)`sqa': def square(a):
    """Return the square of a"""
    return a ** 2
```

Найдя искомую команду, нажмите Enter, и поиск завершится. После этого можно использовать найденную команду и продолжить работу в сеансе:

```
In [1]: def square(a):
    """Return the square of a"""
    return a ** 2

In [2]: square(2)
Out[2]: 4
```

Обратите внимание, что также можно использовать сочетания клавиш Ctrl+P/Ctrl+N или стрелки вверх/вниз для поиска по истории команд, но только по совпадающим символам в начале строки. Если вы введете `def` и нажмете Ctrl+P, в истории будет найдена последняя команда, начинающаяся с символов `def`, если таковая имеется.

Прочие горячие клавиши

Имеется еще несколько сочетаний клавиш, не относящихся ни к одной из предыдущих категорий, но заслуживающих упоминания (табл. 1.4).

Таблица 1.4. Дополнительные горячие клавиши

Комбинация клавиш	Действие
Ctrl+L	Очистить экран терминала
Ctrl+C (или стрелка «вниз»)	Прервать выполнение текущей команды Python
Ctrl+D	Выйти из сеанса Ipython[1]

Сочетание Ctrl+C особенно удобно при случайном запуске очень долго работающего задания.

Хотя использование некоторых сочетаний горячих клавиш может показаться утомительным, вскоре у вас появится соответствующая мышечная память и вы будете жалеть, что эти команды недоступны в других программах.

[1] В отличие от упомянутого выше идентичного сочетания горячих клавиш, это сочетание работает при пустой строке приглашения к вводу.

ГЛАВА 2

Расширенные интерактивные возможности

https://t.me/it_boooks/2

Большая часть возможностей IPython и Jupyter обеспечивается дополнительными интерактивными инструментами. В этой главе мы рассмотрим ряд таких инструментов, в том числе так называемые магические команды, инструменты для исследования истории ввода и вывода, а также для взаимодействия с оболочкой.

Магические команды IPython

В предыдущей главе мы познакомились с возможностями эффективного использования и изучения языка Python с помощью оболочки IPython. А здесь мы начнем обсуждать некоторые предоставляемые IPython расширения обычного синтаксиса языка Python. В IPython они известны как *«магические» команды* (magic commands) и отличаются префиксом `%`. Эти «магические» команды предназначены для быстрого решения различных распространенных задач стандартного анализа данных. Существуют два вида «магических» команд: *строчные «магические» команды* с префиксом с одним символом `%` и работающие с одной строкой ввода, и *блочные «магические» команды* с префиксом с двумя символами `%%` и работающие с несколькими строками ввода. Мы обсудим несколько коротких примеров и потом вернемся к более подробному обсуждению некоторых «магических» команд.

Выполнение внешнего кода: %run

Начав писать более обширный код, вы, вероятно, будете проводить исследования в интерактивной оболочке IPython и пользоваться текстовым редактором, чтобы сохранить код, который предполагается использовать неоднократно. Будет удобно выполнять этот код не в отдельном окне, а непосредственно в сеансе оболочки IPython. Для этого можно применять «магическую» функцию `%run`.

Например, допустим, что вы создали файл `myscript.py`, содержащий:

```
# Файл: myscript.py

def square(x):
    """square a number"""
    return x ** 2

for N in range(1, 4):
    print(f"{N} squared is {square(N)}")
```

Вот как можно запустить его в сеансе IPython:

```
In [1]: %run myscript.py
1 squared is 1
2 squared is 4
3 squared is 9
```

Обратите внимание, что после выполнения этого сценария все функции, которые в нем определены, становятся доступными для использования в сеансе оболочки IPython:

```
In [2]: square(5)
Out[2]: 25
```

Существует несколько параметров для точной настройки способа выполнения кода. Посмотреть относящуюся к этому документацию можно обычным способом, набрав команду `%run?` в интерпретаторе IPython.

Измерение продолжительности выполнения кода: %timeit

Еще одна полезная «магическая» функция — `%timeit`, которая автоматически определяет продолжительность выполнения следующей за ней инструкции на языке Python. Например, вот как можно измерить производительность генератора списков:

```
In [3]: %timeit L = [n ** 2 for n in range(1000)]
430 µs ± 3.21 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Преимущество использования функции `%timeit` в том, что она автоматически выполнит переданную ей инструкцию множество раз, чтобы получить максимально надежный результат. Для многострочных инструкций добавление второго знака `%` превратит ее в блочную «магическую» функцию, способную запускать многострочные инструкции. Например, вот эквивалентная конструкция для цикла `for`:

```
In [4]: %%timeit
   ...: L = []
   ...: for n in range(1000):
   ...:     L.append(n ** 2)
   ...:
484 µs ± 5.67 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Справка по «магическим» функциям: ?, %magic и %lsmagic

Подобно обычным функциям в Python, «магические» функции IPython имеют свои строки документации (docstring), и к ним можно обратиться обычным способом. Например, чтобы просмотреть документацию по «магической» функции `%timeit`, просто введите:

```
In [5]: %timeit?
```

Документацию с описанием других функций можно получить аналогичным образом. Для доступа к общему описанию имеющихся «магических» функций введите команду:

```
In [6]: %magic
```

Чтобы получить простой список всех доступных «магических» функций, введите:

```
In [7]: %lsmagic
```

Наконец, как уже упоминалось выше, вы тоже можете добавлять описания в свои «магические» функции. Если вас интересует этот вопрос, то загляните в раздел «Дополнительные источники информации об оболочке IPython» главы 3.

История ввода и вывода

Как мы уже видели, командная оболочка IPython позволяет получать доступ к предыдущим командам с помощью стрелок «вверх» (↑) и «вниз» (↓) или сочетаний клавиш Ctrl+P/Ctrl+N. Кроме того, в командной оболочке и в блокнотах IPython дает возможность получать вывод предыдущих команд, а также строковые версии самих этих команд.

Объекты In и Out оболочки IPython

Полагаю, вы уже хорошо знакомы с приглашениями `In[1]:`/`Out[1]:` в оболочке IPython. Это не просто изящные украшения; они подсказывают, как можно обратиться к предыдущим вводам и выводам в текущем сеансе. Допустим, вы начали сеанс, который выглядит так:

```
In [1]: import math

In [2]: math.sin(2)
Out[2]: 0.9092974268256817
```

```
In [3]: math.cos(2)
Out[3]: -0.4161468365471424
```

Мы импортировали встроенный пакет `math`, затем вычислили синус и косинус числа 2. Ввод и вывод отображаются в командной оболочке с метками `In`/`Out`, но за этим кроется нечто большее: оболочка IPython на самом деле создает переменные Python с именами `In` и `Out`, автоматически обновляемые так, что они отражают историю:

```
In [4]: In
Out[4]: ['', 'import math', 'math.sin(2)', 'math.cos(2)', 'In']

In [5]: Out
Out[5]:
{2: 0.9092974268256817,
 3: -0.4161468365471424,
 4: ['', 'import math', 'math.sin(2)', 'math.cos(2)', 'In', 'Out']}
```

Объект `In` — это список, хранящий последовательность команд (первый элемент этого списка — «заглушка», чтобы элемент `In[1]` ссылался на первую команду):

```
In [6]: print(In[1])
import math
```

Объект `Out` — не список, а словарь, связывающий ввод с выводом (если таковой есть).

```
In [7]: print(Out[2])
.9092974268256817
```

Обратите внимание, что не все операции генерируют вывод: например, операторы `import` и `print` на вывод не влияют. Последнее заявление может показаться странным, но смысл его становится понятен, если знать, что функция `print` возвращает `None`, а все команды, возвращающие `None`, не вносят вклада в объект `Out`.

Это может пригодиться для манипуляций с ранее полученными результатами. Например, вычислим сумму sin`(2) ** 2` и `cos(2) ** 2`, используя найденные ранее значения:

```
In [8]: Out[2] ** 2 + Out[3] ** 2
Out[8]: 1.0
```

Результат получился равным `1.0`, как и следовало ожидать из хорошо известного тригонометрического тождества. В данном случае использовать ранее полученные результаты, вероятно, не было необходимости, но эта возможность может очень пригодиться для повторного использования результатов ресурсоемких вычислений!

Быстрый доступ к предыдущим выводам с помощью знака подчеркивания

В обычной командной оболочке Python имеется лишь одна функция быстрого доступа к предыдущему выводу: значение переменной _ (единственный символ подчеркивания) соответствует предыдущему выводу. Он работает точно так же и в оболочке IPython:

```
In [9]: print(_)
.0
```

Но IPython не останавливается на этом и предлагает двойной символ подчеркивания для доступа к выводу на шаг ранее и тройной — для предшествовавшего ему (не считая команд, не генерировавших никакого вывода):

```
In [10]: print(__)
-0.4161468365471424

In [11]: print(___)
.9092974268256817
```

Но дальше оболочка IPython не идет: более трех подчеркиваний уже сложно отсчитывать и на этом этапе проще сослаться на вывод по номеру строки.

Однако существует еще одна функция быстрого доступа, заслуживающая упоминания: сокращенная форма записи для `Out[X]` выглядит как `_X` (символ подчеркивания с номером строки):

```
In [12]: Out[2]
Out[12]: 0.9092974268256817

In [13]: _2
Out[13]: 0.9092974268256817
```

Подавление вывода

Иногда может понадобиться подавить вывод инструкции (чаще такая потребность возникает при работе с командами рисования графиков, которые мы рассмотрим в части IV). Бывает так, что выполняемая команда выводит результат, который не требуется сохранять в истории команд, например, чтобы соответствующий ресурс можно было освободить после удаления других ссылок. Простейший способ подавления вывода — добавить точку с запятой в конце строки:

```
In [14]: math.sin(2) + math.cos(2);
```

Обратите внимание, что результат при этом вычисляется «молча» и вывод не отображается на экране и не сохраняется в словаре `Out`:

```
In [15]: 14 in Out
Out[15]: False
```

Соответствующие «магические» команды

Для доступа сразу к нескольким предыдущим вводам весьма удобно использовать «магическую» команду `%history`. Вот как можно вывести первые четыре ввода:

```
In [16]: %history -n 1-3
   1: import math
   2: math.sin(2)
   3: math.cos(2)
```

Как обычно, вы можете ввести команду `%history?`, чтобы получить дополнительную информацию о ней и описания доступных параметров (см. описание использования символа `?` в главе 1). Другие аналогичные «магические» команды — `%rerun` (выполняющая повторно заданную часть истории команд) и `%save` (сохраняющая какую-либо часть истории команд в файле).

IPython и использование системного командного процессора

При интерактивной работе со стандартным интерпретатором Python вы столкнетесь с досадным неудобством в виде необходимости переключаться между несколькими окнами для обращения к инструментам Python и системным утилитам командной строки. Оболочка IPython исправляет эту ситуацию, предоставляя пользователям синтаксис для выполнения инструкций системного командного процессора непосредственно из терминала IPython. Для этого используется восклицательный знак: все инструкции, следующие за `!`, будут выполняться не ядром языка Python, а системной командной оболочкой.

Далее в этом разделе предполагается, что вы работаете в Unix-подобной операционной системе, например Linux или macOS. Некоторые из следующих примеров не будут работать в операционной системе Windows, использующей по умолчанию другой тип командного процессора, однако если вы используете подсистему Windows для Linux (https://oreil.ly/H5MEE), то примеры должны выполняться корректно. Если инструкции системного командного процессора вам не знакомы, рекомендую просмотреть руководство по нему (https://oreil.ly/RrD2Y), составленное фондом Software Carpentry.

Краткое введение в использование командного процессора

Полный вводный курс использования командного процессора/терминала/командной строки выходит за пределы данной главы, но непосвященных мы кратко познакомим с ним. Командный процессор — способ текстового взаимодействия с компьютером. Начиная с середины 1980-х годов, когда корпорации Microsoft и Apple представили первые версии своих графических операционных систем, большинство пользователей компьютеров взаимодействуют со своей операционной системой посредством привычных щелчков кнопкой мыши на меню и движений «перетаскивания». Но операционные системы существовали задолго до этих графических интерфейсов пользователя и управлялись в основном посредством ввода текста: в приглашении командной строки пользователь вводил команду, а компьютер выполнял ее. Эти первые системы командной строки были предшественниками командных процессоров и терминалов, используемых до сих пор наиболее деятельными специалистами по науке о данных.

Человек, не знакомый с командными процессорами, мог бы задать вопрос: зачем вообще тратить на это время, если можно многого добиться, просто щелкая на пиктограммах и меню? Пользователь командного процессора мог бы ответить на этот вопрос другим вопросом: зачем гоняться за пиктограммами и щелкать на меню, если того же самого можно добиться гораздо проще, с помощью ввода команд? Хотя это может показаться типичным вопросом предпочтений, при выходе за пределы простых задач быстро становится понятно, что командный процессор предоставляет неизмеримо больше возможностей управления для сложных задач, даже при том, что кривая обучения немного круче.

В качестве примера приведу фрагмент сеанса командного процессора операционной системы Linux/macOS, в котором пользователь просматривает, создает и меняет каталоги и файлы в своей системе (`osx:~ $` представляет собой приглашение к вводу, а все после знака `$` — набираемая команда; текст, перед которым указан символ `#`, — это просто описание, а не действительно вводимый вами текст):

```
osx:~ $ echo "hello world"          # echo подобна функции print в Python
hello world

osx:~ $ pwd                         # pwd выводит путь к рабочему каталогу
/home/jake                          # это "путь" к текущему каталогу

osx:~ $ ls                          # ls выводит содержимое
notebooks  projects                 # рабочего каталога

osx:~ $ cd projects/                # cd меняет каталог

osx:projects $ pwd
/home/jake/projects

osx:projects $ ls
datasci_book   mpld3   myproject.txt
```

```
osx:projects $ mkdir myproject          # mkdir создает новый каталог

osx:projects $ cd myproject/

osx:myproject $ mv ../myproject.txt ./  # mv перемещает файл. Здесь мы перемещаем
                                        # файл myproject.txt, находящийся в каталоге
                                        # уровнем выше (../) текущего каталога (./)
osx:myproject $ ls
myproject.txt
```

Обратите внимание, что все это лишь краткий способ выполнения привычных операций (навигации по дереву каталогов, создания каталога, перемещения файла и т. д.) путем набора команд вместо щелчков на пиктограммах и меню. Кроме того, с помощью всего нескольких команд (`pwd`, `ls`, `cd`, `mkdir` и `cp`) можно выполнить большинство распространенных операций с файлами. А уж когда вы выходите за эти простейшие операции, подход с использованием командного процессора открывает по-настоящему широкие возможности.

Инструкции командного процессора в оболочке IPython

В оболочке IPython можно использовать любые команды, работающие в командной строке, просто поставив перед ними символ `!`. Например, команды `ls`, `pwd` и `echo` можно выполнить так:

```
In [1]: !ls
myproject.txt

In [2]: !pwd
/home/jake/projects/myproject

In [3]: !echo "printing from the shell"
printing from the shell
```

Передача значений в командный процессор и из него

Инструкции командного процессора не только могут запускаться из оболочки IPython, но также взаимодействовать с пространством имен IPython. Например, вывод любой инструкции командного процессора можно сохранить с помощью оператора присваивания (`=`):

```
In [4]: contents = !ls

In [5]: print(contents)
['myproject.txt']

In [6]: directory = !pwd

In [7]: print(directory)
['/Users/jakevdp/notebooks/tmp/myproject']
```

Обратите внимание, что эти результаты возвращаются не в виде списков, а как специальный тип возвращаемого значения, определенный в IPython для командного процессора:

```
In [8]: type(directory)
IPython.utils.text.Slist
```

Этот тип выглядит и работает во многом подобно спискам Python, но у него есть и дополнительная функциональность, в частности методы `grep` и `fields`, а также свойства `s`, `n` и `p`, позволяющие выполнять поиск, фильтрацию и отображение результатов. Чтобы узнать об этом больше, воспользуйтесь встроенными справочными возможностями оболочки IPython.

Отправка информации в обратную сторону — передача переменных Python в командный процессор — возможна с применением синтаксиса `{`*`имя_переменной`*`}`:

```
In [9]: message = "Hello from Python"

In [10]: !echo {message}
Hello from Python
```

Фигурные скобки содержат имя переменной, заменяемое в инструкции командного процессора ее значением.

«Магические» команды для командного процессора

Поэкспериментировав немного с инструкциями командного процессора, вы можете обратить внимание, что команда `!cd` не позволяет перемещаться по файловой системе:

```
In [11]: !pwd
/home/jake/projects/myproject

In [12]: !cd ..

In [13]: !pwd
/home/jake/projects/myproject
```

Причина заключается в том, что инструкции командного процессора в блокноте оболочки IPython выполняются во временной командной подоболочке. Чтобы действительно сменить рабочий каталог, можно воспользоваться «магической» командой `%cd`:

```
In [14]: %cd ..
/home/jake/projects
```

На самом деле по умолчанию ее можно использовать даже без символа `%`:

```
In [15]: cd myproject
/home/jake/projects/myproject
```

Такие функции называют «автомагическими» (automagic), и их можно настраивать с помощью «магической» функции `%automagic`.

Кроме `%cd`, доступны и другие «автомагические» функции: `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm` и `%rmdir`, каждую из которых можно применять без знака `%`, если активизировано поведение `automagic`. При этом командную строку оболочки IPython можно использовать практически как обычный командный процессор:

```
In [16]: mkdir tmp

In [17]: ls
myproject.txt  tmp/

In [18]: cp myproject.txt tmp/

In [19]: ls tmp
myproject.txt

In [20]: rm -r tmp
```

Доступ к командному процессору в том же окне терминала, где выполняется сеанс Python, означает резкое снижение числа необходимых переключений между интерпретатором и командным процессором при написании кода на языке Python.

ГЛАВА 3

Отладка и профилирование

В дополнение к расширенным интерактивным инструментам, рассмотренным в предыдущей главе, Jupyter предлагает несколько способов исследования кода, например, с целью поиска ошибок в логике или неожиданно медленного выполнения. Эта глава обсуждает некоторые из этих инструментов.

Ошибки и отладка

Разработка кода и анализ данных всегда предполагают движение вперед методом проб и ошибок, и в оболочке IPython есть инструменты для упрощения этого процесса. В данном разделе будут вкратце рассмотрены некоторые возможности по управлению оповещением об ошибках в Python, а также утилиты для отладки ошибок в коде.

Управление исключениями: %xmode

Почти всегда при сбое сценарии на Python генерируют исключение. В случае появления любого из таких исключений информацию о его причине можно найти в *трассировке* (traceback), к которой можно обратиться из Python. С помощью «магической» функции `%xmode` оболочка IPython дает возможность управлять количеством информации, которая выводится, когда генерируется исключение. Рассмотрим следующий код:

```
In[1]: def func1(a, b):
           return a / b

       def func2(x):
           a = x
           b = x - 1
           return func1(a, b)
```

```
In[2]: func2(1)
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-b2e110f6fc8f^gt; in <module>()
----> 1 func2(1)

<ipython-input-1-d849e34d61fb> in func2(x)
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

<ipython-input-1-d849e34d61fb> in func1(a, b)
      1 def func1(a, b):
----> 2     return a / b
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero
```

Вызов функции `func2` приводит к ошибке, и чтение трассировки позволяет точно понять, что произошло. По умолчанию трассировка включает несколько строк, описывающих контекст каждого из шагов, приведших к ошибке. С помощью «магической» функции `%xmode` (сокращение от *exception mode* — режим отображения исключений) можно управлять тем, какая информация будет выводиться.

Функция `%xmode` принимает единственный аргумент, режим, который может иметь одно из трех значений: `Plain` (простой), `Context` (по контексту) и `Verbose` (подробный). Режим по умолчанию — `Context`. Вывод в этом режиме показан выше. Режим `Plain` дает более сжатый вывод и меньше информации:

```
In [3]: %xmode Plain
Out[3]: Exception reporting mode: Plain

In [4]: func2(1)
Traceback (most recent call last):

  File "<ipython-input-4-b2e110f6fc8f>", line 1, in <module>
    func2(1)

  File "<ipython-input-1-d849e34d61fb>", line 7, in func2
    return func1(a, b)

  File "<ipython-input-1-d849e34d61fb>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero
```

Режим `Verbose` добавляет еще некоторую информацию, включая аргументы всех вызываемых функций:

```
In [5]: %xmode Verbose
Out[5]: Exception reporting mode: Verbose

In [6]: func2(1)
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-6-b2e110f6fc8f> in <module>()
----> 1 func2(1)
        global func2 = <function func2 at 0x103729320>

<ipython-input-1-d849e34d61fb> in func2(x=1)
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)
        global func1 = <function func1 at 0x1037294d0>
        a = 1
        b = 0

<ipython-input-1-d849e34d61fb> in func1(a=1, b=0)
      1 def func1(a, b):
----> 2     return a / b
        a = 1
        b = 0
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero
```

Эта дополнительная информация может помочь сузить круг возможных причин исключения. Почему бы тогда не использовать режим `Verbose` всегда? Дело в том, что исключения в сложном коде могут порождать очень длинные трассировки. В зависимости от контекста иногда проще работать с более кратким выводом, который дает режим по умолчанию.

Отладка: что делать, если информации в трассировке недостаточно

Стандартная утилита для интерактивной отладки кода на Python называется `pdb` (сокращенно от Python Debugger — «отладчик Python»). Она дает возможность выполнять код строку за строкой, чтобы выяснить, что могло стать причиной ошибки. Расширенная версия этого отладчика в оболочке IPython называется `ipdb` (сокращенно от IPython Debugger — «отладчик IPython»).

Существует множество способов запуска и использования этих отладчиков; мы не станем описывать их все. Для более полной информации по данным утилитам обращайтесь к онлайн-документации.

Вероятно, наиболее удобный интерфейс для отладки в IPython — «магическая» команда %debug. Если ее вызвать после встречи с исключением, она автоматически откроет интерактивную командную строку откладки в точке возникновения ошибки. Командная строка ipdb позволяет изучать текущее состояние стека, доступные переменные и даже выполнять команды Python!

Посмотрим на самые недавние исключения, затем выполним несколько простейших действий — выведем значения a и b, а затем наберем quit для выхода из сеанса отладки:

```
In [7]: %debug <ipython-input-1-d849e34d61fb>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3
ipdb> print(a)
1
ipdb> print(b)
0
ipdb> quit
```

Однако, кроме этого, интерактивный отладчик позволяет перемещаться вверх и вниз по стеку, изучая значения переменных:

```
In [8]: %debug <ipython-input-1-d849e34d61fb>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3

ipdb> up <ipython-input-1-d849e34d61fb>(7)func2()
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

ipdb> print(x)
1
ipdb> up <ipython-input-6-b2e110f6fc8f>(1)<module>()
----> 1 func2(1)

ipdb> down <ipython-input-1-d849e34d61fb>(7)func2()
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

ipdb> quit
```

Это позволяет быстро находить не только причины ошибок, но и вызовы функций, приведших к ним.

Чтобы отладчик автоматически запускался при появлении исключения, можно воспользоваться «магической» функцией `%pdb` для включения такого автоматического поведения:

```
In [9]: %xmode Plain
        %pdb on
        func2(1)
Exception reporting mode: Plain
Automatic pdb calling has been turned ON
ZeroDivisionError: division by zero <ipython-input-1-d849e34d61fb>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3

ipdb> print(b)
0
ipdb> quit
```

Наконец, если у вас есть сценарий, который желательно запускать в начале работы в интерактивном режиме, то его можно запустить с помощью команды `%run -d` и использовать команду `next` для пошагового интерактивного перемещения по строкам кода.

В табл. 3.1 перечислены команды для интерактивной отладки, но это неполный список — на самом деле их намного больше.

Таблица 3.1. Неполный список команд отладки

Команда	Описание
`l(ist)`	Отображает текущее место в файле
`h(elp)`	Отображает список команд или справку по конкретной команде
`q(uit)`	Выход из отладчика и программы
`c(ontinue)`	Выход из отладчика, продолжение выполнения программы
`n(ext)`	Переход к следующему шагу программы
`<enter>`	Повтор предыдущей команды
`p(rint)`	Вывод [значений] переменных
`s(tep)`	Вход в подпрограмму
`r(eturn)`	Возврат из подпрограммы

Для получения дополнительной информации используйте команду `help` в отладчике или загляните в онлайн-документацию по `ipdb` (https://oreil.ly/TVSAT).

Профилирование и хронометраж выполнения кода

В процессе разработки кода и создания конвейеров обработки данных всегда присутствуют компромиссы между различными реализациями. В начале создания алгоритма забота о подобных вещах может оказаться контрпродуктивной. Согласно знаменитому афоризму Дональда Кнута: «Лучше не держать в голове подобные “малые” вопросы производительности, скажем, в 97 % случаев: преждевременная оптимизация — корень всех зол».

Однако, как только ваш код начинает работать, полезно заняться его производительностью. Иногда бывает удобно оценить время выполнения заданной команды или набора команд, а иногда — покопаться в процессе, состоящем из множества строк, и выяснить, где находится узкое место. Оболочка IPython предоставляет широкий выбор возможностей для подобного мониторинга производительности кода и его профилирования. Здесь мы обсудим следующие «магические» команды оболочки IPython:

- `%time` — длительность выполнения отдельной инструкции;
- `%timeit` — длительность выполнения отдельной инструкции при неоднократном выполнении, для большей точности;
- `%prun` — выполнение кода с использованием профилировщика;
- `%lprun` — пошаговое выполнение кода с применением профилировщика;
- `%memit` — оценка потребления оперативной памяти отдельной инструкцией;
- `%mprun` — пошаговое выполнение кода с применением профилировщика памяти.

Последние четыре команды не включены в пакет IPython — для их использования необходимо установить расширения `line_profiler` и `memory_profiler`, которые мы обсудим в следующих разделах.

Хронометраж выполнения фрагментов кода: %timeit и %time

Мы уже встречались с «магическими» командами `%timeit` и `%%timeit` во введении в «магические» функции в разделе «Магические команды IPython» главы 2;

команду `%%timeit` можно использовать для хронометража многократного выполнения фрагментов кода:

```
In [1]: %timeit sum(range(100))
1.53 µs ± 47.8 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Обратите внимание, что, поскольку данная операция должна выполняться очень быстро, команда `%timeit` автоматически выполняет ее много раз. При анализе медленных инструкций команда `%timeit` автоматически подстроится и выполнит их меньшее количество раз:

```
In [2]: %%timeit
        total = 0
        for i in range(1000):
            for j in range(1000):
                total += i * (-1) ** j
536 ms ± 15.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Иногда повторное выполнение операции — не лучший вариант. Например, в случае сортировки списка повтор операции мог бы ввести нас в заблуждение. Сортировка предварительно отсортированного списка происходит намного быстрее, чем сортировка неотсортированного, так что повторное выполнение исказит результат:

```
In [3]: import random
        L = [random.random() for i in range(100000)]
        %timeit L.sort()
Out[3]: 1.71 ms ± 334 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

В этом случае лучшим выбором будет «магическая» функция `%timeit`. Она также подойдет для анализа долго выполняющихся команд, в которых короткие, системно обусловленные задержки вряд ли существенно повлияют на результат. Оценим время сортировки неотсортированного и предварительно отсортированного списков:

```
In [4]: import random
        L = [random.random() for i in range(100000)]
        print("sorting an unsorted list:")
        %time L.sort()
Out[4]: sorting an unsorted list:
        CPU times: user 31.3 ms, sys: 686 µs, total: 32 ms
        Wall time: 33.3 ms

In [5]: print("sorting an already sorted list:")
        %time L.sort()
Out[5]: sorting an already sorted list:
        CPU times: user 5.19 ms, sys: 268 µs, total: 5.46 ms
        Wall time: 14.1 ms
```

Обратите внимание, насколько быстрее сортируется предварительно отсортированный список, а также насколько больше времени занимает хронометраж с помощью функции `%time` по сравнению с функцией `%timeit`, даже в случае предварительно отсортированного списка! Это обусловлено тем, что «магическая» функция `%timeit` незаметно для нас применяет некоторые хитрые трюки, чтобы системные вызовы не мешали оценке времени. Например, она запрещает удаление неиспользуемых объектов Python (известное как сборка мусора), которое могло бы повлиять на оценку времени выполнения. Именно поэтому выдаваемое функцией `%timeit` время обычно заметно меньше времени, выдаваемого функцией `%time`.

Обе команды, `%time` и `%timeit`, поддерживают синтаксис с двойным знаком процента, позволяя оценивать время выполнения многострочных сценариев:

```
In [6]: %%time
        total = 0
        for i in range(1000):
            for j in range(1000):
                total += i * (-1) ** j
CPU times: user 655 ms, sys: 5.68 ms, total: 661 ms
Wall time: 710 ms
```

За дополнительной информацией по «магическим» функциям `%time` и `%timeit`, а также их параметрам обращайтесь к справочным функциям оболочки IPython (например, наберите `%time?` в командной строке IPython).

Профилирование сценариев целиком: %prun

Программы состоят из множества отдельных инструкций, и иногда хронометраж их выполнения в контексте важнее, чем по отдельности. В языке Python имеется встроенный профилировщик кода (о котором можно прочитать в документации Python), но оболочка IPython предоставляет намного более удобный способ его использования в виде «магической» функции `%prun`.

В качестве примера я опишу простую функцию, выполняющую некоторые вычисления:

```
In [7]: def sum_of_lists(N):
            total = 0
            for i in range(5):
                L = [j ^ (j >> i) for j in range(N)]
                total += sum(L)
            return total
```

Теперь обратимся к «магической» функции `%prun` и передадим ей вызов функции, чтобы увидеть результаты профилирования:

```
In [8]: %prun sum_of_lists(1000000)
14 function calls in 0.932 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     5    0.808    0.162    0.808  0.162 <ipython-input-7-f105717832a2>:4(<listcomp>)
     5    0.066    0.013    0.066  0.013 {built-in method builtins.sum}
     1    0.044    0.044    0.918  0.918 <ipython-input-7-f105717832a2>:1
     > (sum_of_lists)
     1    0.014    0.014    0.932  0.932 <string>:1(<module>)
     1    0.000    0.000    0.932  0.932 {built-in method builtins.exec}
     1    0.000    0.000    0.000  0.000 {method 'disable' of '_lsprof.Profiler'
     > objects}
```

Результат выводится в виде таблицы, в которой перечислены функции в порядке уменьшения общего времени, затраченного на каждый вызов. В данном случае бóльшую часть времени занимает генератор списков внутри функции `sum_of_lists`. Зная это, можно начинать обдумывать возможные изменения в алгоритме для улучшения производительности.

За дополнительной информацией по «магической» функции `%prun`, а также ее параметрам обращайтесь к функциям получения справки в оболочке IPython (то есть наберите `%prun?` в командной строке IPython).

Пошаговое профилирование с помощью %lprun

Профилирование по функциям с помощью `%prun` довольно удобно, но иногда больше пользы может принести отчет профилировщика о продолжительности выполнения отдельных строк. Такая функциональность не включена по умолчанию в Python или IPython, но можно установить пакет `line_profiler`, обладающий такой возможностью. Для начала воспользуемся утилитой `pip`, чтобы установить пакет `line_profiler`:

```
$ pip install line_profiler
```

Далее можно воспользоваться IPython для загрузки расширения `line_profiler` в оболочку IPython, входящего в состав указанного пакета:

```
In [9]: %load_ext line_profiler
```

Теперь можно воспользоваться командой `%lprun` для построчного профилирования любой функции. В нашем случае необходимо указать явно, какие функции мы хотели бы профилировать:

```
In [10]: %lprun -f sum_of_lists sum_of_lists(5000)
Timer unit: 1e-06 s

Total time: 0.014803 s
File: <ipython-input-7-f105717832a2>
Function: sum_of_lists at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_of_lists(N):
     2         1          6.0      6.0      0.0      total = 0
     3         6         13.0      2.2      0.1      for i in range(5):
     4         5      14242.0   2848.4     96.2          L = [j ^ (j >> i) for j
     5         5        541.0    108.2      3.7          total += sum(L)
     6         1          1.0      1.0      0.0      return total
```

Информация в заголовке дает ключ к чтению результатов: время указывается в микросекундах, и мы можем увидеть, в каком месте программа проводит больше всего времени. На этой стадии мы получаем возможность использовать эту информацию для модификации кода и улучшения его производительности.

За дополнительной информацией о «магической» функции `%lprun`, а также о доступных параметрах обращайтесь к функциям получения справки в оболочке IPython (то есть наберите `%lprun?` в командной строке IPython).

Профилирование потребления памяти: %memit и %mprun

Другой аспект профилирования — объем потребляемой операциями памяти. Этот объем можно оценить с помощью еще одного расширения оболочки IPython — `memory_profiler`. Как и в случае с утилитой `line_profiler`, начнем с установки расширения с помощью утилиты `pip`:

```
$ pip install memory_profiler
```

Затем используем оболочку IPython для загрузки расширения:

```
In [11]: %load_ext memory_profiler
```

Профилировщик памяти предлагает две удобные «магические» функции: `%memit` и `%mprun` (аналоги `%timeit` и `%lprun` соответственно для измерения потребления памяти). Применять функцию `%memit` несложно:

```
In [12]: %memit sum_of_lists(1000000)
peak memory: 141.70 MiB, increment: 75.65 MiB
```

Мы видим, что данная функция использует около 100 Мбайт памяти.

Для оценки потребления памяти отдельными строками можно использовать «магическую» функцию `%mprun`. К сожалению, она работает только для функций, описанных в отдельных модулях, а не в самом блокноте, так что начнем с применения «магической» функции `%%file` для создания простого модуля с именем `mprun_demo.py`, содержащего нашу функцию `sum_of_lists`, с одним дополнением, которое немного прояснит нам результаты профилирования памяти:

```
In [13]: %%file mprun_demo.py
         def sum_of_lists(N):
             total = 0
             for i in range(5):
                 L = [j ^ (j >> i) for j in range(N)]
                 total += sum(L)
                 del L # удалить ссылку на L
             return total
Overwriting mprun_demo.py
```

Теперь можно импортировать новую версию нашей функции и запустить построчный профилировщик памяти:

```
In [14]: from mprun_demo import sum_of_lists
         %mprun -f sum_of_lists sum_of_lists(1000000)

Filename: /Users/jakevdp/github/jakevdp/PythonDataScienceHandbook/notebooks_v2/
> m prun_demo.py

Line #    Mem usage    Increment  Occurrences   Line Contents
============================================================
     1     66.7 MiB     66.7 MiB           1   def sum_of_lists(N):
     2     66.7 MiB      0.0 MiB           1       total = 0
     3     75.1 MiB      8.4 MiB           6       for i in range(5):
     4    105.9 MiB     30.8 MiB     5000015           L = [j ^ (j >> i) for j
     5    109.8 MiB      3.8 MiB           5           total += sum(L)
     6     75.1 MiB    -34.6 MiB           5           del L # удалить ссылку на L
     7     66.9 MiB     -8.2 MiB           1       return total
```

Столбец `Increment` сообщает, какой вклад в потребление памяти вносит каждая строка. Обратите внимание, что создание и удаление списка `L` добавляет примерно 30 Мбайт в общее потребление памяти, помимо потребления памяти самим интерпретатором Python.

За дополнительной информацией о «магических» функциях `%memit` и `%mprun`, а также о доступных параметрах обращайтесь к функциям получения справки в оболочке IPython (например, наберите `%memit?` в командной строке IPython).

Дополнительные источники информации об оболочке IPython

В данной части мы лишь в общих чертах рассмотрели применение языка Python для решения задач науки о данных. Гораздо больше информации доступно как в печатном виде, так и в Интернете. Здесь мы приведем ссылки на некоторые дополнительные ресурсы, которые могут вам пригодиться.

Веб-ресурсы

- *Сайт проекта IPython* (http://ipython.org/). Содержит ссылки на документацию, примеры, руководства и множество других ресурсов.
- *Сайт nbviewer* (http://nbviewer.jupyter.org/). Этот сайт демонстрирует статические визуализации из блокнотов Jupyter, доступных в Интернете. На главной странице сайта показано несколько примеров блокнотов, демонстрирующих практическое применение Python другими разработчиками!
- *Галерея интересных блокнотов Jupyter* (https://github.com/jupyter/jupyter/wiki). Этот непрерывно растущий список блокнотов, поддерживаемый nbviewer, демонстрирует глубину и размах численного анализа, возможного с помощью оболочки IPython. Он включает все, начиная от коротких примеров и руководств и заканчивая полноразмерными курсами и книгами, составленными в формате блокнотов!
- *Видеоруководства*. В Интернете вы найдете немало видеоруководств по оболочке IPython. Особенно рекомендую руководства с конференций PyCon, SciPy and PyData, написанные Фернандо Пересом и Брайаном Грейнджером — двумя основными разработчиками, создавшими и поддерживающими оболочку IPython и проект Jupiter.

Книги

- *Python for Data Analysis*[1] (https://oreil.ly/ik2g7). Эта книга Уэса Маккинли включает главу, описывающую использование оболочки IPython с точки зрения исследователя данных. Хотя многое в ней пересекается с тем, что мы тут обсуждали, вторая точка зрения никогда не помешает.
- *Learning IPython for Interactive Computing and Data Visualization* («Изучаем оболочку IPython для целей интерактивных вычислений и визуализации данных»).

[1] *Маккинли У.* Python и анализ данных.

Эта книга Цириллы Россан предлагает неплохое введение в использование оболочки IPython для анализа данных.

- *IPython Interactive Computing and Visualization Cookbook* («Справочник по интерактивным вычислениям и визуализации с помощью языка IPython»). Данная книга также написана Цириллой Россан и представляет собой более полное и продвинутое руководство по использованию оболочки IPython для науки о данных. Несмотря на название, она посвящена не только оболочке IPython, в книге рассмотрены некоторые темы, относящиеся к науке о данных.

Вы можете и сами найти справочные материалы: справка, доступная с помощью символа ? в оболочке IPython (обсуждалась в разделе «Справка и документация в IPython» главы 1), может оказаться чрезвычайно полезной, если ее использовать правильно. Работая с примерами из этой и других книг, используйте встроенную справку для знакомства со всеми утилитами, которые предоставляет IPython.

ЧАСТЬ II

Введение в NumPy

В этой части и в части III мы рассмотрим методы эффективной загрузки, хранения и управления данными, находящимися в оперативной памяти. Это очень широкая тема, потому что наборы данных могут поступать из разных источников и в разных форматах (наборы документов, изображений, аудиоклипов, массивы численных измерений и др.). Несмотря на столь очевидную разнородность, все эти данные удобно рассматривать как массивы числовых значений.

Например, изображения (особенно цифровые) можно представить в виде простых двумерных массивов чисел, отражающих яркость пикселов соответствующей области; аудиоклипы — как одномерные массивы интенсивности звука в определенные моменты времени. Текст можно преобразовать в числовое представление различными путями, например с двоичными числами, представляющими частоту слов или пар слов. Вне зависимости от характера данных первым шагом к их анализу является преобразование в числовые массивы (мы обсудим некоторые примеры этого процесса в главе 40 «Проектирование признаков»).

Эффективное хранение и работа с числовыми массивами очень важны для процесса исследования данных. Мы изучим специализированные инструменты языка Python, предназначенные для обработки подобных массивов, — пакет NumPy и пакет Pandas (обсуждается в части III).

В этой части мы подробно обсудим библиотеку NumPy (сокращенно от Numerical Python — «числовой Python»), обеспечивающую эффективный интерфейс для хранения и обработки плотных буферов данных. Массивы библиотеки NumPy похожи на тип данных `list` в языке Python, но обеспечивают гораздо более эффективное хранение и обработку данных с ростом размеров массивов. Массивы NumPy образуют основу практически всей экосистемы утилит Python для исследования данных. Поэтому время, потраченное на изучение пакета NumPy, не будет потрачено впустую вне зависимости от интересующего вас аспекта науки о данных.

Если вы последовали приведенному в предисловии совету и установили стек утилит Anaconda, то пакет NumPy уже готов к использованию. Если же вы относитесь к числу тех, кто любит все делать своими руками, то перейдите на сайт NumPy.org (http://www.numpy.org/) и следуйте имеющимся там указаниям. После этого импортируйте NumPy и тщательно проверьте его версию:

```
In [1]: import numpy
        numpy.__version__
Out[1]: '1.21.2'
```

Для целей обсуждения в этой главе я рекомендую NumPy версии 1.8 или выше. По традиции большинство людей в мире SciPy/PyData импортируют пакет NumPy под псевдонимом `np`:

```
In [2]: import numpy as np
```

На протяжении книги мы именно так и будем импортировать и применять NumPy.

НАПОМИНАНИЕ О ВСТРОЕННОЙ ДОКУМЕНТАЦИИ

Читая эту часть, не забывайте, что оболочка IPython позволяет быстро просматривать содержимое пакетов (посредством автодополнения при нажатии клавиши Tab), а также документацию по различным функциям (используя символ `?`). Загляните в раздел «Справка и документация в IPython» главы 1, если вам нужно освежить в памяти эти возможности.

Например, отобразить все содержимое пространства имен `numpy` можно командой:

```
In [3]: np.<TAB>
```

Получить встроенную справку о пакете NumPy можно командой:

```
In [4]: np?
```

Более подробные сведения, руководства и ссылки на другие источники информации можно найти на сайте http://www.numpy.org.

ГЛАВА 4

Типы данных в Python

Для реализации эффективных научных вычислений, ориентированных на работу с данными, необходимо знать, как хранятся и обрабатываются данные. В этой главе описываются и сравниваются способы обработки массивов данных в языке Python, а также усовершенствования, вносимые в этот процесс библиотекой NumPy. Знание различий очень важно для понимания большей части материала в книге.

Пользователей Python зачастую привлекает его простота, немаловажной частью которой является динамическая типизация. В то время как в языках со статической типизацией, таких как C и Java, необходимо явно объявлять все переменные, языки с динамической типизацией, например Python, этого не требуют. Например, вот как выглядит типичное определение операции на языке C:

```
/* Код на языке C */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

На Python та же операция определяется так:

```
# Код на языке Python
result = 0
for i in range(100):
    result += i
```

Обратите внимание на главное отличие: в языке C для каждой переменной явно объявляется ее тип, а в Python типы переменных определяются динамически. Это значит, что любой переменной можно присвоить данные любого типа:

```
# Код на языке Python
x = 4
x = "four"
```

Здесь мы сначала присвоили переменной `x` целое число, а затем строку. Подобное действие в языке C могло бы привести к ошибке компиляции или другим неожиданным последствиям в зависимости от настроек компилятора:

```
/* Код на языке C */
int x = 4;
x = "four";  // СБОЙ
```

Подобная гибкость делает Python и другие языки с динамической типизацией удобными и простыми в использовании. Понимать, *как* это работает, очень важно, чтобы научиться эффективно анализировать данные с помощью языка Python. Однако такая гибкость при работе с типами указывает на то, что переменные в Python — это нечто большее, чем просто значения, они содержат также дополнительную информацию о *типах* хранимых в них значений. Мы рассмотрим этот аспект подробнее в следующих разделах.

Целое число в Python — больше, чем просто целое число

Стандартная реализация Python написана на C. Это значит, что каждый объект Python — просто искусно замаскированная структура языка C, содержащая не только значение, но и другую информацию. Например, при описании целочисленной переменной на Python, такой как `x = 10000`, `x` — это не просто целое число, а указатель на составную структуру языка C, содержащую несколько значений. Заглянув в исходный код Python 3.10, можно заметить, что описание целочисленного типа (типа `long` в C) фактически выглядит так (после развертывания макросов языка C):

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

Всякое целое число в Python 3.10 фактически состоит из четырех частей:

- `ob_refcnt` — счетчик ссылок, с помощью которого Python незаметно выполняет выделение и освобождение памяти;
- `ob_type` — код типа переменной;
- `ob_size` — размер следующих элементов данных;
- `ob_digit` — фактическое целочисленное значение, которое представляет переменная в Python.

Это значит, что в отличие от компилирумых языков, таких как C, в Python вместе с самим числом хранится некоторая избыточная информация (рис. 4.1).

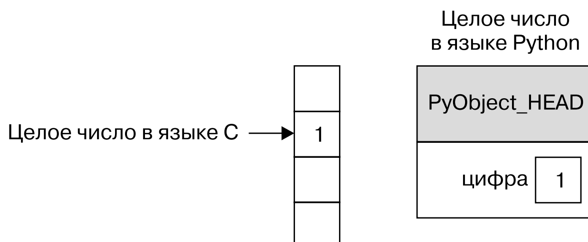


Рис. 4.1. Разница между целыми числами в языках C и Python

`PyObject_HEAD` на рис. 4.1 — часть структуры, содержащая счетчик ссылок, код типа и другие упомянутые выше элементы.

Еще раз акцентируем внимание на основном отличии: целочисленная переменная в языке C — это метка, определяющая место в памяти — байты, хранящие целочисленное значение. Целочисленная переменная в Python — это указатель на место в памяти, где хранится вся информация об объекте языка Python, включая байты, хранящие целочисленное значение. Именно эта дополнительная информация и позволяет так свободно программировать на языке Python с использованием динамической типизации. Однако эта дополнительная информация в типах Python влечет и накладные расходы, что особенно заметно при работе со структурами, объединяющими значительное количество таких объектов.

Список в Python — больше, чем просто список

Теперь посмотрим, что происходит при использовании структуры данных, содержащей много объектов. Стандартным контейнером для множества элементов в Python является список. Создать список целочисленных значений можно так:

```
In [1]: L = list(range(10))
        L
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [2]: type(L[0])
Out[2]: int
```

А так — список строк:

```
In [3]: L2 = [str(c) for c in L]
        L2
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
In [4]: type(L2[0])
Out[4]: str
```

Благодаря динамической типизации в Python можно создавать даже неоднородные списки:

```
In[5]: L3 = [True, "2", 3.0, 4]
       [type(item) for item in L3]

Out[5]: [bool, str, float, int]
```

Однако подобная гибкость имеет свою цену: для использования гибких типов данных каждый элемент списка должен содержать информацию о типе, счетчик ссылок и другую информацию, то есть каждый элемент — это целый объект языка Python. В частном случае, когда все элементы имеют один и тот же тип, бо́льшая часть этой информации избыточна: намного рациональнее хранить данные в массиве фиксированного типа. Различие между списком со значениями динамического типа и массивом фиксированного типа (в стиле библиотеки NumPy) показано на рис. 4.2.

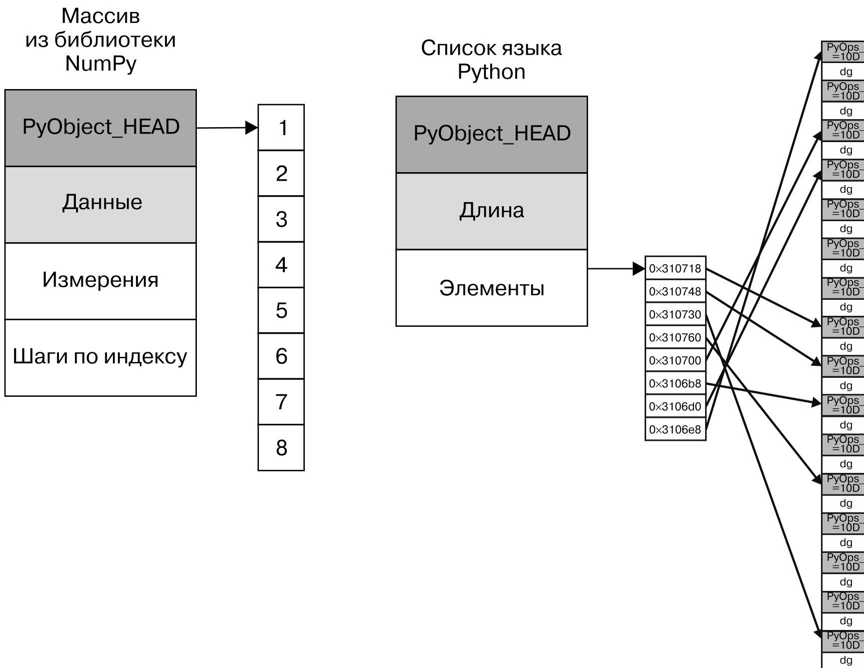


Рис. 4.2. Различие между списками в языках C и Python

На уровне реализации массив фактически содержит один указатель на непрерывный блок данных. Список в Python, в свою очередь, содержит указатель на блок указателей, каждый из которых указывает на целый объект языка

Python, например, представляющий целое число. Преимущество такого списка в его гибкости: каждый элемент списка — полномасштабная структура, содержащая данные и информацию о типе, поэтому список можно заполнить данными любого требуемого типа. Массивам фиксированного типа из библиотеки NumPy недостает этой гибкости, но они гораздо эффективнее хранят данные и работают с ними.

Массивы фиксированного типа в Python

Язык Python предоставляет несколько возможностей для хранения данных в эффективно работающих буферах фиксированного типа. Встроенный модуль `array` (доступен начиная с версии Python 3.3) можно использовать для создания плотных массивов данных одного типа:

```
In [6]: import array
        L = list(range(10))
        A = array.array('i', L)
        A
Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Здесь `'i'` — код типа, сообщающий, что массив хранит целые числа.

Намного удобнее объект `ndarray` из библиотеки NumPy. В то время как объект `array` в языке Python обеспечивает эффективное хранение данных в формате массива, библиотека NumPy добавляет еще и возможность выполнения эффективных *операций* с этими данными. Мы рассмотрим такие операции в следующих разделах, а здесь продемонстрируем несколько способов создания NumPy-массива.

Создание массивов из списков

Для начала импортируем пакет NumPy под псевдонимом `np`:

```
In[7]: import numpy as np
```

Для создания массивов из списков можно воспользоваться функцией `np.array`:

```
In[8]: # массив целочисленных значений:
       np.array([1, 4, 2, 5, 3])
Out[8]: array([1, 4, 2, 5, 3])
```

В отличие от списков языка Python, массивы NumPy могут хранить элементы только одного типа. Если типы элементов не совпадают, NumPy попытается выполнить повышающее приведение типов (в данном случае целочисленные значения приводятся к числам с плавающей точкой):

```
In[9]: np.array([3.14, 4, 2, 3])
```

```
Out[9]: array([ 3.14,  4.  ,  2.  ,  3.  ])
```

Чтобы явно задать тип итогового массива, можно воспользоваться именованным аргументом `dtype`:

```
In[10]: np.array([1, 2, 3, 4], dtype='float32')
Out[10]: array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Наконец, в отличие от списков в языке Python массивы NumPy можно явно описывать как многомерные. Вот один из способов инициализации многомерного массива с помощью списка списков:

```
In[11]: # Вложенные списки преобразуются в многомерный массив
        np.array([range(i, i + 3) for i in [2, 4, 6]])
Out[11]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

Внутренние списки интерпретируются как строки в итоговом двумерном массиве.

Создание массивов с нуля

Большие массивы эффективнее создавать с нуля с помощью имеющихся в пакете NumPy методов. Вот несколько примеров:

```
In[12]: # Создаем массив с десятью целыми числами, заполненный нулями
        np.zeros(10, dtype=int)
Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In[13]: # Создаем массив 3 × 5 значений с плавающей точкой,
        # заполненный единицами
        np.ones((3, 5), dtype=float)
Out[13]: array([[ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.]])

In[14]: # Создаем массив 3 × 5, заполненный значением 3.14
        np.full((3, 5), 3.14)
Out[14]: array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14]])

In[15]: # Создаем массив, заполненный линейной последовательностью,
        # начинающейся с 0 и заканчивающейся 20, с шагом 2
        # (действует подобно встроенной функции range())
        np.arange(0, 20, 2)
Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In[16]: # Создаем массив с пятью значениями, располагающимися
        # через равные интервалы в диапазоне между 0 и 1
        np.linspace(0, 1, 5)
```

```
Out[16]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])

In[17]: # Создаем массив 3 × 3 равномерно распределенных
        # случайных значений от 0 до 1
        np.random.random((3, 3))
Out[17]: array([[0.09610171, 0.88193001, 0.70548015],
                [0.35885395, 0.91670468, 0.8721031 ],
                [0.73237865, 0.09708562, 0.52506779]])

In[18]: # Создаем массив 3 × 3 нормально распределенных
        # случайных значений со средним 0 и стандартным отклонением 1
        np.random.normal(0, 1, (3, 3))
Out[18]: array([[-0.46652655, -0.59158776, -1.05392451],
                [-1.72634268,  0.03194069, -0.51048869],
                [ 1.41240208,  1.77734462, -0.43820037]])

In[19]: # Создаем массив 3 × 3 случайных целых чисел
        # в интервале [0, 10)
        np.random.randint(0, 10, (3, 3))
Out[19]: array([[4, 3, 8],
                [6, 5, 0],
                [1, 1, 4]])

In[20]: # Создаем единичную матрицу размером 3 × 3
        np.eye(3)
Out[20]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])

In[21]: # Создаем неинициализированный массив из трех целочисленных
        # значений. Значениями будут произвольные данные, случайно
        # оказавшиеся в соответствующих ячейках памяти
        np.empty(3)

Out[21]: array([ 1.,  1.,  1.])
```

Стандартные типы данных NumPy

Массивы NumPy содержат значения простых типов, поэтому важно знать и понимать возможности и ограничения этих типов. Поскольку библиотека NumPy написана на языке C, эти типы будут знакомы пользователям языков C, Fortran и др.

Стандартные типы данных NumPy перечислены в табл. 4.1. Обратите внимание, что при создании массива их можно указывать в виде строк:

```
np.zeros(10, dtype='int16')
```

или использовать соответствующие объекты из библиотеки NumPy.

```
np.zeros(10, dtype=np.int16)
```

Таблица 4.1. Стандартные типы данных библиотеки NumPy

Тип данных	Описание
bool_	Булев тип (True или False), хранящийся как байт
int_	Тип целочисленного значения по умолчанию (аналогичен типу long в C; обычно int64 или int32)
intc	Идентичен типу int в C (обычно int32 или int64)
intp	Целочисленное значение, используемое для индексов (аналогично типу ssize_t в C; обычно int32 или int64)
int8	Байт (от –128 до 127)
int16	Целое число (от –32 768 до 32 767)
int32	Целое число (от –2 147 483 648 до 2 147 483 647)
int64	Целое число (от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807)
uint8	Целое число без знака (от 0 до 255)
uint16	Целое число без знака (от 0 до 65535)
uint32	Целое число без знака (от 0 до 4 294 967 295)
uint64	Целое число без знака (от 0 до 18 446 744 073 709 551 615)
float_	Сокращение для названия типа float64
float16	Число с плавающей точкой с половинной точностью: 1 бит знака, 5 битов порядка, 10 битов мантиссы
float32	Число с плавающей точкой с одинарной точностью: 1 бит знака, 8 битов порядка, 23 бита мантиссы
float64	Число с плавающей точкой с удвоенной точностью: 1 бит знака, 11 битов порядка, 52 бита мантиссы
complex_	Сокращение для названия типа complex128
complex64	Комплексное число, представленное двумя 32-битными числами с плавающей точкой
complex128	Комплексное число, представленное двумя 64-битными числами с плавающей точкой

ГЛАВА 5

Введение в массивы NumPy

Под работой с данными в Python практически всегда подразумевается работа с массивами NumPy: даже новейшие инструменты, такие как Pandas (часть III), основаны на массивах NumPy. В этой главе мы рассмотрим несколько примеров использования массивов NumPy для доступа к данным и подмассивам, а также получение срезов, изменение формы и объединение массивов. Хотя демонстрируемые типы операций могут показаться несколько скучными и обыденными, они являются своеобразными «кирпичиками» для множества других примеров в этой книге. Изучите их хорошенько!

Мы рассмотрим несколько категорий простейших операций с массивами:

- *получение атрибутов массивов* — определение размера, формы, объема занимаемой памяти и типов данных массивов;
- *индексация массивов* — получение и изменение значений отдельных элементов массивов;
- *получение срезов массивов* — получение и изменение значений подмассивов внутри большого массива;
- *изменение формы массивов* — изменение формы заданного массива;
- *объединение и разбиение массивов* — объединение нескольких массивов в один и разделение одного массива на несколько.

Атрибуты массивов NumPy

Обсудим некоторые атрибуты массивов. Начнем с описания трех массивов случайных чисел: одно-, двух- и трехмерного. Воспользуемся генератором случайных чисел в библиотеке NumPy, задав для него *начальное значение*, чтобы гарантировать получение одних и тех же массивов при каждом выполнении кода:

```
In[1]: import numpy as np
       np.random.seed(1701)  # начальное значение для воспроизводимости

       x1 = np.random.randint(10, size=6)          # одномерный массив
       x2 = np.random.randint(10, size=(3, 4))     # двумерный массив
       x3 = np.random.randint(10, size=(3, 4, 5))  # трехмерный массив
```

У каждого массива есть атрибуты `ndim` (число размерностей), `shape` (размер каждой размерности) и `size` (общий размер массива):

```
In [2]: print("x3 ndim: ", x3.ndim)
        print("x3 shape:", x3.shape)
        print("x3 size: ", x3.size)
        print("dtype:   ", x3.dtype)
Out[2]: x3 ndim:  3
        x3 shape: (3, 4, 5)
        x3 size:  60
        dtype:    int64
```

Более полное обсуждение типов данных вы найдете в главе 4.

Индексация массива: доступ к отдельным элементам

Если вы знакомы с индексацией стандартных списков в Python, то индексация массивов NumPy будет для вас привычной. В одномерном массиве обратиться к i-му (считая с 0) значению можно, указав требуемый индекс в квадратных скобках точно так же, как при работе с обычными списками:

```
In [3]: x1
Out[3]: array([9, 4, 0, 3, 8, 6])

In [4]: x1[0]
Out[4]: 9

In [5]: x1[4]
Out[5]: 8
```

Для индексации с конца массива можно использовать отрицательные индексы:

```
In [6]: x1[-1]
Out[6]: 6

In [7]: x1[-2]
Out[7]: 8
```

Обращаться к элементам в многомерном массиве можно с помощью кортежей индексов вида (*строка, столбец*):

```
In [8]: x2
Out[8]: array([[3, 1, 3, 7],
               [4, 0, 2, 3],
               [0, 0, 6, 9]])

In [9]: x2[0, 0]
Out[9]: 3

In [10]: x2[2, 0]
Out[10]: 0

In [11]: x2[2, -1]
Out[11]: 9
```

С использованием любой из перечисленных выше форм индексации также можно изменять значения:

```
In [12]: x2[0, 0] = 12
         x2
Out[12]: array([[12,  1,  3,  7],
                [ 4,  0,  2,  3],
                [ 0,  0,  6,  9]])
```

Не забывайте, что, в отличие от списков Python, тип данных массивов NumPy фиксирован. При вставке в массив целых чисел значения с плавающей точкой это значение будет незаметно усечено до целого. Не попадитесь в ловушку!

```
In [13]: x1[0] = 3.14159  # this will be truncated!
         x1
Out[13]: array([3, 4, 0, 3, 8, 6])
```

Срезы массивов: доступ к подмассивам

С помощью квадратных скобок можно определять не только отдельные элементы массива, но и целые подмассивы. Для этого NymPy поддерживает нотацию *срезов* (slice) — перечисление индексов границ среза через двоеточие (`:`). Синтаксис срезов в NumPy соответствует аналогичному синтаксису для стандартных списков в Python. Чтобы получить срез массива `x`, используйте синтаксис:

```
x[начало:конец:шаг]
```

Если какие-либо из этих значений не указаны, то применяются значения по умолчанию: `начало = 0, конец = размер соответствующего измерения, шаг = 1`. Давайте рассмотрим доступ к срезам одно- и многомерных массивов.

Одномерные подмассивы

Вот несколько примеров доступа к элементам одномерных подмассивов.

```
In [14]: x1
Out[14]: array([3, 4, 0, 3, 8, 6])

In [15]: x1[:3]  # первые три элемента
Out[15]: array([3, 4, 0])

In [16]: x1[3:]  # элементы после индекса 3
Out[16]: array([3, 8, 6])

In [17]: x1[1:4]  # подмассив из середины
Out[17]: array([4, 0, 3])

In [18]: x1[::2]  # каждый второй элемент
Out[18]: array([3, 0, 8])

In [19]: x1[1::2]  # каждый второй элемент, начиная с элемента с индексом 1
Out[19]: array([4, 3, 6])
```

Потенциально к небольшой путанице может привести отрицательное значение параметра `шаг`. В этом случае значения по умолчанию для `начало` и `конец` меняются местами. Это удобный способ «перевернуть» массив:

```
In [20]: x1[::-1]  # вернет все элементы в обратном порядке
Out[20]: array([6, 8, 3, 0, 4, 3])

In [21]: x1[4::-2]        # каждый второй элемент, начиная с индекса 4,
Out[21]: array([8, 0, 3]) # в обратном порядке
```

Многомерные подмассивы

Многомерные срезы создаются схожим образом: путем перечисления одномерных срезов через запятую. Например:

```
In [22]: x2
Out[22]: array([[12,  1,  3,  7],
                [ 4,  0,  2,  3],
                [ 0,  0,  6,  9]])

In [23]: x2[:2, :3]  # первые две строки и три столбца
Out[23]: array([[12,  1,  3],
                [ 4,  0,  2]])

In [24]: x2[:3, ::2]  # три строки, каждый второй столбец
Out[24]: array([[12,  3],
                [ 4,  2],
                [ 0,  6]])
```

```
In [25]: x2[::-1, ::-1]  # все строки и столбцы в обратном порядке
Out[25]: array([[ 9,  6,  0,  0],
               [ 3,  2,  0,  4],
               [ 7,  3,  1, 12]])
```

Одна из востребованных операций с массивами — получение одной строки или столбца. Эту операцию легко реализовать, указав индекс требуемой строки/столбца и пустой срез (одно лишь двоеточие):

```
In [26]: x2[:, 0]  # первый столбец из массива x2
Out[26]: array([12,  4,  0])

In [27]: x2[0, :]  # первая строка из массива x2
Out[27]: array([12,  1,  3,  7])
```

В случае доступа к одной строке пустой срез можно опустить и использовать более лаконичный синтаксис:

```
In [28]: x2[0]  # эквивалентно x2[0, :]
Out[28]: array([12,  1,  3,  7])
```

Подмассивы как представления

В отличие от срезов списков, срезы массивов NumPy возвращают *представления* (views), а не *копии* массивов. Рассмотрим уже знакомый нам двумерный массив:

```
In [29]: print(x2)
Out[29]: [[12  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

Извлечем из него двумерный подмассив 2 × 2:

```
In [30]: x2_sub = x2[:2, :2]
         print(x2_sub)
Out[30]: [[12  1]
          [ 4  0]]
```

Если теперь изменить этот подмассив, то исходный массив тоже изменится! Смотрите:

```
In [31]: x2_sub[0, 0] = 99
         print(x2_sub)
Out[31]: [[99  1]
          [ 4  0]]
```

```
In [32]: print(x2)
Out[32]: [[99  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

Кого-то такое поведение может удивить, но иногда оно оказывается весьма кстати, позволяя при работе с большими наборами данных обрабатывать их фрагменты без копирования в промежуточный буфер.

Создание копий массивов

Несмотря на все замечательные возможности представлений массивов, иногда бывает удобно явно скопировать данные из массива или подмассива. Это легко сделать с помощью метода `copy`:

```
In [33]: x2_sub_copy = x2[:2, :2].copy()
         print(x2_sub_copy)
Out[33]: [[99  1]
          [ 4  0]]
```

Теперь никакие изменения в этом подмассиве не отразятся на исходном массиве:

```
In [34]: x2_sub_copy[0, 0] = 42
         print(x2_sub_copy)
Out[34]: [[42  1]
          [ 4  0]]

In [35]: print(x2)
Out[35]: [[99  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

Изменение формы массивов

Еще одна полезная операция — изменение формы массивов, которую выполняет метод `reshape`. Например, если потребуется поместить числа от 1 до 9 в таблицу 3 × 3, то сделать это можно следующим образом:

```
In [36]: grid = np.arange(1, 10).reshape(3, 3)
         print(grid)
Out[36]: [[1 2 3]
          [4 5 6]
          [7 8 9]]
```

Обратите внимание, что размер исходного массива должен соответствовать размеру результирующего и в большинстве случаев метод `reshape` будет возвращать представления исходного массива.

Операция изменения формы часто используется для преобразования одномерного массива в двумерную матрицу-строку или матрицу-столбец:

```
In [37]: x = np.array([1, 2, 3])
         x.reshape((1, 3))  # преобразовать в вектор-строку с помощью reshape
Out[37]: array([[1, 2, 3]])

In [38]: x.reshape((3, 1))  # преобразовать в вектор-столбец с помощью reshape
Out[38]: array([[1],
                [2],
                [3]])
```

Настолько часто, что был реализован более короткий синтаксис `np.newaxis` с применением срезов:

```
In [39]: x[np.newaxis, :]  # преобразование в вектор-строку с помощью newaxis
Out[39]: array([[1, 2, 3]])

In [40]: x[:, np.newaxis]  # преобразование в вектор-столбец с помощью newaxis
Out[40]: array([[1],
                [2],
                [3]])
```

Далее в книге мы будем часто использовать этот паттерн.

Слияние и разбиение массивов

Все предыдущие операции работают с одним массивом. Но NumPy также предлагает операции для объединения нескольких массивов в один и разбиения одного массива на несколько подмассивов.

Слияние массивов

Слияние, или объединение, двух массивов в библиотеке NumPy выполняется в основном с помощью методов `np.concatenate`, `np.vstack` и `np.hstack`. Метод `np.concatenate` принимает на входе кортеж или список массивов в первом аргументе:

```
In [41]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])
Out[41]: array([1, 2, 3, 3, 2, 1])
```

Можно объединять более двух массивов одновременно:

```
In [42]: z = np.array([99, 99, 99])
         print(np.concatenate([x, y, z]))
Out[42]: [ 1  2  3  3  2  1 99 99 99]
```

Также можно объединять двумерные массивы:

```
In [43]: grid = np.array([[1, 2, 3],
                          [4, 5, 6]])

In [44]: # объединение по первой оси координат
         np.concatenate([grid, grid])
Out[44]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In [45]: # объединение по второй оси координат (нумерация начинается с 0)
         np.concatenate([grid, grid], axis=1)
Out[45]: array([[1, 2, 3, 1, 2, 3],
                [4, 5, 6, 4, 5, 6]])
```

Для работы с массивами с различным числом измерений удобнее и понятнее использовать функции `np.vstack` (вертикальное объединение) и `np.hstack` (горизонтальное объединение):

```
In [46]: # объединение массивов по вертикали
         np.vstack([x, grid])
Out[46]: array([[1, 2, 3],
                [1, 2, 3],
                [4, 5, 6]])

In [47]: # объединение массивов по горизонтали
         y = np.array([[99],
                       [99]])
         np.hstack([grid, y])
Out[47]: array([[ 1,  2,  3, 99],
                [ 4,  5,  6, 99]])
```

Функция `np.dstack` аналогично объединяет массивы по третьей оси.

Разбиение массивов

Противоположностью слияния является разбиение, выполняемое с помощью функций `np.split`, `np.hsplit` и `np.vsplit`. Каждой из них нужно передать список индексов, задающих точки разбиения:

```
In [48]: x = [1, 2, 3, 99, 99, 3, 2, 1]
         x1, x2, x3 = np.split(x, [3, 5])
         print(x1, x2, x3)
Out[48]: [1 2 3] [99 99] [3 2 1]
```

Обратите внимание, что N точек разбиения создают $N + 1$ подмассив. Соответствующие функции `np.hsplit` и `np.vsplit` действуют аналогично:

```
In [49]: grid = np.arange(16).reshape((4, 4))
         grid
Out[49]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

In [50]: upper, lower = np.vsplit(grid, [2])
         print(upper)
         print(lower)
Out[50]: [[0 1 2 3]
          [4 5 6 7]]
         [[ 8  9 10 11]
          [12 13 14 15]]

In [51]: left, right = np.hsplit(grid, [2])
         print(left)
         print(right)
Out[51]: [[ 0  1]
          [ 4  5]
          [ 8  9]
          [12 13]]
         [[ 2  3]
          [ 6  7]
          [10 11]
          [14 15]]
```

Функция `np.dsplit` аналогично разделяет массивы по третьей оси.

ГЛАВА 6

Вычисления с массивами NumPy: универсальные функции

К настоящему моменту мы обсудили некоторые основные возможности NumPy. В следующих нескольких главах подробнее рассмотрим причины, по которым NumPy так важна для обработки данных с помощью программ на Python, и одна из них — простой и гибкий интерфейс для оптимизированных вычислений с массивами данных.

Вычисления с массивами NumPy могут выполняться очень быстро или очень медленно. Ключ к ускорению — использование векторизованных операций, обычно реализуемых посредством *универсальных функций* (universal functions, ufuncs) в NumPy. Эта глава обоснует необходимость универсальных функций NumPy и объяснит, как они могут ускорить выполнение повторяющихся вычислений с элементами массивов, а также познакомит со множеством наиболее полезных универсальных арифметических функций в библиотеке NumPy.

Медлительность циклов

Реализация языка Python по умолчанию (известная под названием CPython) выполняет некоторые операции очень медленно. Частично это происходит из-за динамической, интерпретируемой природы языка. Гибкость типов не позволяет скомпилировать последовательности операций в столь же производительный машинный код, как в случае языков C и Fortran. В последнее время было предпринято несколько попыток решить эту проблему:

- проект PyPy (http://pypy.org/), реализация языка Python с динамической компиляцией;

- проект Cython (http://cython.org/), преобразующий код на языке Python в компилируемый код на языке C;
- проект Numba (http://numba.pydata.org/), преобразующий фрагменты кода на языке Python в быстрый LLVM-байткод.

У каждого проекта есть свои сильные и слабые стороны, но ни один из них пока не обошел стандартный механизм CPython по популярности.

Относительная медлительность Python обычно обнаруживается при многократном повторении мелких операций, например при обработке элементов массива в цикле. Пусть у нас имеется массив значений и для каждого нужно вычислить обратную величину. Очевидное решение выглядит следующим образом:

```
In [1]: import numpy as np
        rng = np.random.default_rng(seed=1701)

        def compute_reciprocals(values):
            output = np.empty(len(values))
            for i in range(len(values)):
                output[i] = 1.0 / values[i]
            return output

        values = rng.integers(1, 10, size=5)
        compute_reciprocals(values)
Out[1]: array([0.11111111, 0.25      , 1.        , 0.33333333, 0.125     ])
```

Такая реализация, вероятно, кажется вполне естественной разработчикам с опытом программирования на языках C или Java. Однако, оценив время выполнения этого кода на большом объеме данных, мы обнаружим, что он выполняется крайне медленно. Оценим это время с помощью «магической» функции `%timeit` оболочки IPython (обсуждавшейся в разделе «Профилирование и хронометраж выполнения кода» главы 3):

```
In [2]: big_array = rng.integers(1, 100, size=1000000)
        %timeit compute_reciprocals(big_array)
Out[2]: 2.61 s ± 192 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Выполнение миллиона операций и сохранение результата заняло несколько секунд! В наши дни, когда даже у смартфонов быстродействие измеряется в гигафлопсах (то есть в миллиардах операций с плавающей точкой в секунду), такая скорость кажется абсурдно низкой. Оказывается, проблема не в самих операциях, а в проверке типов и диспетчеризации функций, выполняемых CPython в каждой итерации цикла. Всякий раз, когда вычисляется обратная величина, Python сначала проверяет тип объекта и выполняет динамический поиск подходящей функции. Если бы мы работали с компилируемым кодом, то сведения о типе были бы известны до выполнения кода, а значит, результат вычислялся бы намного эффективнее.

Введение в универсальные функции

Библиотека NumPy предоставляет удобный интерфейс к компилируемым процедурам со статической типизацией для многих типов операций. Он известен под названием *векторизованной* операции. Для простых операций, таких как деление на значение элемента массива, под векторизованной операцией подразумевается простое применение операции к массиву в целом, которая затем будет применена к каждому из его элементов. Векторизованный подход спроектирован так, чтобы перенести цикл в скомпилированный слой, лежащий в основе библиотеки NumPy, что обеспечивает гораздо более высокую производительность.

Сравните результаты следующих двух операций:

```
In [3]: print(compute_reciprocals(values))
        print(1.0 / values)
Out[3]: [0.11111111 0.25       1.         0.33333333 0.125     ]
        [0.11111111 0.25       1.         0.33333333 0.125     ]
```

Выполнив хронометраж на большом массиве данных, можно заметить, что векторизованная операция выполняется на несколько порядков быстрее стандартного цикла Python:

```
In [4]: %timeit (1.0 / big_array)
Out[4]: 2.54 ms ± 383 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Векторизованные операции в библиотеке NumPy реализованы посредством универсальных функций (ufuncs), главная задача которых — быстрое выполнение повторяющихся операций над значениями из массивов NumPy. Универсальные функции исключительно гибкие. Выше была показана операция со скалярным значением и массивом, но операции также можно выполнять с двумя массивами:

```
In [5]: np.arange(5) / np.arange(1, 6)
Out[5]: array([0.        , 0.5       , 0.66666667, 0.75      , 0.8       ])
```

Универсальные функции не ограничиваются одномерными массивами и могут работать также с многомерными:

```
In [6]: x = np.arange(9).reshape((3, 3))
        2 ** x
Out[6]: array([[  1,   2,   4],
               [  8,  16,  32],
               [ 64, 128, 256]])
```

Векторизованные вычисления с применением универсальных функций практически всегда эффективнее их эквивалентов, реализованных с помощью циклов Python, особенно на больших массивах. Столкнувшись с подобным циклом в сценарии на языке Python, подумайте — возможно, стоит заменить его векторизованным выражением.

Обзор универсальных функций в библиотеке NumPy

Существуют два вида универсальных функций: *унарные универсальные функции*, принимающие один аргумент, и *бинарные*, с двумя аргументами. Мы рассмотрим примеры обоих типов функций.

Арифметические операции над массивами

Универсальные функции библиотеки NumPy очень просты в использовании, поскольку применяют обычные арифметические операторы языка Python. С их помощью можно выполнять обычные операции сложения, вычитания, умножения и деления:

```
In [7]: x = np.arange(4)
        print("x      =", x)
        print("x + 5  =", x + 5)
        print("x - 5  =", x - 5)
        print("x * 2  =", x * 2)
        print("x / 2  =", x / 2)
        print("x // 2 =", x // 2)  # деление с округлением вниз
Out[7]: x      = [0 1 2 3]
        x + 5  = [5 6 7 8]
        x - 5  = [-5 -4 -3 -2]
        x * 2  = [0 2 4 6]
        x / 2  = [0.  0.5 1.  1.5]
        x // 2 = [0 0 1 1]
```

Существует также унарная универсальная функция для операции изменения знака, оператор ** для возведения в степень и оператор % для деления по модулю:

```
In [8]: print("-x     = ", -x)
        print("x ** 2 = ", x ** 2)
        print("x % 2  = ", x % 2)
Out[8]: -x     =  [ 0 -1 -2 -3]
        x ** 2 =  [0 1 4 9]
        x % 2  =  [0 1 0 1]
```

Кроме того, эти операции можно комбинировать любыми способами с соблюдением стандартного приоритета выполнения операций:

```
In [9]: -(0.5*x + 1) ** 2
Out[9]: array([-1.  , -2.25, -4.  , -6.25])
```

Все эти арифметические операции — просто удобные обертки вокруг встроенных функций библиотеки NumPy. Например, оператор + является оберткой вокруг функции `add`:

```
In [10]: np.add(x, 2)
Out[10]: array([2, 3, 4, 5])
```

В табл. 6.1 перечислены реализованные в библиотеке NumPy арифметические операторы.

Таблица 6.1. Реализованные в библиотеке NumPy арифметические операторы

Оператор	Эквивалентная универсальная функция	Описание
`+`	`np.add`	Сложение (например, `1 + 1 = 2`)
`-`	`np.subtract`	Вычитание (например, `3 - 2 = 1`)
`-`	`np.negative`	Унарная операция изменения знака (например, `-2`)
`*`	`np.multiply`	Умножение (например, `2 * 3 = 6`)
`/`	`np.divide`	Деление (например, `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Деление с округлением в меньшую сторону (например, `3 // 2 = 1`)
`**`	`np.power`	Возведение в степень (например, `2 ** 3 = 8`)
`%`	`np.mod`	Модуль/остаток (например, `9 % 4 = 1`)

Помимо этого, существуют еще логические/побитовые операции, которые мы рассмотрим в главе 9.

Абсолютное значение

Библиотека NumPy подменяет не только встроенные арифметические операторы, но также встроенную функцию получения абсолютного значения:

```
In [11]: x = np.array([-2, -1, 0, 1, 2])
         abs(x)
Out[11]: array([2, 1, 0, 1, 2])
```

Соответствующая универсальная функция библиотеки NumPy — `np.absolute`, доступна также под псевдонимом `np.abs`:

```
In [12]: np.absolute(x)
Out[12]: array([2, 1, 0, 1, 2])

In [13]: np.abs(x)
Out[13]: array([2, 1, 0, 1, 2])
```

Эта универсальная функция может также обрабатывать комплексные значения, возвращая их модуль:

```
In [14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
         np.abs(x)
Out[14]: array([5., 5., 2., 1.])
```

Тригонометрические функции

Библиотека NumPy предоставляет множество универсальных функций, в том числе такие важные для специалистов по обработке данных, как тригонометрические функции. Начнем с описания массива углов:

```
In [15]: theta = np.linspace(0, np.pi, 3)
```

Теперь мы можем вычислить некоторые тригонометрические функции от этих значений:

```
In [16]: print("theta      = ", theta)
         print("sin(theta) = ", np.sin(theta))
         print("cos(theta) = ", np.cos(theta))
         print("tan(theta) = ", np.tan(theta))
Out[16]: theta      =  [0.         1.57079633 3.14159265]
         sin(theta) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
         cos(theta) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
         tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

Значения вычислены в пределах машинной точности, поэтому те из них, которые должны быть нулевыми, не всегда в точности равны нулю. Доступны для использования также обратные тригонометрические функции:

```
In [17]: x = [-1, 0, 1]
         print("x         = ", x)
         print("arcsin(x) = ", np.arcsin(x))
         print("arccos(x) = ", np.arccos(x))
         print("arctan(x) = ", np.arctan(x))
Out[17]: x         =  [-1, 0, 1]
         arcsin(x) =  [-1.57079633  0.          1.57079633]
         arccos(x) =  [3.14159265 1.57079633 0.        ]
         arctan(x) =  [-0.78539816  0.          0.78539816]
```

Показательные функции и логарифмы

Показательные функции — еще один распространенный тип операций, доступный в библиотеке NumPy:

```
In [18]: x = [1, 2, 3]
         print("x   =", x)
         print("e^x =", np.exp(x))
         print("2^x =", np.exp2(x))
         print("3^x =", np.power(3., x))
Out[18]: x   = [1, 2, 3]
         e^x = [ 2.71828183  7.3890561  20.08553692]
         2^x = [2. 4. 8.]
         3^x = [ 3.  9. 27.]
```

Функции, обратные к показательным, — логарифмы — тоже имеются в библиотеке. Простейшая функция `np.log` возвращает натуральный логарифм числа. Если вам требуется логарифм по основанию 2 или 10, они также доступны:

```
In [19]: x = [1, 2, 4, 10]
         print("x        =", x)
         print("ln(x)    =", np.log(x))
         print("log2(x)  =", np.log2(x))
         print("log10(x) =", np.log10(x))
Out[19]: x        = [1, 2, 4, 10]
         ln(x)    = [0.         0.69314718 1.38629436 2.30258509]
         log2(x)  = [0.         1.         2.         3.32192809]
         log10(x) = [0.         0.30103    0.60205999 1.        ]
```

Имеются некоторые специальные версии функций, удобные для сохранения точности при очень малых входных значениях:

```
In [20]: x = [0, 0.001, 0.01, 0.1]
         print("exp(x) - 1 =", np.expm1(x))
         print("log(1 + x) =", np.log1p(x))
Out[20]: exp(x) - 1 = [0.         0.0010005  0.01005017 0.10517092]
         log(1 + x) = [0.         0.0009995  0.00995033 0.09531018]
```

При очень малых значениях элементов вектора `x` данные функции возвращают намного более точные результаты, чем обычные функции `np.log` и `np.exp`.

Специализированные универсальные функции

В библиотеке NumPy имеется немало других универсальных функций, включая гиперболические тригонометрические функции, функции поразрядной арифметики, сравнения, преобразования из радианов в градусы, округления и получения остатков от деления, а также многие другие. Если заглянуть в документацию по библиотеке NumPy, то можно обнаружить немало интересных функций.

Еще один замечательный источник специализированных и сложных универсальных функций — подмодуль `scipy.special`. Если вам необходимо вычислить значение какой-то хитрой математической функции на ваших данных, то высока вероятность, что она уже реализована в `scipy.special`. Следующий пример демонстрирует несколько функций, которые могут пригодиться для статистических вычислений:

```
In [21]: from scipy import special
In [22]: # Гамма-функции (обобщенные факториалы) и тому подобные функции
         x = [1, 5, 10]
         print("gamma(x)     =", special.gamma(x))
```

```
        print("ln|gamma(x)| =", special.gammaln(x))
        print("beta(x, 2)   =", special.beta(x, 2))
Out[22]: gamma(x)     = [1.0000e+00 2.4000e+01 3.6288e+05]
        ln|gamma(x)| = [ 0.          3.17805383 12.80182748]
        beta(x, 2)   = [0.5        0.03333333 0.00909091]

In [23]: # Функция ошибок (интеграл от гауссовой функции),
        # дополнительная и обратная к ней функции
        x = np.array([0, 0.3, 0.7, 1.0])
        print("erf(x)  =", special.erf(x))
        print("erfc(x) =", special.erfc(x))
        print("erfinv(x) =", special.erfinv(x))
Out[23]: erf(x)  = [0.         0.32862676 0.67780119 0.84270079]
        erfc(x) = [1.         0.67137324 0.32219881 0.15729921]
        erfinv(x) = [0.         0.27246271 0.73286908        inf]
```

В библиотеках NumPy и `scipy.special` имеется много универсальных функций. Документация по ним доступна в Интернете. Простой поиск по фразе «гамма-функции в Python» почти наверняка выдаст вам соответствующую информацию.

Продвинутые возможности универсальных функций

Многие пользователи пакета NumPy работают с универсальными функциями, даже не подозревая обо всех их возможностях. Мы вкратце рассмотрим некоторые специализированные возможности универсальных функций.

Сохранение результатов в массиве

При выполнении операций с большими объемами данных удобно задать массив, куда должны сохраняться результаты вычисления. Вместо создания временного массива можно воспользоваться этой возможностью для записи результатов вычислений непосредственно в нужное место в памяти. Сделать это для любой универсальной функции можно с помощью аргумента `out`:

```
In [24]: x = np.arange(5)
        y = np.empty(5)
        np.multiply(x, 10, out=y)
        print(y)
Out[24]: [ 0. 10. 20. 30. 40.]
```

Эту возможность можно использовать даже вместе с представлениями массивов. Например, можно записать результаты вычислений в каждый второй элемент заданного массива:

```
In [25]: y = np.zeros(10)
         np.power(2, x, out=y[::2])
         print(y)
Out[25]: [ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

Если бы мы вместо этого написали `y[::2] = 2 ** x`, то был бы создан временный массив для хранения результатов операции `2 ** x` с последующим их копированием в массив `y`. Для столь незначительных объемов вычислений особой разницы нет, но для очень больших массивов экономия памяти за счет использования аргумента `out` может оказаться значительной.

Сводные показатели

Бинарные универсальные функции предлагают возможность вычислять некоторые сводные данные непосредственно на основе объекта. Например, если понадобится *свернуть* массив с помощью конкретной операции, то можно воспользоваться методом `reduce` соответствующей универсальной функции. Операция `reduce` многократно применяет заданную операцию к элементам массива до тех пор, пока не останется только один результат.

Например, вызов метода `reduce` для универсальной функции `add` возвращает сумму всех элементов массива:

```
In [26]: x = np.arange(1, 6)
         np.add.reduce(x)
Out[26]: 15
```

Аналогично вызов метода `reduce` для универсальной функции `multiply` вернет произведение всех элементов массива:

```
In [27]: np.multiply.reduce(x)
Out[27]: 120
```

Если, напротив, нужно сохранить все промежуточные результаты вычислений, то можно вместо `reduce` воспользоваться функцией `accumulate`:

```
In [28]: np.add.accumulate(x)
Out[28]: array([ 1,  3,  6, 10, 15])

In [29]: np.multiply.accumulate(x)
Out[29]: array([  1,   2,   6,  24, 120])
```

Обратите внимание, что в данных конкретных случаях для вычисления этих значений в NumPy существуют специализированные функции (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`), которые мы рассмотрим в главе 7.

Векторные произведения

Все универсальные функции могут выводить результат применения соответствующей операции ко всем парам двух аргументов с помощью метода `outer`. Это дает возможность одной строкой кода создать, например, таблицу умножения:

```
In [30]: x = np.arange(1, 6)
         np.multiply.outer(x, x)
Out[30]: array([[ 1,  2,  3,  4,  5],
                [ 2,  4,  6,  8, 10],
                [ 3,  6,  9, 12, 15],
                [ 4,  8, 12, 16, 20],
                [ 5, 10, 15, 20, 25]])
```

Универсальные функции дают возможность работать с массивами различных размеров и форм, используя набор операций под названием *транслирование* (broadcasting). Эта тема достаточно важна, поэтому ей будет посвящена целая глава (см. главу 8).

Универсальные функции: дополнительная информация

Дополнительную информацию об универсальных функциях (включая полный список имеющихся функций) можно найти на сайтах проектов NumPy (http://www.numpy.org/) и SciPy (http://www.scipy.org/).

Получить доступ к этой информации можно непосредственно из оболочки IPython, импортировав эти пакеты и использовав функцию автодополнения табуляцией (клавиша Tab) и получения справки (`?`), как описано в главе 1.

ГЛАВА 7

Агрегирование: минимум, максимум и все, что посередине

Очень часто при работе с большими объемами данных первый шаг заключается в вычислении сводных статистических показателей по этим данным. Среднее значение и стандартное отклонение, позволяющие выявить «типичные» значения в наборе данных, — наиболее распространенные сводные статистические показатели, но и другие сводные показатели тоже полезны (сумма, произведение, медиана, минимум и максимум, квантили и т. д.).

В библиотеке NumPy имеются быстрые функции агрегирования для работы с массивами. Давайте обсудим и опробуем некоторые из них.

Суммирование значений в массиве

В качестве примера рассмотрим вычисление суммы значений в массиве. На «чистом» Python это можно реализовать с помощью встроенной функции `sum`:

```
In [1]: import numpy as np
        rng = np.random.default_rng()

In [2]: L = rng.random(100)
        sum(L)
Out[2]: 52.76825337322368
```

Ее синтаксис очень похож на синтаксис функции `sum` из библиотеки NumPy, и результат в простейшем случае тот же:

```
In [3]: np.sum(L)
Out[3]: 52.76825337322366
```

Однако, поскольку функция `sum` выполняет операцию в скомпилированном коде, версия из библиотеки NumPy работает намного быстрее:

```
In [4]: big_array = rng.random(1000000)
        %timeit sum(big_array)
        %timeit np.sum(big_array)
Out[4]: 89.9 ms ± 233 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
        521 µs ± 8.37 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Будьте осторожны: функции `sum` и `np.sum` не идентичны! Например, смысл их необязательных аргументов различен. Вызов `sum(x, 1)` инициализирует сумму значением `1`, тогда как вызов `np.sum(x, 1)` выполнит суммирование по оси `1`. И да, функция `np.sum` умеет работать с многомерными массивами, как вы увидите в следующем разделе.

Минимум и максимум

В «чистом» Python имеются встроенные функции `min` и `max`, используемые для поиска минимального и максимального значений в любом данном массиве:

```
In [5]: min(big_array), max(big_array)
Out[5]: (2.0114398036064074e-07, 0.9999997912802653)
```

В NumPy имеются функции с аналогичным синтаксисом, действующие намного быстрее:

```
In [6]: np.min(big_array), np.max(big_array)
Out[6]: (2.0114398036064074e-07, 0.9999997912802653)
In [7]: %timeit min(big_array)
        %timeit np.min(big_array)
Out[7]: 72 ms ± 177 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
        564 µs ± 3.11 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

В NumPy поддерживаются операции, аналогичные `min`, `max`, `sum` и некоторым другим агрегирующим операциям, реализованные как методы самого объекта массива:

```
In [8]: print(big_array.min(), big_array.max(), big_array.sum())
Out[8]: 2.0114398036064074e-07 0.9999997912802653 499854.0273321711
```

При работе с массивами NumPy обязательно проверяйте, используете ли вы NumPy-версии функций для вычисления сводных показателей!

Многомерные сводные показатели

Агрегирование по столбцу или строке — один из часто применяемых видов операций агрегирования. Пусть имеются какие-либо данные, находящиеся в двумерном массиве:

```
In [9]: M = rng.integers(0, 10, (3, 4))
        print(M)
Out[9]: [[0 3 1 2]
         [1 9 7 0]
         [4 8 3 7]]
```

По умолчанию все функции агрегирования в NumPy возвращают сводный показатель по всему массиву:

```
In [10]: M.sum()
Out[10]: 45
```

Но они принимают дополнительный аргумент, позволяющий указать ось, по которой вычисляется сводный показатель. Например, можно найти минимальное значение каждого из столбцов, указав `axis=0`:

```
In [11]: M.min(axis=0)
Out[11]: array([0, 3, 1, 0])
```

Функция возвращает четыре значения, соответствующие четырем столбцам чисел.

Аналогично можно вычислить максимальное значение в каждой из строк:

```
In [12]: M.max(axis=1)
Out[12]: array([3, 9, 8])
```

Способ задания оси в этих примерах может вызвать затруднения у пользователей, работавших ранее с другими языками программирования. Именованный аргумент `axis` задает измерение массива, которое будет *свернуто*, а не возвращаемое измерение. Поэтому аргумент `axis=0` означает, что свернута будет ось 0 (первая в массиве): в двумерном массиве будут агрегированы значения в каждом из столбцов.

Другие функции агрегирования

Библиотека NumPy предоставляет много других агрегирующих функций. Большинство имеет `NaN`-безопасный эквивалент, вычисляющий результат с игнорированием отсутствующих значений, отмеченных специальным значением с плавающей точкой `NaN`, определенным организацией IEEE (см. главу 16).

В табл. 7.1 приводится список полезных агрегирующих функций, доступных в NumPy.

Таблица 7.1. Доступные в библиотеке NumPy функции агрегирования

Имя функции	NaN-безопасная версия	Описание
`np.sum`	`np.nansum`	Вычисляет сумму элементов
`np.prod`	`np.nanprod`	Вычисляет произведение элементов
`np.mean`	`np.nanmean`	Вычисляет среднее значение элементов
`np.std`	`np.nanstd`	Вычисляет стандартное отклонение
`np.var`	`np.nanvar`	Вычисляет дисперсию
`np.min`	`np.nanmin`	Вычисляет минимальное значение
`np.max`	`np.nanmax`	Вычисляет максимальное значение
`np.argmin`	`np.nanargmin`	Возвращает индекс минимального значения
`np.argmax`	`np.nanargmax`	Возвращает индекс максимального значения
`np.median`	`np.nanmedian`	Вычисляет медиану элементов
`np.percentile`	`np.nanpercentile`	Вычисляет квантили элементов
`np.any`	—	Проверяет, существуют ли элементы со значением `True`
`np.all`	—	Проверяет, все ли элементы имеют значение `True`

Мы часто будем встречаться с этими агрегирующими функциями в дальнейшем.

Пример: чему равен средний рост президентов США

Имеющиеся в библиотеке NumPy функции агрегирования могут очень пригодиться для обобщения набора значений. В качестве простого примера рассмотрим рост всех президентов США. Эти данные доступны в файле `president_heights.csv` — простом списке меток и значений, разделенных запятыми:

```
In [13]: !head -4 data/president_heights.csv
Out[13]: order,name,height(cm)
         1,George Washington,189
         2,John Adams,170
         3,Thomas Jefferson,189
```

Для чтения файла и извлечения данной информации (обратите внимание, что рост указан в сантиметрах) мы воспользуемся пакетом Pandas, который изучим более детально в части III:

```
In [14]: import pandas as pd
         data = pd.read_csv('data/president_heights.csv')
         heights = np.array(data['height(cm)'])
         print(heights)
Out[14]: [189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173
          174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
          177 185 188 188 182 185 191 182]
```

Теперь, получив массив данных, можно вычислить множество сводных статистических показателей:

```
In [15]: print("Mean height:       ", heights.mean())
         print("Standard deviation:", heights.std())
         print("Minimum height:    ", heights.min())
         print("Maximum height:    ", heights.max())
Out[15]: Mean height:        180.04545454545453
         Standard deviation: 6.983599441335736
         Minimum height:     163
         Maximum height:     193
```

Обратите внимание, что в каждом случае операция агрегирования применяется ко всему массиву и возвращает единственное итоговое значение, дающее информацию о распределении значений. Возможно, вам также захочется вычислить квантили:

```
In [16]: print("25th percentile:   ", np.percentile(heights, 25))
         print("Median:            ", np.median(heights))
         print("75th percentile:   ", np.percentile(heights, 75))
Out[16]: 25th percentile:    174.75
         Median:             182.0
         75th percentile:    183.5
```

Как видите, медиана роста президентов США составляет 182 см, то есть чуть-чуть не дотягивает до шести футов.

Иногда полезнее видеть графическое представление подобных данных, которое можно получить с помощью инструментов из пакета Matplotlib (мы подробно рассмотрим Matplotlib в части IV). Например, следующий код генерирует график, показанный на рис. 7.1:

```
In [17]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')
```

```
In [18]: plt.hist(heights)
         plt.title('Height Distribution of US Presidents') # Распределение роста
                                                           # президентов США
         plt.xlabel('height (cm)')                         # Рост (см)
         plt.ylabel('number');                             # Количество
```

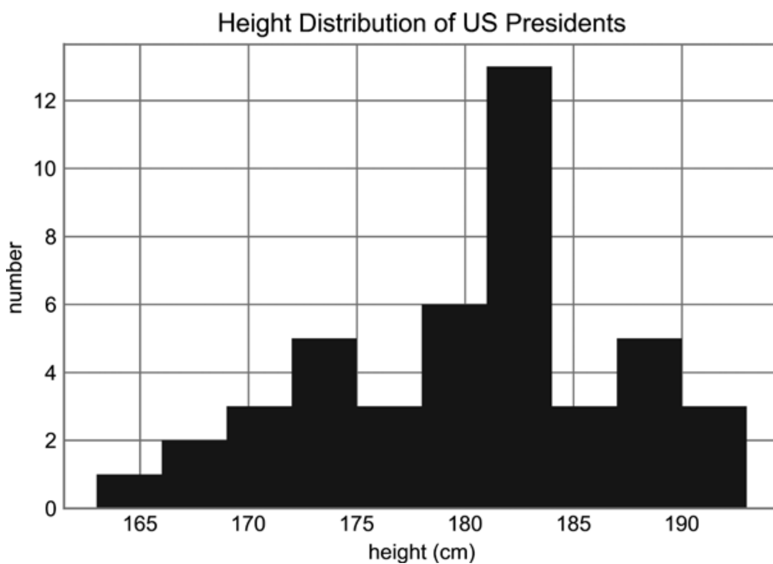


Рис. 7.1. Гистограмма роста президентов

ГЛАВА 8

Операции над массивами. Транслирование

В главе 6 мы видели, как можно использовать универсальные функции из библиотеки NumPy для *векторизации* операций и замены медленных стандартных циклов Python. В этой главе мы обсудим *транслирование* (broadcasting) — набор правил, которые позволяют применять функции из NumPy для выполнения бинарных операций (таких как сложение, вычитание, умножение и т. д.) с массивами разных форм и размеров.

Введение в транслирование

Для массивов одинакового размера бинарные операции выполняются поэлементно:

```
In [1]: import numpy as np

In [2]: a = np.array([0, 1, 2])
        b = np.array([5, 5, 5])
        a + b
Out[2]: array([5, 6, 7])
```

Транслирование дает возможность выполнять подобные виды бинарных операций с массивами различных размеров, например, можно легко прибавить скалярное значение (рассматривая его как нульмерный массив) к массиву:

```
In [3]: a + 5
Out[3]: array([5, 6, 7])
```

Можно рассматривать транслирование как операцию, превращающую путем растягивания (или дублирования) значение `5` в массив `[5, 5, 5]`, после чего складывающую полученный результат с массивом `a`.

Аналогично можно распространить транслирование на массивы большей размерности. Посмотрите на результат сложения одномерного и двумерного массивов:

```
In [4]: M = np.ones((3, 3))
        M
Out[4]: array([[1., 1., 1.],
               [1., 1., 1.],
               [1., 1., 1.]])

In [5]: M + a
Out[5]: array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.]])
```

Здесь одномерный массив `a` растягивается (транслируется) на второе измерение, чтобы соответствовать форме массива `M`.

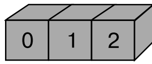
Эти примеры просты и понятны. Более сложные случаи могут включать транслирование обоих массивов. Рассмотрим следующий пример:

```
In [6]: a = np.arange(3)
        b = np.arange(3)[:, np.newaxis]

        print(a)
        print(b)
Out[6]: [0 1 2]
        [[0]
         [1]
         [2]]

In [7]: a + b
Out[7]: array([[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
```

Аналогично тому, как мы раньше растягивали (транслировали) один массив, чтобы он соответствовал форме другого, здесь мы растягиваем *оба* массива `a` и `b`, чтобы привести их к общей форме. В результате мы получаем двумерный массив! Геометрическое представление этих примеров наглядно показано на рис. 8.1.

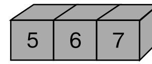
Полупрозрачные кубики представляют транслируемые значения: соответствующая дополнительная память фактически не выделяется в ходе операции, но мысленно удобно представлять себе, что именно так все и происходит.

np.arange(3)+5

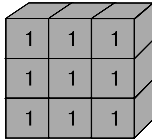


+

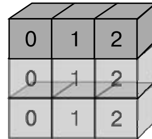
=



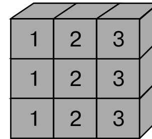
np.ones((3,3))+np.arrange(3)



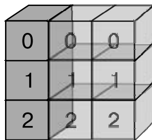
+



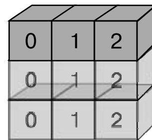
=



np.arrange(3).reshape((3,1))+np.arrange(3)



+



=

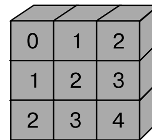


Рис. 8.1. Визуализация транслирования массивов библиотекой NumPy

Правила транслирования

Транслирование в библиотеке NumPy следует строгому набору правил, определяющему взаимодействие двух массивов.

- *Правило 1*: если размерности двух массивов отличаются, то форма массива с меньшей размерностью дополняется единицами с ведущей (левой) стороны.
- *Правило 2*: если формы двух массивов не совпадают в каком-то измерении, то массив с формой, равной 1 в данном измерении, растягивается вплоть до соответствия форме другого массива.
- *Правило 3*: если в каком-либо измерении размеры массивов различаются и ни один не равен 1, то генерируется ошибка.

Для разъяснения этих правил рассмотрим несколько примеров.

Транслирование. Пример 1

Рассмотрим сложение двумерного массива с одномерным:

```
In[8]: M = np.ones((2, 3))
       a = np.arange(3)
```

Разберем эту операцию подробнее. Массивы имеют следующие формы:

- `M.shape = (2, 3)`;
- `a.shape = (3,)`.

Согласно правилу 1, поскольку размерность массива `a` меньше, мы добавляем недостающее измерение слева:

- `M.shape -> (2, 3)`;
- `a.shape -> (1, 3)`.

Согласно правилу 2, поскольку первые измерения массивов различаются, мы растягиваем первое измерение в `a` до размеров первого измерения в `M`:

- `M.shape -> (2, 3)`;
- `a.shape -> (2, 3)`.

Формы совпадают, и мы видим, что итоговая форма будет равна `(2, 3)`:

```
In [9]: M + a
Out[9]: array([[1., 2., 3.],
               [1., 2., 3.]])
```

Транслирование. Пример 2

Рассмотрим пример, когда необходимо транслировать оба массива:

```
In[10]: a = np.arange(3).reshape((3, 1))
        b = np.arange(3)
```

Для начала определим формы наших массивов:

- `a.shape = (3, 1)`;
- `b.shape = (3,)`.

Правило 1 гласит, что мы должны дополнить форму массива `b` еще одним измерением:

- `a.shape -> (3, 1)`;
- `b.shape -> (1, 3)`.

Правило 2 гласит, что нужно дополнить это измерение единицами вплоть до совпадения с размером данного измерения в другом массиве:

- `a.shape -> (3, 3)`;
- `b.shape -> (3, 3)`.

Поскольку результаты совпадают, формы совместимы. Получаем:

```
In[11]: a + b
Out[11]: array([[0, 1, 2],
                [1, 2, 3],
                [2, 3, 4]])
```

Транслирование. Пример 3

Рассмотрим пример, в котором два массива несовместимы:

```
In [12]: M = np.ones((3, 2))
         a = np.arange(3)
```

Эта ситуация лишь немного отличается от примера 1: матрица `M` транспонирована. Какое влияние этот факт окажет на вычисления? Массивы имеют следующие формы:

- `M.shape = (3, 2)`;
- `a.shape = (3,)`.

Правило 1 требует дополнить форму массива `a`:

- `M.shape -> (3, 2)`;
- `a.shape -> (1, 3)`.

Согласно правилу 2 первое измерение массива `a` растягивается, чтобы соответствовать первому измерению в массиве `M`:

- `M.shape -> (3, 2)`;
- `a.shape -> (3, 3)`.

Теперь вступает в действие правило 3 — итоговые формы не совпадают, поэтому массивы считаются несовместимыми, что мы и видим, попытавшись выполнить данную операцию:

```
In [13]: M + a
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Обратите внимание на имеющийся потенциальный источник ошибки: можно было бы сделать массивы `a` и `M` совместимыми, скажем, путем дополнения формы `a` единицами справа, а не слева. Но правила транслирования так не работают! Если вам захочется дополнить форму массива справа, то вы должны сделать это явно, поменяв форму массива (мы воспользуемся ключевым словом `np.newaxis`, описанным в главе 5):

```
In [14]: a[:, np.newaxis].shape
Out[14]: (3, 1)
In [15]: M + a[:, np.newaxis]
```

```
Out[15]: array([[1., 1.],
               [2., 2.],
               [3., 3.]])
```

Хотя мы сосредоточили все свое внимание на операторе `+`, данные правила транслирования применимы ко *всем* бинарным универсальным функциям. Например, рассмотрим функцию `logaddexp(a, b)`, вычисляющую `log(exp(a) + exp(b))` с большей точностью, чем при стандартном подходе:

```
In [16]: np.logaddexp(M, a[:, np.newaxis])
Out[16]: array([[1.31326169, 1.31326169],
               [1.69314718, 1.69314718],
               [2.31326169, 2.31326169]])
```

За дополнительной информацией по множеству доступных универсальных функций обращайтесь к главе 6.

Транслирование на практике

Операции транслирования — основное ядро множества примеров, приводимых в дальнейшем в нашей книге. Рассмотрим несколько простых примеров сфер их возможного применения.

Центрирование массива

Универсальные функции из библиотеки NumPy избавляют от необходимости использовать медленные циклы Python. Транслирование расширяет эти возможности. Один из часто встречающихся примеров в науке о данных — центрирование массива. Пусть есть массив из десяти наблюдений, каждое состоит из трех значений. Используя стандартные соглашения, сохраним эти данные в массиве 10×3:

```
In [17]: rng = np.random.default_rng(seed=1701)
         X = rng.random((10, 3))
```

Вычислить среднее значение каждого признака можно, применив функцию агрегирования `mean` по первому измерению:

```
In [18]: Xmean = X.mean(0)
         Xmean
Out[18]: array([0.38503638, 0.36991443, 0.63896043])
```

Теперь можно центрировать массив `X` путем вычитания среднего значения (это операция транслирования):

```
In[19]: X_centered = X - Xmean
```

Чтобы убедиться в правильности выполнения данной операции, вычислим средние значения признаков в центрированном массиве, которые должны быть близки к 0:

```
In [20]: X_centered.mean(0)
Out[20]: array([ 4.99600361e-17, -4.44089210e-17,  0.00000000e+00])
```

В пределах машинной точности средние значения действительно равны 0.

Построение графика двумерной функции

Одна из частых сфер применения транслирования — визуализация двумерных функций. Чтобы описать функцию $z = f(x, y)$, можно воспользоваться транслированием для вычисления значений этой функции на координатной сетке:

```
In[21]: # Задаем для x и y 50 шагов от 0 до 5
        x = np.linspace(0, 5, 50)
        y = np.linspace(0, 5, 50)[:, np.newaxis]

        z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

Воспользуемся библиотекой Matplotlib для построения двумерного графика (мы обсудим эти инструменты подробно в главе 28):

```
In [22]: %matplotlib inline
         import matplotlib.pyplot as plt

In [23]: plt.imshow(z, origin='lower', extent=[0, 5, 0, 5])
         plt.colorbar();
```

На рис. 8.2 показана великолепная визуализация двумерной функции.

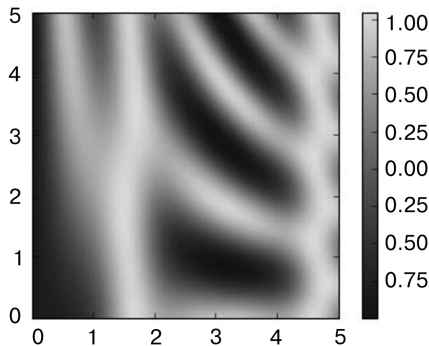


Рис. 8.2. Визуализация двумерного массива

ГЛАВА 9

Сравнения, маски и булева логика

Эта глава охватывает использование булевых масок для проверки и изменения значений в массивах NumPy. Маскирование удобно для извлечения, модификации, подсчета или других манипуляций со значениями в массиве по какому-либо критерию. Например, вам может понадобиться подсчитать все значения, превышающие определенное число, или, возможно, удалить все аномальные значения, превышающие какую-либо пороговую величину. В библиотеке NumPy булевы маски зачастую самый эффективный способ решения подобных задач.

Пример: подсчет количества дождливых дней

Пусть у вас есть последовательности данных, отражающих количество осадков в каждый день года для конкретного города. Например, с помощью библиотеки Pandas (рассматриваемой подробнее в части III) мы загрузили ежедневную статистику по осадкам для Сиэтла за 2015 год:

```
In [1]: import numpy as np
        from vega_datasets import data

        # Используем Pandas для извлечения количества осадков в NumPy-массив
        rainfall_mm = np.array(
            data.seattle_weather().set_index('date')['precipitation']['2015'])
        len(rainfall_mm)
Out[1]: 365
```

Этот массив содержит 365 значений, соответствующих ежедневной величине осадков в миллиметрах, с 1 января по 31 декабря 2015 года.

В качестве первой простейшей визуализации рассмотрим гистограмму дождливых дней, показанную на рис. 9.1, сгенерированную с помощью библиотеки Matplotlib (подробнее этот инструмент мы рассмотрим в части IV):

```
In [2]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')

In [3]: plt.hist(rainfall_mm, 40);
```

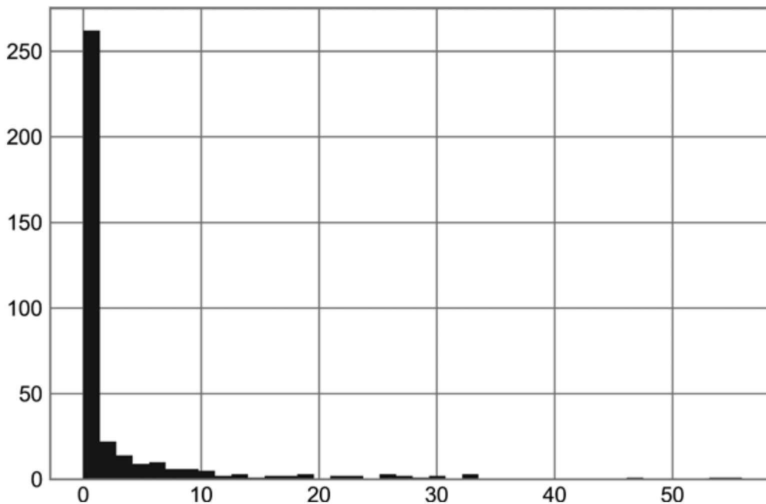


Рис. 9.1. Гистограмма осадков в 2015 году в Сиэтле

Эта гистограмма дает общее представление о том, что такое наши данные: несмотря на репутацию дождливого города, измеренное количество осадков в Сиэтле в абсолютное большинство дней в 2015 году близко к нулю. Но она плохо отражает нужную информацию: например, сколько было дождливых дней? Каково было среднее количество осадков в эти дождливые дни? Сколько было дней с более чем 10 мм осадков?

Один из возможных подходов — ответить на эти вопросы «вручную»: организовать цикл по данным, увеличивая счетчик на единицу всякий раз при попадании на находящиеся в нужном диапазоне значения. По уже обсуждавшимся в этой главе причинам такой подход окажется очень неэффективным с точки зрения как времени, затраченного на написание кода, так и времени на вычисление результата. Как мы видели в главе 6, той же цели можно достичь, воспользовавшись универсальными функциями из библиотеки NumPy для выполнения быстрых поэлементных

арифметических операций с массивами. В то же время можно использовать другие универсальные функции для поэлементных *сравнений* в массивах и по их результатам получить ответы на интересующие нас вопросы. Мы ненадолго отложим наши данные в сторону и обсудим некоторые общие инструменты библиотеки NumPy, позволяющие применять *маскирование* (masking) для быстрого ответа на подобные вопросы.

Операторы сравнения как универсальные функции

В главе 6 вы познакомились с универсальными функциями, сосредоточив внимание на арифметических операторах. Вы увидели, что применение операторов `+`, `-`, `*`, `/` и других для операций с массивами приводит к выполнению поэлементных операций. В библиотеке NumPy также реализованы операторы сравнения, такие как `<` («меньше») и `>` («больше»), в виде поэлементных универсальных функций. Результатом этих операторов сравнения всегда является массив с данными булева типа. Для использования доступны все шесть стандартных операторов сравнения:

```
In [4]: x = np.array([1, 2, 3, 4, 5])

In [5]: x < 3  # меньше
Out[5]: array([ True,  True, False, False, False])

In [6]: x > 3  # больше
Out[6]: array([False, False, False,  True,  True])

In [7]: x <= 3  # меньше или равно
Out[7]: array([ True,  True,  True, False, False])

In [8]: x >= 3  # больше или равно
Out[8]: array([False, False,  True,  True,  True])

In [9]: x != 3  # не равно
Out[9]: array([ True,  True, False,  True,  True])

In [10]: x == 3  # равно
Out[10]: array([False, False,  True, False, False])
```

Можно также выполнять поэлементное сравнение двух массивов и использовать составные выражения:

```
In [11]: (2 * x) == (x ** 2)
Out[11]: array([False,  True, False, False, False])
```

Подобно арифметическим операторам, операторы сравнения реализованы в библиотеке NumPy как универсальные функции (табл. 9.1). Например, когда вы пишете `x < 3`, библиотека NumPy на самом деле вызывает функцию `np.less(x, 3)`.

Таблица 9.1. Краткий список операторов сравнения и эквивалентных им универсальных функций

Оператор	Эквивалентная универсальная функция
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

Подобно арифметическим универсальным функциям, они могут работать с массивами любых размеров и форм. Вот пример для двумерного массива:

```
In [12]: rng = np.random.default_rng(seed=1701)
         x = rng.integers(10, size=(3, 4))
         x
Out[12]: array([[9, 4, 0, 3],
                [8, 6, 3, 1],
                [3, 7, 4, 0]])

In [13]: x < 6
Out[13]: array([[False,  True,  True,  True],
                [False, False,  True,  True],
                [ True, False,  True,  True]])
```

Во всех случаях результатом является массив булевых значений, и для работы с этими булевыми результатами в библиотеке NumPy имеется набор простых паттернов.

Работа с булевыми массивами

Для работы с массивами булевых значений существует множество доступных и удобных операций. Мы будем работать с созданным нами ранее двумерным массивом `x`.

```
In [14]: print(x)
Out[14]: [[9 4 0 3]
          [8 6 3 1]
          [3 7 4 0]]
```

Подсчет количества элементов

Для подсчета количества элементов со значением True в булевом массиве удобно использовать функцию np.count_nonzero:

```
In[15]: # Сколько значений в массиве меньше 6?
        np.count_nonzero(x < 6)
Out[15]: 8
```

Как видите, в массиве есть восемь элементов со значением меньше 6. Другой способ получить эту информацию — воспользоваться функцией np.sum. В этом случае False интерпретируется как 0, а True — как 1:

```
In[16]: np.sum(x < 6)
Out[16]: 8
```

Преимущество функции np.sum в том, что, подобно другим функциям агрегирования в библиотеке NumPy, она также может выполнять суммирование по столбцам или строкам:

```
In[17]: # Сколько значений меньше 6 содержится в каждой строке?
        np.sum(x < 6, axis=1)
Out[17]: array([4, 2, 2])
```

Эта инструкция подсчитывает количество значений меньше 6 в каждой строке матрицы.

Если потребуется быстро проверить, существует ли хотя бы одно истинное значение или все ли значения истинны, можно воспользоваться (как вы наверняка догадались) функциями np.any и np.all:

```
In[18]: # Имеются ли в массиве значения, превышающие 8?
        np.any(x > 8)
Out[18]: True

In[19]: # Имеются ли в массиве значения меньше 0?
        np.any(x < 0)
Out[19]: False

In[20]: # Все значения меньше 10?
        np.all(x < 10)
Out[20]: True

In[21]: # Все значения равны 6?
        np.all(x == 6)
Out[21]: False
```

Функции np.any и np.all также можно использовать по конкретным осям. Например:

```
In[22]: # Все значения в каждой строке меньше 8?
        np.all(x < 8, axis=1)
Out[22]: array([False, False,  True])
```

В третьей строке все значения меньше 8, а в первой и второй — нет.

Наконец, небольшое предупреждение: как упоминалось в главе 7, в языке Python имеются встроенные функции `sum`, `any` и `all`. Их синтаксис отличается от аналогичных функций в библиотеке NumPy. В частности, они будут выдавать ошибку или неожиданные результаты при применении к многомерным массивам. Убедитесь, что в своих примерах используете функции `np.sum`, `np.any` и `np.all`.

Булевы операторы

Вы уже знаете, как подсчитать все дни с осадками менее 20 мм или все дни с осадками более 10 мм. Но что, если нужна информация обо всех днях с количеством осадков от 10 до 20 мм? Ее можно получить с помощью *побитовых логических операторов* (bitwise logic operators) языка Python: `&`, `|`, `^` и `~`. Как и обычные арифметические операторы, библиотека NumPy перегружает их с использованием универсальных функций, поэлементно работающих с (обычно булевыми) массивами.

Например, подобную составную задачу можно решить следующим образом:

```
In [23]: np.sum((rainfall_mm > 10) & (rainfall_mm < 20))
Out[23]: 16
```

Как показывают результаты, в 2015 году в Сиэтле было 16 дней с количеством осадков от 10 до 20 мм.

Обратите внимание: круглые скобки здесь играют важную роль в силу правил приоритета операторов, без скобок это выражение вычислялось бы следующим образом, что привело бы к ошибке:

```
rainfall_mm > (10 & rainfall_mm) < 20
```

Рассмотрим более сложное выражение. Согласно закону де Моргана тот же результат можно получить иначе:

```
In [24]: np.sum(~( (rainfall_mm <= 10) | (rainfall_mm >= 20) ))
Out[24]: 16
```

Комбинируя операторы сравнения и булевы операторы при работе с массивами, можно получить целый диапазон эффективных логических операций.

В табл. 9.2 перечислены побитовые булевы операторы и эквивалентные им универсальные функции.

Таблица 9.2. Побитовые булевы операторы и эквивалентные им универсальные функции

Оператор	Эквивалентная универсальная функция
&	np.bitwise_and
|	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

С помощью этих инструментов можно извлекать различную полезную информацию из наших данных по осадкам. Вот примеры результатов, которые можно вычислить, комбинируя булевы операции и операции агрегирования:

```
In [25]: print("Number days without rain:  ", np.sum(rainfall_mm == 0))
         print("Number days with rain:     ", np.sum(rainfall_mm != 0))
         print("Days with more than 10 mm: ", np.sum(rainfall_mm > 10))
         print("Rainy days with < 5 mm:    ", np.sum((rainfall_mm > 0) &
                                                     (rainfall_mm < 5)))
Out[25]: Number days without rain:   221
         Number days with rain:      144
         Days with more than 10 mm:  34
         Rainy days with < 5 mm:     83
```

Булевы массивы как маски

В предыдущем разделе мы рассмотрели примеры вычисления сводных показателей непосредственно по булевым массивам. Гораздо больше возможностей дает использование булевых массивов в качестве масок, для выбора нужных подмножеств самих данных. Вернемся к массиву x:

```
In [26]: x
Out[26]: array([[9, 4, 0, 3],
               [8, 6, 3, 1],
               [3, 7, 4, 0]])
```

и допустим, что нам необходимо получить все значения из массива x меньше, скажем, 5. Мы легко можем получить булев массив, соответствующий этому условию:

```
In [27]: x < 5
Out[27]: array([[False,  True,  True,  True],
               [False, False,  True,  True],
               [ True, False,  True,  True]])
```

Чтобы *выбрать* нужные значения, достаточно проиндексировать исходный массив x по этому булеву массиву. Такое действие носит название операции *наложения маски* или *маскирования*:

```
In [28]: x[x < 5]
Out[28]: array([4, 0, 3, 3, 1, 3, 4, 0])
```

Мы получили одномерный массив, заполненный всеми значениями, удовлетворяющими условию. Другими словами, мы получили все значения, находящиеся в массиве `x` в позициях, в которых в массиве-маске находятся значения `True`.

Теперь с этими значениями можно выполнять требуемые операции. Например, можно вычислить какой-нибудь статистический показатель на наших данных о величине осадков в Сиэтле:

```
In [29]: # создаем маску для всех дождливых дней
         rainy = (rainfall_mm > 0)

         # создаем маску для всех летних дней (21 июня[1] -- это 172-й день года)
         days = np.arange(365)
         summer = (days > 172) & (days < 262)

         print("Median precip on rainy days in 2015 (mm):   ",
               np.median(rainfall_mm[rainy]))
         print("Median precip on summer days in 2015 (mm):  ",
               np.median(rainfall_mm[summer]))
         print("Maximum precip on summer days in 2015 (mm): ",
               np.max(rainfall_mm[summer]))
         print("Median precip on non-summer rainy days (mm):",
               np.median(rainfall_mm[rainy & ~summer]))

Out[29]: Median precip on rainy days in 2015 (mm):    3.8
         Median precip on summer days in 2015 (mm):   0.0
         Maximum precip on summer days in 2015 (mm):  32.5
         Median precip on non-summer rainy days (mm): 4.1
```

Комбинируя булевы операции с операциями маскирования и агрегирования, можно очень быстро получать ответы на подобные вопросы относительно набора данных.

Ключевые слова and/or и операторы &/|

Разница между ключевыми словами `and` и `or` и операторами `&` и `|` — распространенный источник путаницы. Давайте разберемся, что и когда следует использовать.

Различие заключается в следующем: ключевые слова `and` и `or` работают с объектами как с единым целым, а операторы `&` и `|` оперируют отдельными битами внутри объектов.

[1] Астрономическое лето в Северном полушарии начинается 21 июня, в день летнего солнцестояния.

Встретив ключевое слово and или or, интерпретатор Python рассматривает объект как единую булеву сущность. В Python все ненулевые целые числа будут рассматриваться как True. То есть:

```
In [30]: bool(42), bool(0)
Out[30]: (True, False)

In [31]: bool(42 and 0)
Out[31]: False

In [32]: bool(42 or 0)
Out[32]: True
```

При применении к целым числам операторы & и | манипулируют их битовыми представлениями, фактически применяя операции and и or к отдельным битам, составляющим числа:

```
In [33]: bin(42)
Out[33]: '0b101010'

In [34]: bin(59)
Out[34]: '0b111011'

In [35]: bin(42 & 59)
Out[35]: '0b101010'

In [36]: bin(42 | 59)
Out[36]: '0b111011'
```

Обратите внимание, что для получения результата сравниваются соответствующие биты двоичного представления чисел.

Массив булевых значений в NumPy можно рассматривать как строку битов, где 1 = True и 0 = False, а результат применения операторов & и | аналогичен вышеприведенному:

```
In [37]: A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
         B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
         A | B
Out[37]: array([ True,  True,  True, False,  True,  True])
```

Использование же ключевого слова or для этих массивов приведет к вычислению истинности или ложности всего объекта массива, которые формально не определены:

```
In [38]: A or B
ValueError: The truth value of an array with more than one element is
          > ambiguous.
          a.any() or a.all()
```

Аналогично при вычислении булева выражения с заданным массивом следует использовать операторы `&` и `|`, а не операции `and` или `or`:

```
In [39]: x = np.arange(10)
         (x > 4) & (x < 8)
Out[39]: array([False, False, False, False, False,  True,  True,  True, False,
                False])
```

Попытка же вычислить истинность или ложность всего массива приведет к уже наблюдавшейся выше ошибке `ValueError`:

```
In [40]: (x > 4) and (x < 8)
ValueError: The truth value of an array with more than one element is
          > ambiguous.
          a.any() or a.all()
```

Итак, запомните: операции `and` и `or` вычисляют единое булево значение для всего объекта, в то время как операторы `&` и `|` вычисляют много булевых значений для содержимого (отдельных битов или байтов) объекта. Второй из этих вариантов практически всегда будет именно той операцией, которая будет вам нужна при работе с булевыми массивами NumPy.

ГЛАВА 10

«Прихотливая» индексация

В предыдущих главах мы рассмотрели доступ и изменение частей массива с помощью простых индексов (например, `arr[0]`), срезов (например, `arr[:5]`) и булевых масок (например, `arr[arr > 0]`). В этой главе мы изучим другую разновидность индексации массивов, так называемую *прихотливую* (fancy) или *векторизованную* (vectorized) индексацию. «Прихотливая» индексация позволяет очень быстро получать и изменять сложные подмножества значений в массивах.

Возможности «прихотливой» индексации

Суть «прихотливой» индексации проста: передача множества индексов с целью одновременного доступа к нескольким элементам массива. Например, рассмотрим следующий массив:

```
In [1]: import numpy as np
        rng = np.random.default_rng(seed=1701)

        x = rng.integers(100, size=10)
        print(x)
Out[1]: [90 40  9 30 80 67 39 15 33 79]
```

Допустим, нам требуется обратиться к трем различным элементам. Это можно сделать так:

```
In [2]: [x[3], x[7], x[2]]
Out[2]: [30, 15, 9]
```

Вместе с тем можно передать единый список индексов и получить тот же результат:

```
In [3]: ind = [3, 7, 4]
        x[ind]
Out[3]: array([30, 15, 80])
```

В случае «прихотливой» индексации форма результата отражает форму *массивов индексов* (index arrays), а не форму исходного *индексируемого массива*:

```
In [4]: ind = np.array([[3, 7],
                        [4, 5]])
        x[ind]
Out[4]: array([[30, 15],
               [80, 67]])
```

«Прихотливая» индексация может применяться и к многомерным массивам. Рассмотрим следующий массив:

```
In [5]: X = np.arange(12).reshape((3, 4))
        X
Out[5]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

Как и в стандартной индексации, первый индекс относится к строкам, а второй — к столбцам:

```
In [6]: row = np.array([0, 1, 2])
        col = np.array([2, 1, 3])
        X[row, col]
Out[6]: array([ 2,  5, 11])
```

Первое значение в результате — `X[0, 2]`, второе — `X[1, 1]`, и третье — `X[2, 3]`. Составление пар индексов при «прихотливой» индексации подчиняется всем правилам транслирования, описанным в главе 8. Так, например, если мы скомбинируем вектор-столбец и вектор-строку в индексах, то получим двумерный результат:

```
In [7]: X[row[:, np.newaxis], col]
Out[7]: array([[ 2,  1,  3],
               [ 6,  5,  7],
               [10,  9, 11]])
```

Каждое значение в строке соединяется с каждым вектором-столбцом точно так же, как при транслировании арифметических операций. Например:

```
In[8]: row[:, np.newaxis] * col
Out[8]: array([[0, 0, 0],
               [2, 1, 3],
               [4, 2, 6]])
```

Используя «прихотливую» индексацию, всегда помните, что возвращаемое значение отражает *транслируемую форму индексов*, а не форму индексируемого массива.

Комбинированная индексация

Для реализации еще более сложных операций «прихотливую» индексацию можно комбинировать с другими схемами индексации. Например, пусть дан массив X:

```
In [9]: print(X)
Out[9]: [[ 0  1  2  3]
         [ 4  5  6  7]
         [ 8  9 10 11]]
```

Можно комбинировать «прихотливые» и простые индексы:

```
In[10]: X[2, [2, 0, 1]]
Out[10]: array([10,  8,  9])
```

Также можно комбинировать «прихотливые» индексы и срезы:

```
In[11]: X[1:, [2, 0, 1]]
Out[11]: array([[ 6,  4,  5],
               [10,  8,  9]])
```

И «прихотливую» индексацию с маскированием:

```
In[12]: mask = np.array([1, 0, 1, 0], dtype=bool)
        X[row[:, np.newaxis], mask]
Out[12]: array([[ 0,  2],
               [ 4,  6],
               [ 8, 10]])
```

Все эти варианты индексации образуют набор чрезвычайно гибких операций доступа к значениям массивов и их изменения.

Пример: выборка случайных точек

Типичная сфера применения «прихотливой» индексации — выборка подмножеств строк из матрицы. Пусть имеется матрица размером $N \times D$, представляющая N точек в D измерениях, как, например, следующие точки, полученные из двумерного нормального распределения:

```
In [13]: mean = [0, 0]
         cov = [[1, 2],
                [2, 5]]
```

```
        X = rng.multivariate_normal(mean, cov, 100)
        X.shape
Out[13]: (100, 2)
```

С помощью инструментов рисования графиков, которые мы обсудим в части IV, можно визуализировать эти точки в виде диаграммы рассеяния (рис. 10.1):

```
In [14]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')

         plt.scatter(X[:, 0], X[:, 1]);
```

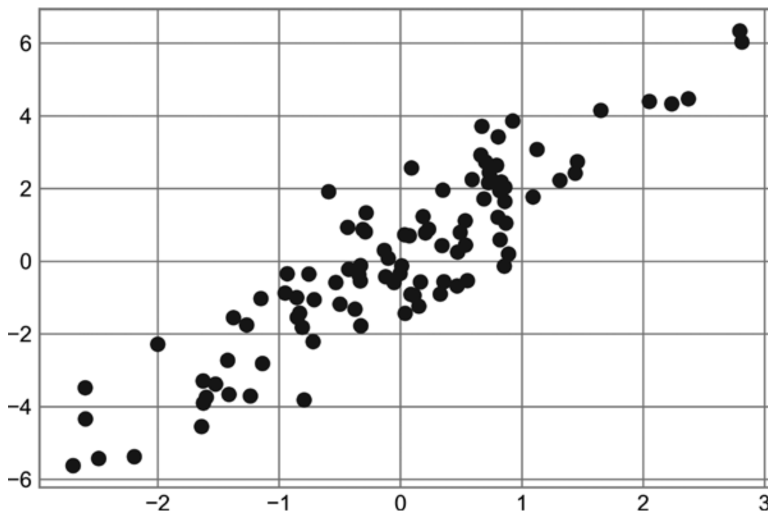


Рис. 10.1. Нормально распределенные точки

Воспользуемся «прихотливой» индексацией для выборки 20 случайных точек. Мы сделаем это с помощью выбора предварительно 20 случайных индексов без повторов и воспользуемся этими индексами для выбора части исходного массива:

```
In [15]: indices = np.random.choice(X.shape[0], 20, replace=False)
         indices
Out[15]: array([82, 84, 10, 55, 14, 33,  4, 16, 34, 92, 99, 64,  8, 76, 68, 18, 59,
                80, 87, 90])

In [16]: selection = X[indices]  # Тут используется "прихотливая" индексация
         selection.shape
Out[16]: (20, 2)
```

Чтобы посмотреть, какие точки выбраны, нарисуем поверх первой диаграммы большие круги в местах расположения выбранных точек (рис. 10.2).

```
In [17]: plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
         plt.scatter(selection[:, 0], selection[:, 1],
                     facecolor='none', edgecolor='black', s=200);
```

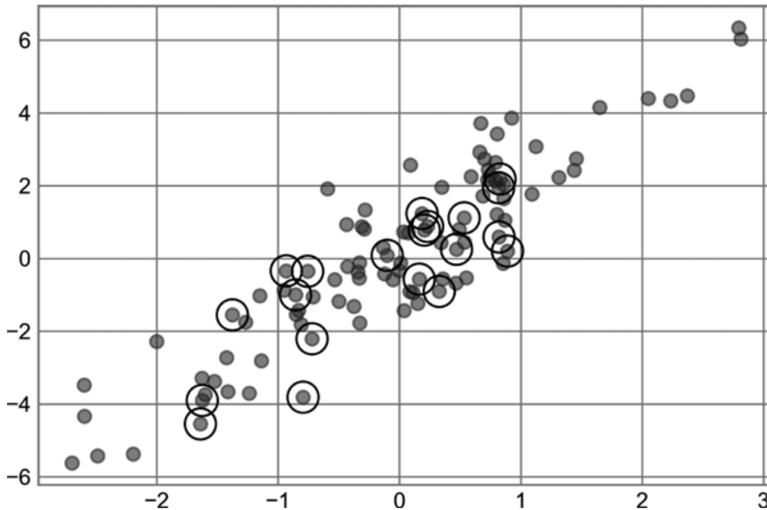


Рис. 10.2. Нормально распределенные точки

Подобная стратегия часто используется для быстрого секционирования наборов данных, нередко требуемого при разделении на обучающую/тестовую последовательность для проверки статистических моделей (см. главу 39), а также в методах выборки ответов на статистические вопросы.

Изменение значений с помощью «прихотливой» индексации

«Прихотливую» индексацию можно использовать не только для доступа к частям массива, но и для их модификации. Например, допустим, что у нас есть массив индексов и нам нужно присвоить соответствующим элементам массива какие-то значения:

```
In [18]: x = np.arange(10)
         i = np.array([2, 1, 8, 4])
```

```
        x[i] = 99
        print(x)
Out[18]: [ 0 99 99  3 99  5  6  7 99  9]
```

Для этого можно использовать любой из операторов присваивания. Например:

```
In [19]: x[i] -= 10
        print(x)
Out[19]: [ 0 89 89  3 89  5  6  7 89  9]
```

Замечу, однако, что повторяющиеся индексы при подобных операциях могут привести к некоторым потенциально неожиданным результатам. Рассмотрим следующий пример:

```
In [20]: x = np.zeros(10)
        x[[0, 0]] = [4, 6]
        print(x)
Out[20]: [6. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Куда пропало значение 4? В результате этой операции сначала выполняется присваивание `x[0] = 4` с последующим присваиванием `x[0] = 6`. В итоге `x[0]` содержит значение 6.

Довольно логично, но рассмотрим такую операцию:

```
In [21]: i = [2, 3, 3, 4, 4, 4]
        x[i] += 1
        x
Out[21]: array([6., 0., 1., 1., 1., 0., 0., 0., 0., 0.])
```

Можно было ожидать, что `x[3]` будет содержать значение 2, а `x[4]` — значение 3, так как именно столько раз повторяется каждый из этих индексов. Почему же это не так? По сути, не из-за того, что выражение `x[i] += 1` задумывалось как сокращенная форма записи для `x[i] = x[i] + 1`. Вычисляется выражение `x[i] + 1`, после чего результат присваивается соответствующим индексам элементам в массиве `x`. Получается, что это не выполняемый несколько раз инкремент, а присваивание, приводящее к интуитивно неочевидным результатам.

Что же делать, если требуется другое поведение от повторяющейся операции? В этом случае можно воспользоваться методом `at` универсальных функций и сделать следующее:

```
In [22]: x = np.zeros(10)
        np.add.at(x, i, 1)
        print(x)
Out[22]: [0. 0. 1. 2. 3. 0. 0. 0. 0. 0.]
```

Метод `at` применяет соответствующий оператор к элементам с заданными индексами (в данном случае `i`) с использованием заданного значения (в данном случае 1). Аналогичный по духу метод универсальных функций `reduceat`, о котором можно прочитать в документации библиотеки NumPy (https://oreil.ly/7ys9D).

Пример: разбиение данных на интервалы

Эти идеи можно использовать для эффективного разбиения данных с целью применения операций к интервалам. Например, есть 1000 значений, и необходимо быстро выяснить, как они распределяются по массиву интервалов. Это можно сделать с помощью метода `at` универсальных функций:

```
In[23]: np.random.seed(42)
        x = np.random.randn(100)

        # Рассчитываем гистограмму вручную
        bins = np.linspace(-5, 5, 20)
        counts = np.zeros_like(bins)

        # Ищем подходящий интервал для каждого x
        i = np.searchsorted(bins, x)

        # Добавляем 1 к каждому из интервалов
        np.add.at(counts, i, 1)
```

Полученные числа отражают количество точек в каждом из интервалов, другими словами, гистограмму (рис. 10.3):

```
In[24]: # Визуализируем результаты
        plt.plot(bins, counts, linestyle='steps');
```

Получать каждый раз гистограмму таким образом нет необходимости. Библиотека Matplotlib предоставляет процедуру `plt.hist`, которая позволяет сделать то же самое одной строкой кода:

```
plt.hist(x, bins, histtype='step');
```

Эта функция создаст практически точно такую же диаграмму, как на рис. 10.3. Для разбиения данных по интервалам библиотека Matplotlib предлагает функцию `np.histogram`, выполняющую вычисления, очень похожие на сделанные нами. Давайте сравним их:

```
In [25]: print(f"NumPy histogram ({len(x)} points):")
         %timeit counts, edges = np.histogram(x, bins)
```

```
        print(f"Custom histogram ({len(x)} points):")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
Out[25]: NumPy histogram (100 points):
        33.8 µs ± 311 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
        Custom histogram (100 points):
        17.6 µs ± 113 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

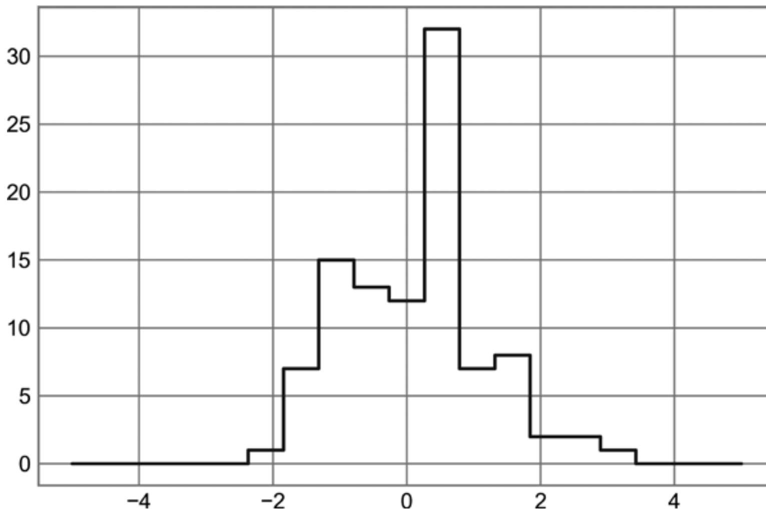


Рис. 10.3. Вычисленная вручную гистограмма

Наш собственный однострочный алгоритм работает в несколько раз быстрее, чем оптимизированный алгоритм из библиотеки NumPy! Как это возможно? Если заглянуть в исходный код процедуры `np.histogram` (для этого в оболочке IPython введите команду `np.histogram??`), то можно заметить, что она гораздо сложнее простого поиска и подсчета, выполненного нами. Дело в том, что алгоритм из библиотеки NumPy более гибкий, потому что разработан с ориентацией на более высокую производительность при значительном увеличении количества точек данных:

```
In [26]: x = rng.normal(size=1000000)
        print(f"NumPy histogram ({len(x)} points):")
        %timeit counts, edges = np.histogram(x, bins)

        print(f"Custom histogram ({len(x)} points):")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
Out[26]: NumPy histogram (1000000 points):
        84.4 ms ± 2.82 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
        Custom histogram (1000000 points):
        128 ms ± 2.04 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Как показывает это сравнение, эффективность алгоритма почти всегда непростой вопрос. Эффективный для больших наборов данных алгоритм не всегда окажется оптимальным вариантом для маленьких, и наоборот. Но преимущество самостоятельного программирования этого алгоритма заключается в том, что, получив понимание работы подобных простых методов, вы сможете «строить» из этих «кирпичиков» очень интересные варианты исследования данных. Ключом к эффективному использованию языка Python в приложениях, требующих обработки больших объемов данных, является знание о существовании удобных процедур, таких как `np.histogram`, и сферах их использования. Кроме того, нужно знать, как применять низкоуровневую функциональность при необходимости в узконаправленном поведении.

ГЛАВА 11

Сортировка массивов

До сих пор мы занимались в основном инструментами доступа и изменения данных в массивах с помощью библиотеки NumPy. Этот раздел охватывает алгоритмы, связанные с сортировкой значений в массивах NumPy. Данные алгоритмы — излюбленная тема вводных курсов по информатике. Если вы когда-либо были слушателем такого курса, то вам, наверное, снились сны (в зависимости от вашего темперамента, возможно, кошмары) о *сортировке вставкой*, *сортировке методом выбора*, *сортировке слиянием*, *быстрой сортировке*, *пузырьковой сортировке* и многих других методах сортировки. Все они представляют собой средства для решения одной и той же задачи — сортировки значений в списке или массиве.

В Python имеется несколько встроенных функций и методов для сортировки списков и других итерируемых объектов. Функция `sorted` принимает список и возвращает его отсортированную версию:

```
In [1]: L = [3, 1, 4, 1, 5, 9, 2, 6]
        sorted(L)  # вернет отсортированную копию
Out[1]: [1, 1, 2, 3, 4, 5, 6, 9]
```

Метод `sort` списков, напротив, сортирует список на месте:

```
In [2]: L.sort()  # выполняет сортировку на месте и возвращает None
        print(L)
Out[2]: [1, 1, 2, 3, 4, 5, 6, 9]
```

Методы сортировки в Python очень гибкие и могут работать с любыми итерируемыми объектами. Вот пример сортировки строки:

```
In [3]: sorted('python')
Out[3]: ['h', 'n', 'o', 'p', 't', 'y']
```

Эти встроенные методы сортировки очень удобны, но, как обсуждалось выше, динамическая типизация значений в Python означает, что они не так эффективны, как процедуры, специально разработанные для работы с массивами чисел и включенные в библиотеку NumPy.

Быстрая сортировка в библиотеке NumPy: функции np.sort и np.argsort

Функция `np.sort` — аналог встроенной функции `sorted` в Python. Она возвращает отсортированную копию массива:

```
In [4]: import numpy as np

        x = np.array([2, 1, 4, 3, 5])
        np.sort(x)
Out[4]: array([1, 2, 3, 4, 5])
```

Аналогично методу `sort` списков в Python массивы NymPy имеют метод `sort`, сортирующий массив на месте:

```
In [5]: x.sort()
        print(x)
Out[5]: [1 2 3 4 5]
```

Имеется также родственная функция `argsort`, возвращающая индексы отсортированных элементов:

```
In [6]: x = np.array([2, 1, 4, 3, 5])
        i = np.argsort(x)
        print(i)
Out[6]: [1 0 3 2 4]
```

Первый элемент этого результата соответствует индексу минимального элемента, второй — индексу второго по величине и т. д. В дальнейшем эти индексы при желании можно будет использовать для построения (посредством «прихотливой» индексации) отсортированного массива:

```
In [7]: x[i]
Out[7]: array([1, 2, 3, 4, 5])
```

Далее в этой главе вы увидите еще примеры использования `argsort`.

Сортировка по строкам и столбцам

У алгоритмов сортировки в NumPy имеется удобная возможность выполнять сортировку по конкретным строкам или столбцам многомерного массива путем задания аргумента `axis`. Например:

```
In [8]: rng = np.random.default_rng(seed=42)
        X = rng.integers(0, 10, (4, 6))
        print(X)
```

```
Out[8]: [[0 7 6 4 4 8]
         [0 6 2 0 5 9]
         [7 7 7 7 5 1]
         [8 4 5 3 1 9]]

In [9]: # Сортируем все столбцы массива X
        np.sort(X, axis=0)
Out[9]: array([[0, 4, 2, 0, 1, 1],
               [0, 6, 5, 3, 4, 8],
               [7, 7, 6, 4, 5, 9],
               [8, 7, 7, 7, 5, 9]])

In [10]: # Сортируем все строки массива X
         np.sort(X, axis=1)
Out[10]: array([[0, 4, 4, 6, 7, 8],
                [0, 0, 2, 5, 6, 9],
                [1, 5, 7, 7, 7, 7],
                [1, 3, 4, 5, 8, 9]])
```

Не забывайте, что при этом все строки или столбцы рассматриваются как отдельные массивы, так что любые возможные взаимосвязи между значениями строк или столбцов будут утеряны.

Частичная сортировка: секционирование

Иногда требуется не отсортировать весь массив, а просто найти *k* наименьших значений в нем. Библиотека NumPy предоставляет для этой цели функцию `np.partition`. Она принимает на входе массив и число *k*. Результат представляет собой новый массив с *k* наименьшими значениями слева от точки разбиения и остальными значениями справа от нее в произвольном порядке:

```
In [11]: x = np.array([7, 2, 3, 1, 6, 5, 4])
         np.partition(x, 3)
Out[11]: array([2, 1, 3, 4, 6, 5, 7])
```

Первые три значения в итоговом массиве — три наименьших значения в нем, а на остальных позициях массива располагаются все прочие значения. Внутри каждой из двух секций элементы располагаются в произвольном порядке.

Аналогично сортировке можно секционировать по произвольной оси многомерного массива:

```
In [12]: np.partition(X, 2, axis=1)
Out[12]: array([[0, 4, 4, 7, 6, 8],
                [0, 0, 2, 6, 5, 9],
                [1, 5, 7, 7, 7, 7],
                [1, 3, 4, 5, 8, 9]])
```

Результат представляет собой массив, в котором на первых двух позициях в каждой строке находятся наименьшие значения из этой строки, а остальные значения заполняют прочие места.

Наконец, аналогично функции `np.argsort`, вычисляющей индексы для сортировки, существует функция `np.argpartition`, вычисляющая индексы для секции. Мы увидим ее в действии в следующем разделе.

Пример: k ближайших соседей

Давайте вкратце рассмотрим, как можно использовать функцию `argsort` по нескольким осям для поиска ближайших соседей каждой точки из определенного набора. Начнем с создания случайного набора из десяти точек на двумерной плоскости. По стандартным соглашениям образуем из них массив 10 × 2:

```
In [13]: X = rng.random((10, 2))
```

Чтобы наглядно представить себе расположение этих точек, нарисуем для них диаграмму рассеяния (рис. 11.1):

```
In [14]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')
         plt.scatter(X[:, 0], X[:, 1], s=100);
```

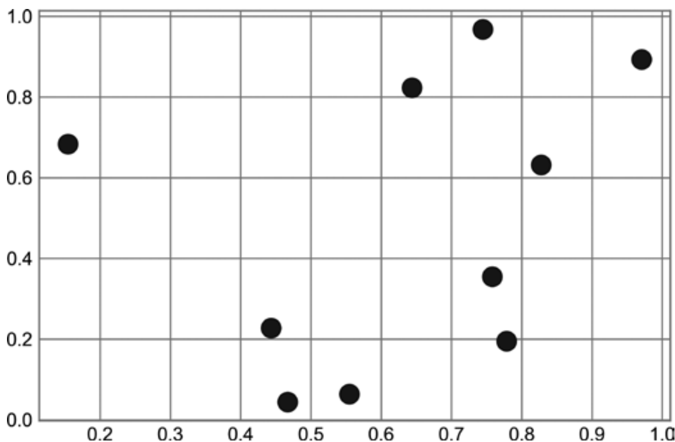


Рис. 11.1. Визуализация точек в примере k ближайших соседей

Теперь вычислим расстояние между всеми парами точек. Вспоминаем, что квадрат расстояния между двумя точками равен сумме квадратов расстояний между ними

по каждой из координат. Воспользовавшись возможностями эффективного транслирования (см. главу 8) и агрегирования (см. главу 7), предоставляемыми библиотекой NumPy, мы можем вычислить матрицу квадратов расстояний одной строкой кода:

```
In [15]: dist_sq = np.sum((X[:, np.newaxis] - X[np.newaxis, :]) ** 2, axis=-1)
```

Эта операция довольно сложна по синтаксису и может привести в замешательство тех, кто плохо знаком с правилами транслирования библиотеки NumPy. В подобных случаях полезно разбить операцию на отдельные шаги:

```
In [16]: # Для каждой пары точек вычисляем разности их координат
         differences = X[:, np.newaxis] - X[np.newaxis, :]
         differences.shape
Out[16]: (10, 10, 2)

In [17]: # Возводим разности координат в квадрат
         sq_differences = differences ** 2
         sq_differences.shape
Out[17]: (10, 10, 2)

In [18]: # Суммируем квадраты разностей координат
         # для получения квадрата расстояния
         dist_sq = sq_differences.sum(-1)
         dist_sq.shape
Out[18]: (10, 10)
```

На всякий случай для контроля проверим, что диагональные элементы матрицы (то есть расстояния между каждой точкой и ей самой) содержат нули:

```
In [19]: dist_sq.diagonal()
Out[19]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Получив матрицу квадратов расстояний между взятыми попарно точками, мы можем воспользоваться функцией `np.argsort` для сортировки каждой строки в отдельности. Крайние слева столбцы будут представлять собой индексы ближайших соседей:

```
In [20]: nearest = np.argsort(dist_sq, axis=1)
         print(nearest)
Out[20]: [[0 9 3 5 4 8 1 6 2 7]
          [1 7 2 6 4 8 3 0 9 5]
          [2 7 1 6 4 3 8 0 9 5]
          [3 0 4 5 9 6 1 2 8 7]
          [4 6 3 1 2 7 0 5 9 8]
          [5 9 3 0 4 6 8 1 2 7]
          [6 4 2 1 7 3 0 5 9 8]
          [7 2 1 6 4 3 8 0 9 5]
          [8 0 1 9 3 4 7 2 6 5]
          [9 0 5 3 4 8 6 1 2 7]]
```

Обратите внимание, что первый столбец содержит числа от 0 до 9 в порядке возрастания: это объясняется тем, что ближайшим соседом каждой точки является она сама, как и можно было ожидать.

Выполнив полную сортировку, мы проделали лишнюю работу. Если нас интересовали *k* ближайших соседей, то было бы достаточно секционировать все строки так, чтобы сначала шли $k + 1$ минимальных квадратов расстояний, а бóльшие расстояния заполняли оставшиеся позиции массива. Сделать это можно с помощью функции `np.argpartition`:

```
In [21]: K = 2
         nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

Чтобы визуализировать эту сеть соседей, выведем на диаграмму точки вдоль линий, связывающих каждую точку с ее ближайшими двумя соседями (рис. 11.2):

```
In [22]: plt.scatter(X[:, 0], X[:, 1], s=100)

         # Рисуем линии из каждой точки к ее двум ближайшим соседям
         K = 2

         for i in range(X.shape[0]):
             for j in nearest_partition[i, :K+1]:
                 # чертим линию от X[i] до X[j]
                 # Используем для этого "магическую" функцию zip:
                 plt.plot(*zip(X[j], X[i]), color='black')
```

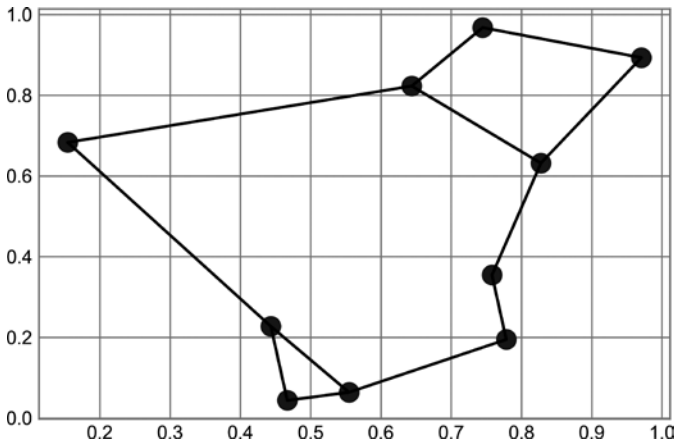


Рис. 11.2. Визуализация соседей каждой точки

От каждой нарисованной на диаграмме точки ведут линии к двум ее ближайшим соседям. На первый взгляд может показаться странным, что из некоторых точек

отходит более двух линий. Дело в том, что, если точка A — один из двух ближайших соседей точки B, вовсе не обязательно, что точка B — один из двух ближайших соседей точки A.

Хотя применяемые при этом транслирование и построчная сортировка могут показаться более запутанным подходом, чем написание цикла, оказывается, что такой способ работы с подобными данными на языке Python весьма эффективен. Как бы ни было заманчиво сделать то же самое, вручную организовав цикл по данным и сортировку каждого набора соседей отдельно, получившийся в итоге алгоритм почти наверняка будет работать медленнее, чем рассмотренная выше векторизованная версия. Красота такого подхода — в его независимости от размера входных данных: можно с одинаковой легкостью вычислить соседей среди 100 или 1 000 000 точек в любом количестве измерений, и код будет выглядеть точно так же.

Наконец, отмечу, что для выполнения поиска соседей в очень больших массивах данных существуют основанные на деревьях и/или приближенные алгоритмы, масштабирующиеся как $O[N \log N]$ или даже лучше, в отличие от метода поиска простым перебором $O[N^2]$. Один из примеров таких алгоритмов — K-мерное дерево (KD-tree), реализованное в библиотеке Scikit-Learn (https://oreil.ly/lUFb8).

ГЛАВА 12

Структурированные данные: структурированные массивы NumPy

Часто данные можно представить в виде массива однородных значений, но иногда это невозможно. В этой главе демонстрируется использование таких возможностей библиотеки NumPy, как *структурированные массивы* (structured arrays) и *массивы записей* (record arrays), обеспечивающих эффективное хранилище для составных неоднородных данных. Хотя демонстрируемые паттерны удобны для простых операций, подобные сценарии часто применяются и при работе со структурами данных `DataFrame` из библиотеки Pandas, которые мы рассмотрим в части III.

```
In [1]: import numpy as np
```

Пускай у нас имеется несколько категорий данных (например, имя, возраст и вес) о нескольких людях, и мы хотели бы хранить эти значения для использования в программе на языке Python. Можно сохранить их в отдельных массивах:

```
In [2]: name = ['Alice', 'Bob', 'Cathy', 'Doug']
        age = [25, 45, 37, 19]
        weight = [55.0, 85.5, 68.0, 61.5]
```

Однако это не очень правильное решение, потому что в нем нет ничего, что указывало бы на связь между массивами. Лучше было бы использовать для хранения этих данных единую структуру. Библиотека NumPy позволяет это осуществить посредством применения структурированных массивов, представляющих собой массивы данных составного типа.

Вспомните, как ранее мы создавали простой массив с помощью следующего выражения:

```
In [3]: x = np.zeros(4, dtype=int)
```

Аналогично можно создать структурированный массив, применяя спецификацию составного типа данных:

```
In [4]: # Используем для структурированного массива составной тип данных
        data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                                  'formats':('U10', 'i4', 'f8')})
        print(data.dtype)
Out[4]: [('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

`'U10'` означает «строку в кодировке Unicode максимальной длины 10», `'i4'` — «4-байтное (то есть 32-битное) целое число», а `'f8'` — «8-байтное (то есть 64-битное) число с плавающей точкой». Мы обсудим другие варианты подобного кодирования типов в следующем разделе.

Создав пустой массив-контейнер, мы можем заполнить его нашим списком значений:

```
In [5]: data['name'] = name
        data['age'] = age
        data['weight'] = weight
        print(data)
Out[5]: [('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )
         ('Doug', 19, 61.5)]
```

Как мы и хотели, данные теперь располагаются все вместе в одном удобном структурированном массиве.

Одно из важнейших удобств структурированных массивов — возможность ссылаться на значения и по именам, и по индексам:

```
In [6]: # Извлечь все имена
        data['name']
Out[6]: array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')

In [7]: # Извлечь первую строку данных
        data[0]
Out[7]: ('Alice', 25, 55.)

In [8]: # Извлечь имя из последней строки
        data[-1]['name']
Out[8]: 'Doug'
```

Появляется возможность с помощью булева маскирования выполнять и более сложные операции, такие как фильтрация по возрасту:

```
In [9]: # Извлечь имена людей с возрастом менее 30
        data[data['age'] < 30]['name']
Out[9]: array(['Alice', 'Doug'], dtype='<U10')
```

Для выполнения более сложных операций лучше использовать пакет Pandas, который будет рассмотрен в части III. Библиотека Pandas предоставляет объект `DataFrame` — структуру, основанную на массивах NumPy, обладающую массой полезных возможностей для работы с данными.

Создание структурированных массивов

Типы данных для структурированных массивов можно задавать несколькими способами. Ранее мы рассмотрели метод с использованием словаря:

```
In [10]: np.dtype({'names':('name', 'age', 'weight'),
                   'formats':('U10', 'i4', 'f8')})
Out[10]: dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

Для ясности можно задавать числовые типы как с применением типов данных Python, так и типов `dtype` из библиотеки NumPy:

```
In [11]: np.dtype({'names':('name', 'age', 'weight'),
                   'formats':((np.str_, 10), int, np.float32)})
Out[11]: dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

Составные типы данных можно задавать в виде списка кортежей:

```
In [12]: np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
Out[12]: dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

Если имена типов для вас не важны, можете задать только сами типы данных, перечислив их в строке через запятую:

```
In [13]: np.dtype('S10,i4,f8')
Out[13]: dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

Сокращенные строковые коды форматов могут показаться запутанными, но они основаны на простых принципах. Первый (необязательный) символ — `<` или `>`, означает «число с обратным порядком байтов» или «число с прямым порядком байтов» соответственно и задает порядок значащих битов. Следующий символ задает тип данных: символ, байт, целое число, число с плавающей точкой и т. д. (табл. 12.1). Последний символ или символы отражают размер объекта в байтах.

Таблица 12.1. Типы данных библиотеки NumPy

Символ	Описание	Пример
`'b'`	Байт	`np.dtype('b')`
`'i'`	Целое число со знаком	`np.dtype('i4') == np.int32`

Символ	Описание	Пример
'u'	Целое число без знака	np.dtype('u1') == np.uint8
'f'	Число с плавающей точкой	np.dtype('f8') == np.int64
'c'	Комплексное число с плавающей точкой	np.dtype('c16') == np.complex128
'S', 'a'	Строка	np.dtype('S5')
'U'	Строка в кодировке Unicode	np.dtype('U') == np.str_
'V'	Неформатированные данные (тип void)	np.dtype('V') == np.void

Более продвинутые типы данных

Можно описывать и еще более продвинутые типы данных. Например, можно создать тип, в котором каждый элемент содержит массив или матрицу значений. В следующем примере создается тип данных, включающий компонент `mat`, содержащий матрицу 3 × 3 значений с плавающей точкой:

```
In [14]: tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
         X = np.zeros(1, dtype=tp)
         print(X[0])
         print(X['mat'][0])
Out[14]: (0, [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
         [[0. 0. 0.]
          [0. 0. 0.]
          [0. 0. 0.]]
```

Теперь каждый элемент массива `X` состоит из целого числа `id` и матрицы 3 × 3. Почему такой массив предпочтительнее простого многомерного или, возможно, словаря Python? Дело в том, что `dtype` библиотеки NumPy напрямую соответствует описанию структуры на языке C, так что можно обращаться к содержащему этот массив буферу памяти непосредственно из требуемым образом написанной программы на языке C. Если вам понадобится написать на Python интерфейс к уже существующей библиотеке на языке C или Fortran, которая работает со структурированными данными, вероятно, структурированные массивы будут вам весьма полезны!

Массивы записей: структурированные массивы с дополнительными возможностями

Библиотека NumPy предоставляет класс `np.recarray`, практически идентичный только что описанным структурированным массивам, но с одной дополнительной возможностью: доступ к полям можно осуществлять как к атрибутам, а не только

как к ключам словаря. Как вы помните, ранее мы обращались к значениям возраста, используя такую строку кода:

```
In [15]: data['age']
Out[15]: array([25, 45, 37, 19], dtype=int32)
```

Если же представить наши данные как массив записей, то можно обращаться к этим данным с помощью чуть более короткого синтаксиса:

```
In [16]: data_rec = data.view(np.recarray)
         data_rec.age
Out[16]: array([25, 45, 37, 19], dtype=int32)
```

Недостаток такого подхода в том, что при доступе к полям массивов записей неизбежны дополнительные накладные расходы, даже при использовании того же синтаксиса. Это показывает следующий пример:

```
In [17]: %timeit data['age']
         %timeit data_rec['age']
         %timeit data_rec.age
Out[17]: 121 ns ± 1.4 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
         2.41 µs ± 15.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
         3.98 µs ± 20.5 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Имеет ли смысл жертвовать дополнительным временем ради более удобного синтаксиса — зависит от вашего приложения.

Вперед, к Pandas

Эта глава о структурированных массивах и массивах записей не случайно стоит в конце второй части — она удачно подводит нас к теме следующего пакета — библиотеки Pandas. В определенных случаях не помешает знать о существовании обсуждавшихся здесь структурированных массивов, особенно если вам нужно, чтобы массивы библиотеки NumPy соответствовали двоичным форматам данных на C, Fortran или другом языке программирования. Для повседневной работы со структурированными данными намного удобнее использовать пакет Pandas, который мы подробно рассмотрим в следующей главе.

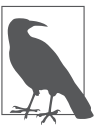
ЧАСТЬ III

Манипуляции над данными с помощью пакета Pandas

В части II мы рассмотрели библиотеку NumPy и ее объект `ndarray`, обеспечивающий эффективное хранение плотных массивов и манипуляций с ними в Python. В этой главе мы, основываясь на этих знаниях, детально ознакомимся со структурами данных библиотеки Pandas. Pandas — более новый пакет, надстройка над библиотекой NumPy, предлагающий эффективную реализацию класса `DataFrame`. Объекты `DataFrame` — многомерные массивы с метками для строк и столбцов, а также зачастую с неоднородными типами данных и/или отсутствующими данными. Помимо удобного интерфейса для хранения маркированных данных, библиотека Pandas реализует множество операций для работы с данными хорошо знакомых пользователям фреймворков баз данных и электронных таблиц.

Структура данных `ndarray` библиотеки NumPy предоставляет все необходимые возможности для работы с хорошо упорядоченными данными в задачах численных вычислений. Для этой цели библиотека NumPy отлично подходит, однако имеет свои ограничения, которые становятся заметными, чуть только нам потребуется немного больше гибкости (маркирование данных, работа с пропущенными данными и т. д.). Эти ограничения проявляются также при попытках выполнения операций, не подходящих для поэлементного транслирования (группировки, создание сводных таблиц и т. д.). Такие операции являются важной частью анализа данных с меньшей степенью структурированности, содержащихся во многих формах окружающего мира. Библиотека Pandas, особенно ее объекты `Series` и `DataFrame`, основана на структурах массивов NumPy и обеспечивает эффективную работу над подобными задачами «очистки данных».

В этой части мы сосредоточимся на стандартных приемах использования объектов `Series`, `DataFrame` и связанных с ними структур. По мере возможности мы будем применять взятые из реальных наборов данных примеры, но они не являются нашей целью.



Для установки пакета Pandas необходимо, чтобы в системе был установлен пакет NumPy, а если вы выполняете сборку библиотеки из исходного кода, то и соответствующие утилиты для компиляции исходных кодов на языках C и Cython, из которых состоит Pandas. Подробные инструкции по установке можно найти в документации пакета Pandas (http://pandas.pydata.org/). Если же вы последовали совету из предисловия и воспользовались стеком Anaconda, то пакет Pandas у вас уже установлен.

После установки пакета Pandas можно импортировать его и проверить версию:

```
In [1]: import pandas
        pandas.__version__
Out[1]: '1.3.5'
```

Аналогично тому, как мы импортировали пакет NumPy под псевдонимом np, пакет Pandas импортируем под псевдонимом pd:

```
In[2]: import pandas as pd
```

Мы будем использовать эти условные обозначения для импорта далее в книге.

НАПОМИНАНИЕ О ВСТРОЕННОЙ ДОКУМЕНТАЦИИ

Оболочка IPython предоставляет возможность быстро просматривать содержимое пакетов (с помощью клавиши Tab), а также документацию по различным функциям (используя символ ?). Загляните в раздел «Справка и документация в IPython» главы 1, если вам нужно освежить в памяти эти возможности.

Для отображения всего содержимого пространства имен numpy можете ввести следующую команду:

```
In [3]: np.<TAB>
```

Для отображения встроенной документации пакета NumPy используйте команду:

```
In [4]: np?
```

Более подробную документацию, а также руководства и другие источники информации можно найти на сайте http://pandas.pydata.org.

ГЛАВА 13

Знакомство с объектами библиотеки Pandas

Упрощенно объекты библиотеки Pandas можно считать расширенными версиями структурированных массивов NumPy, в которых строки и столбцы идентифицируются метками, а не простыми числовыми индексами. Библиотека Pandas предоставляет множество полезных инструментов, методов и функций в дополнение к базовым структурам данных, и практически все последующие обсуждения требуют понимания этих базовых структур. Позвольте познакомить вас с тремя фундаментальными структурами данных библиотеки Pandas: классами `Series`, `DataFrame` и `Index`.

Начнем сеанс программирования с импортирования библиотек NumPy и Pandas:

```
In[1]: import numpy as np
       import pandas as pd
```

Объект Series

Объект `Series` — одномерный массив индексированных данных. Вот как можно создать его из списка или массива:

```
In [2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
        data
Out[2]: 0    0.25
        1    0.50
        2    0.75
        3    1.00
        dtype: float64
```

Объект `Series` объединяет последовательность значений с явно заданной последовательностью индексов, к которым можно получить доступ посредством атрибутов `values` и `index`. Атрибут `values` — это уже знакомый нам массив NumPy:

```
In [3]: data.values
Out[3]: array([0.25, 0.5 , 0.75, 1.  ])
```

`index` — это объект типа `pd.Index`, подобный массиву, который мы рассмотрим подробнее далее:

```
In [4]: data.index
Out[4]: RangeIndex(start=0, stop=4, step=1)
```

К данным в `Series`, как и к данным в массиве NumPy, можно обращаться по индексам, используя нотацию с квадратными скобками, как это принято в языке Python:

```
In [5]: data[1]
Out[5]: 0.5

In [6]: data[1:3]
Out[6]: 1    0.50
        2    0.75
        dtype: float64
```

Однако объект `Series` намного универсальнее и гибче, чем эмулируемый им одномерный массив NumPy.

Объект Series как обобщенный массив NumPy

Может показаться, что объект `Series` и одномерный массив NumPy взаимозаменяемы. Основное различие между ними — индекс. В отличие от массивов NumPy, поддерживающих *неявные* целочисленные индексы, объекты `Series` в библиотеке Pandas используют *явно определяемые* индексы, присваиваемые значениям.

Явное определение индексов расширяет возможности объекта `Series`. Такой индекс может быть не только целым числом, но и значением любого желаемого типа. Например, при желании можно использовать в качестве индекса строковые значения:

```
In [7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=['a', 'b', 'c', 'd'])
        data
Out[7]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64
```

При этом доступ к элементам осуществляется как обычно:

```
In [8]: data['b']
Out[8]: 0.5
```

Можно применять даже индексы, состоящие из несмежных или непоследовательных значений:

```
In [9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=[2, 5, 3, 7])
        data
Out[9]: 2    0.25
        5    0.50
        3    0.75
        7    1.00
        dtype: float64

In [10]: data[5]
Out[10]: 0.5
```

Объект Series как специализированный словарь

Объект `Series` можно рассматривать как специализированную разновидность словаря Python. Словарь — структура, отображающая произвольные ключи в набор произвольных значений, а объект `Series` — структура, отображающая типизированные ключи в набор типизированных значений. Типизация важна: точно так же, как соответствующий типу специализированный код реализации массивов NumPy делает их намного эффективнее стандартных списков Python, информация о типах в объекте `Series` делает его намного более эффективным для определенных операций, чем словари Python.

Аналогию «объект Series как словарь» можно сделать еще более наглядной, сконструировав объект `Series` непосредственно из словаря Python. Вот пример с пятью наиболее населенными штатами в США по результатам переписи 2020 года:

```
In [11]: population_dict = {'California': 39538223, 'Texas': 29145505,
                            'Florida': 21538187, 'New York': 20201249,
                            'Pennsylvania': 13002700}
         population = pd.Series(population_dict)
         population
Out[11]: California      39538223
         Texas           29145505
         Florida         21538187
         New York        20201249
         Pennsylvania    13002700
         dtype: int64
```

Доступ к элементам может осуществляться с применением обычного синтаксиса словарей:

```
In [12]: population['California']
Out[12]: 39538223
```

Однако, в отличие от словаря, объект `Series` поддерживает характерные для массивов операции, такие как срезы:

```
In [13]: population['California':'Florida']
Out[13]: California    39538223
         Texas         29145505
         Florida       21538187
         dtype: int64
```

Мы рассмотрим некоторые нюансы индексации и срезов в библиотеке Pandas в главе 14.

Создание объектов Series

Мы уже изучили несколько способов создания объектов `Series` с нуля. Все они представляют собой различные варианты следующего синтаксиса:

```
pd.Series(data, index=index)
```

где `index` — необязательный аргумент, а `data` может быть одной из множества сущностей.

Например, аргумент `data` может быть списком или массивом NumPy. В этом случае `index` по умолчанию будет целочисленной последовательностью:

```
In [14]: pd.Series([2, 4, 6])
Out[14]: 0    2
         1    4
         2    6
         dtype: int64
```

Аргумент `data` может быть скалярным значением, которое будет повторено нужное количество раз для заполнения заданного индекса:

```
In [15]: pd.Series(5, index=[100, 200, 300])
Out[15]: 100    5
         200    5
         300    5
         dtype: int64
```

Аргумент `data` может быть словарем, в котором `index` по умолчанию получает значения его ключей:

```
In [16]: pd.Series({2:'a', 1:'b', 3:'c'})
Out[16]: 2    a
         1    b
         3    c
         dtype: object
```

В каждом случае индекс можно указать вручную, если необходимо получить другой результат:

```
In [17]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[1, 2])
Out[17]: 1    b
         2    a
         dtype: object
```

Объект DataFrame

Следующая фундаментальная структура в библиотеке Pandas — `DataFrame`. Как и объект `Series`, обсуждавшийся в предыдущем разделе, объект `DataFrame` можно рассматривать как обобщение массива NumPy или как специализированную версию словаря Python. Рассмотрим оба варианта.

DataFrame как обобщенный массив NumPy

Если объект `Series` — аналог одномерного массива с гибкой индексацией, то объект `DataFrame` — аналог двумерного массива с явно определяемыми индексами строк и столбцов. Двумерный массив можно рассматривать как упорядоченную последовательность согласованных столбцов, объект `DataFrame` — как упорядоченную последовательность согласованных объектов `Series`. Под «согласованными» имеется в виду то, что они используют один и тот же индекс.

Для демонстрации сначала создадим новый объект `Series`, содержащий площадь каждого из пяти упомянутых в предыдущем разделе штатов:

```
In [18]: area_dict = {'California': 423967, 'Texas': 695662, 'Florida': 170312,
                      'New York': 141297, 'Pennsylvania': 119280}
         area = pd.Series(area_dict)
         area
Out[18]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         dtype: int64
```

Воспользовавшись объектом `population` типа `Series`, сконструируем на основе словаря единый двумерный объект, содержащий всю эту информацию:

```
In [19]: states = pd.DataFrame({'population': population,
                                'area': area})
         states
```

```
Out[19]:               population    area
         California      39538223  423967
         Texas           29145505  695662
         Florida         21538187  170312
         New York        20201249  141297
         Pennsylvania    13002700  119280
```

Подобно объектам `Series`, объекты `DataFrame` имеют атрибут `index`, обеспечивающий доступ к меткам индекса:

```
In [20]: states.index
Out[20]: Index(['California', 'Texas', 'Florida', 'New York', 'Pennsylvania'],
          > dtype='object')
```

Помимо этого, у объекта `DataFrame` есть атрибут `columns` — объект `Index`, содержащий метки столбцов:

```
In [21]: states.columns
Out[21]: Index(['population', 'area'], dtype='object')
```

Таким образом, объект `DataFrame` можно рассматривать как обобщение двумерного массива NumPy, строки и столбцы которого имеют обобщенные индексы для доступа к данным.

Объект DataFrame как специализированный словарь

`DataFrame` можно рассматривать как специализированный словарь. Если словарь задает соответствие ключей значениям, то `DataFrame` — имени столбца объекту `Series` с данными этого столбца. Например, запрос данных по атрибуту `'area'` приведет к тому, что будет возвращен объект `Series`, содержащий уже виденные нами ранее площади штатов:

```
In [22]: states['area']
Out[22]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         Name: area, dtype: int64
```

Обратите внимание на возможный источник путаницы: в двумерном массиве NumPy обращение `data[0]` возвращает первую строку. По этой причине объекты `DataFrame` лучше рассматривать как обобщенные словари, а не обобщенные массивы, хотя обе точки зрения имеют право на жизнь. Мы изучим более гибкие средства индексации в объектах `DataFrame` в главе 14.

Создание объектов DataFrame

Существует множество способов создания объектов `DataFrame`. Далее описывается несколько примеров.

Из одного объекта Series

Объект `DataFrame` — это набор объектов `Series`. `DataFrame`, состоящий из одного столбца, можно создать на основе одного объекта `Series`:

```
In [23]: pd.DataFrame(population, columns=['population'])
Out[23]:             population
           California   39538223
                Texas   29145505
              Florida   21538187
             New York   20201249
         Pennsylvania   13002700
```

Из списка словарей

Любой список словарей можно преобразовать в объект `DataFrame`. Мы воспользуемся простым генератором списков для создания данных:

```
In [24]: data = [{'a': i, 'b': 2 * i}
                 for i in range(3)]
         pd.DataFrame(data)
Out[24]:    a  b
         0  0  0
         1  1  2
         2  2  4
```

Даже если некоторые ключи в словаре отсутствуют, библиотека Pandas просто заполнит их значениями `NaN` (то есть Not a number — «не является числом»; см. главу 16):

```
In [25]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
Out[25]:      a  b    c
         0  1.0  2  NaN
         1  NaN  3  4.0
```

Из словаря объектов Series

Объект `DataFrame` также можно создать на основе словаря объектов `Series`:

```
In [26]: pd.DataFrame({'population': population,
                       'area': area})
Out[26]:              population    area
         California      39538223  423967
         Texas           29145505  695662
```

```
        Florida         21538187  170312
        New York        20201249  141297
        Pennsylvania    13002700  119280
```

Из двумерного массива NumPy

Если у нас есть двумерный массив данных, мы можем создать объект `DataFrame` с любыми заданными именами столбцов и индексов. Для всех отсутствующих значений будут использоваться целочисленные индексы:

```
In [27]: pd.DataFrame(np.random.rand(3, 2),
                      columns=['foo', 'bar'],
                      index=['a', 'b', 'c'])
Out[27]:         foo       bar
         a  0.471098  0.317396
         b  0.614766  0.305971
         c  0.533596  0.512377
```

Из структурированного массива NumPy

Мы рассматривали структурированные массивы в главе 12. Объект `DataFrame` ведет себя во многом подобно структурированному массиву и может быть создан непосредственно из него:

```
In [28]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
         A
Out[28]: array([(0, 0.), (0, 0.), (0, 0.)], dtype=[('A', '<i8'), ('B', '<f8')])
In [29]: pd.DataFrame(A)
Out[29]:    A    B
         0  0  0.0
         1  0  0.0
         2  0  0.0
```

Объект Index

Как объект `Series`, так и объект `DataFrame` содержат явный *индекс*, обеспечивающий возможность ссылаться на данные и модифицировать их. Объект `Index` можно рассматривать как *неизменяемый массив* (immutable array) или как *упорядоченное множество* (ordered set) — формально мультимножество, так как объекты `Index` могут содержать повторяющиеся значения. Из этих способов его представления следуют некоторые интересные возможности операций над объектами `Index`. В качестве простого примера создадим `Index` из списка целых чисел:

```
In [30]: ind = pd.Index([2, 3, 5, 7, 11])
         ind
Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Объект Index как неизменяемый массив

Объект `Index` во многом ведет себя подобно массиву. Например, для извлечения из него значений или срезов можно использовать стандартную для Python нотацию индексации:

```
In [31]: ind[1]
Out[31]: 3

In [32]: ind[::2]
Out[32]: Int64Index([2, 5, 11], dtype='int64')
```

У объектов `Index` есть много атрибутов, знакомых нам по массивам NumPy:

```
In [33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)
Out[33]: 5 (5,) 1 int64
```

Одно из различий между объектами `Index` и массивами NumPy — неизменяемость индексов, то есть их нельзя модифицировать стандартными средствами:

```
In [34]: ind[1] = 0
TypeError: Index does not support mutable operations
```

Неизменяемость делает безопаснее совместное использование индексов несколькими объектами `DataFrame` и массивами, исключая возможность побочных эффектов в виде случайной модификации индекса по неосторожности.

Index как упорядоченное множество

Объекты библиотеки Pandas спроектированы с прицелом на упрощение таких операций, как соединение наборов данных, зависящее от многих аспектов арифметики множеств. Объект `Index` следует большинству соглашений, используемых встроенной структурой данных `set` языка Python, так что объединение, пересечение, разность и другие операции над множествами можно выполнять привычным образом:

```
In [35]: indA = pd.Index([1, 3, 5, 7, 9])
         indB = pd.Index([2, 3, 5, 7, 11])

In [36]: indA.intersection(indB)
Out[36]: Int64Index([3, 5, 7], dtype='int64')

In [37]: indA.union(indB)
Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In [38]: indA.symmetric_difference(indB)
Out[38]: Int64Index([1, 2, 9, 11], dtype='int64')
```

ГЛАВА 14

Индексация и выборка данных

В части II мы подробно рассмотрели методы и инструменты доступа к значениям в массивах NumPy и их изменения, в том числе: индексацию (`arr[2, 1]`), срезы (`arr[:,1:5]`), маскирование (`arr[arr > 0]`), «прихотливую» индексацию (`arr[0,[1,5]]`), а также их комбинации (`arr[:,[1,5]]`). Здесь мы изучим аналогичные средства доступа к значениям в объектах `Series` и `DataFrame` и их изменения. Если вы использовали паттерны библиотеки NumPy, то соответствующие паттерны библиотеки Pandas будут для вас привычны.

Начнем с простого случая одномерного объекта `Series`, после чего перейдем к более сложному двумерному объекту `DataFrame`.

Выборка данных из объекта Series

Объект `Series` во многом ведет себя подобно одномерному массиву NumPy и стандартному словарю Python. Если вы освоили эти аналогии, то это поможет вам лучше понять паттерны индексации и выборки данных из этих массивов.

Объект Series как словарь

Объект `Series` задает соответствие набора ключей набору значений аналогично словарю:

```
In [1]: import pandas as pd
        data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=['a', 'b', 'c', 'd'])
        data
Out[1]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64
In [2]: data['b']
Out[2]: 0.5
```

Для просмотра ключей/индексов и значений выражения можно также использовать методы языка Python, которые применяются к словарям:

```
In [3]: 'a' in data
Out[3]: True

In [4]: data.keys()
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [5]: list(data.items())
Out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

Объекты `Series` можно модифицировать с помощью синтаксиса, подобного синтаксису для словарей. Аналогично расширению словаря путем присваивания значения для нового ключа можно расширить объект `Series`, присвоив значение для нового значения индекса:

```
In [6]: data['e'] = 1.25
        data
Out[6]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        e    1.25
        dtype: float64
```

Такая простота изменения объектов — удобная возможность: библиотека Pandas сама, незаметно для нас, принимает решения о размещении в памяти и необходимости копирования данных. Пользователю, как правило, не приходится заботиться о подобных вопросах.

Объект Series как одномерный массив

Объект `Series`, основываясь на интерфейсе, напоминающем словарь, предоставляет возможность выборки элементов с помощью тех же базовых механизмов, что и для массивов NumPy, то есть срезов, маскирования и «прихотливой» индексации. Вот несколько примеров:

```
In [7]: # получение среза с явными индексами
        data['a':'c']
Out[7]: a    0.25
        b    0.50
        c    0.75
        dtype: float64

In [8]: # получение среза с неявными целочисленными индексами
        data[0:2]
Out[8]: a    0.25
        b    0.50
        dtype: float64
```

```
In [9]: # маскирование
        data[(data > 0.3) & (data < 0.8)]
Out[9]: b    0.50
        c    0.75
        dtype: float64

In [10]: # "прихотливая" индексация
         data[['a', 'e']]
Out[10]: a    0.25
         e    1.25
         dtype: float64
```

Наибольшие затруднения из этих операций могут вызвать срезы. Обратите внимание, что при выполнении среза с помощью явного индекса (`data['a':'c']`) значение, соответствующее последнему индексу, *включается* в срез, а при срезе неявным индексом (`data[0:2]`) — *не включается*.

Индексаторы: loc и iloc

Подобные обозначения для срезов и индексации могут привести к путанице. Например, при наличии у объекта Series явного целочисленного индекса операция индексации (`data[1]`) будет использовать явные индексы, а операция среза (`data[1:3]`) — неявный индекс в стиле языка Python.

```
In [11]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data
Out[11]: 1    a
         3    b
         5    c
         dtype: object

In [12]: # использование явного индекса при индексации
         data[1]
Out[12]: 'a'

In [13]: # использование неявного индекса при срезе
         data[1:3]
Out[13]: 3    b
         5    c
         dtype: object
```

Из-за этой потенциальной путаницы в случае целочисленных индексов в библиотеке Pandas предусмотрены специальные *атрибуты-индексаторы*, позволяющие явным образом применять определенные схемы индексации. Они являются не функциональными методами, а именно атрибутами, предоставляющими для данных из объекта `Series` конкретный интерфейс для выполнения срезов.

Во-первых, атрибут `loc` позволяет выполнить индексацию и получить срезы с использованием явного индекса:

```
In [14]: data.loc[1]
Out[14]: 'a'

In [15]: data.loc[1:3]
Out[15]: 1    a
         3    b
         dtype: object
```

Атрибут `iloc` дает возможность выполнять индексацию и получать срезы, применяя неявный индекс в стиле языка Python:

```
In [16]: data.iloc[1]
Out[16]: 'b'
In [17]: data.iloc[1:3]
Out[17]: 3    b
         5    c
         dtype: object
```

Один из руководящих принципов программирования на языке Python — «явное лучше неявного». То, что атрибуты `loc` и `iloc` по своей природе явные, делает их очень удобными для обеспечения ясности и удобочитаемости кода. Я рекомендую использовать оба, особенно в случае целочисленных индексов, чтобы сделать код более простым для чтения и понимания и избежать случайных малозаметных ошибок при обозначении индексации и срезов.

Выборка данных из объекта DataFrame

Объект `DataFrame` во многом ведет себя подобно двумерному или структурированному массиву, а также словарю объектов `Series` с общим индексом. Эти аналогии следует иметь в виду при изучении способов выборки данных из объекта.

Объект DataFrame как словарь

Первая аналогия, которую мы обсудим, — объект `DataFrame` как словарь схожих между собой объектов `Series`. Вернемся к примеру про площадь и численность населения штатов:

```
In [18]: area = pd.Series({'California': 423967, 'Texas': 695662,
                           'Florida': 170312, 'New York': 141297,
                           'Pennsylvania': 119280})
         pop = pd.Series({'California': 39538223, 'Texas': 29145505,
                          'Florida': 21538187, 'New York': 20201249,
                          'Pennsylvania': 13002700})
```

```
        data = pd.DataFrame({'area':area, 'pop':pop})
        data
Out[18]:                area       pop
        California    423967  39538223
        Texas         695662  29145505
        Florida       170312  21538187
        New York      141297  20201249
        Pennsylvania  119280  13002700
```

К отдельным объектам `Series`, составляющим столбцы объекта `DataFrame`, можно обращаться посредством такой же индексации, как и для словарей, — по имени столбца:

```
In [19]: data['area']
Out[19]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         Name: area, dtype: int64
```

Можно обращаться к данным и с помощью атрибутов, используя их как строковые имена столбцов:

```
In [20]: data.area
Out[20]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         Name: area, dtype: int64
```

Хотя это и удобное сокращение, не забывайте, что оно работает не всегда! Например, если имена столбцов — не строки или имена столбцов конфликтуют с методами объекта `DataFrame`, доступ по именам атрибутов невозможен. Допустим у объекта `DataFrame` есть метод `pop`, тогда выражение `data.pop` будет обозначать его, а не столбец `"pop"`:

```
In [21]: data.pop is data["pop"]
Out[21]: False
```

Не поддавайтесь искушению присваивать значения столбцов посредством атрибутов. Лучше использовать выражение `data['pop'] = z` вместо `data.pop = z`.

Как и в случае с обсуждавшимися ранее объектами `Series`, такой «словарный» синтаксис можно применять для модификации объекта, например добавления еще одного столбца:

```
In [22]: data['density'] = data['pop'] / data['area']
         data
```

```
Out[22]:                 area       pop     density
        California    423967  39538223   93.257784
        Texas         695662  29145505   41.896072
        Florida       170312  21538187  126.463121
        New York      141297  20201249  142.970120
        Pennsylvania  119280  13002700  109.009893
```

Приведенный пример демонстрирует простоту синтаксиса поэлементных операций над объектами `Series`. Этот вопрос мы изучим подробнее в главе 15.

Объект DataFrame как двумерный массив

Объект `DataFrame` можно рассматривать как двумерный массив с расширенными возможностями. Взглянем на содержимое исходного массива с помощью атрибута `values`:

```
In [23]: data.values
Out[23]: array([[4.23967000e+05, 3.95382230e+07, 9.32577842e+01],
               [6.95662000e+05, 2.91455050e+07, 4.18960717e+01],
               [1.70312000e+05, 2.15381870e+07, 1.26463121e+02],
               [1.41297000e+05, 2.02012490e+07, 1.42970120e+02],
               [1.19280000e+05, 1.30027000e+07, 1.09009893e+02]])
```

С объектом `DataFrame` можно выполнить множество привычных для массивов действий. Например, транспонировать весь `DataFrame`, поменяв местами строки и столбцы:

```
In [24]: data.T
Out[24]:           California         Texas       Florida      New York  Pennsylvania
        area     4.239670e+05  6.956620e+05  1.703120e+05  1.412970e+05  1.192800e+05
        pop      3.953822e+07  2.914550e+07  2.153819e+07  2.020125e+07  1.300270e+07
        density  9.325778e+01  4.189607e+01  1.264631e+02  1.429701e+02  1.090099e+02
```

Однако, когда речь заходит об индексации объектов `DataFrame`, становится ясно, что словарная индексация мешает нам рассматривать их просто как массивы NumPy. В частности, единственный индекс означает доступ к строке:

```
In [25]: data.values[0]
Out[25]: array([4.23967000e+05, 3.95382230e+07, 9.32577842e+01])
```

а единственный «индекс» (в кавычках) — доступ к столбцу в `DataFrame`:

```
In [26]: data['area']
Out[26]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         Name: area, dtype: int64
```

Соответственно для индексации в стиле обычных массивов необходимо использовать еще одно соглашение. Библиотека Pandas применяет упомянутые ранее индексаторы `loc` и `iloc`. С помощью индексатора `iloc` можно индексировать исходный массив, как будто это простой массив NumPy (используя неявный синтаксис языка Python), но с сохранением в результирующих данных меток объекта `DataFrame` для индекса и столбцов:

```
In [27]: data.iloc[:3, :2]
Out[27]:               area       pop
        California  423967  39538223
        Texas       695662  29145505
        Florida     170312  21538187
```

Аналогично, используя индексатор `loc`, можно обращаться к данным в привычном стиле работы с массивами, но используя явные индексы и имена столбцов:

```
In [28]: data.loc[:'Florida', :'pop']
Out[28]:               area       pop
        California  423967  39538223
        Texas       695662  29145505
        Florida     170312  21538187
```

В этих индексаторах можно использовать все уже знакомые вам паттерны доступа к данным в стиле библиотеки NumPy. Например, в индексаторе `loc` можно сочетать маскирование и «прихотливую» индексацию:

```
In [29]: data.loc[data.density > 120, ['pop', 'density']]
Out[29]:                pop     density
        Florida   21538187  126.463121
        New York  20201249  142.970120
```

Любой такой синтаксис индексации можно применять для задания или изменения значений. Это выполняется обычным, уже привычным вам по работе с библиотекой NumPy способом:

```
In [30]: data.iloc[0, 2] = 90
        data
Out[30]:                 area       pop     density
        California    423967  39538223   90.000000
        Texas         695662  29145505   41.896072
        Florida       170312  21538187  126.463121
        New York      141297  20201249  142.970120
        Pennsylvania  119280  13002700  109.009893
```

Чтобы достичь уверенности при манипуляции данными с помощью библиотеки Pandas, потратьте немного времени и поэкспериментируйте с простым объектом `DataFrame` и приемами индексации, получения срезов, маскирования и «прихотливой» индексации.

Дополнительный синтаксис для индексации

Существует еще несколько вариантов синтаксиса для индексации, казалось бы, плохо согласующихся с обсуждавшимся ранее, но очень удобных на практике. Во-первых, если *индексация* относится к столбцам, то *срезы* относятся к строкам:

```
In [31]: data['Florida':'New York']
Out[31]:             area       pop     density
         Florida   170312  21538187  126.463121
         New York  141297  20201249  142.970120
```

Извлекая срезы, можно также ссылаться на строки по номеру, а не по индексу:

```
In [32]: data[1:3]
Out[32]:            area       pop     density
         Texas    695662  29145505   41.896072
         Florida  170312  21538187  126.463121
```

Непосредственные операции маскирования также интерпретируются построчно, а не по столбцам:

```
In [33]: data[data.density > 120]
Out[33]:             area       pop     density
         Florida   170312  21538187  126.463121
         New York  141297  20201249  142.970120
```

Эти два варианта синтаксически подобны таковым для массивов NumPy, и они, возможно, хотя и не вполне вписываются в шаблоны синтаксиса библиотеки Pandas, но весьма удобны на практике.

ГЛАВА 15

Операции над данными в библиотеке Pandas

Одна из важнейших особенностей библиотеки NumPy — способность выполнять быстрые поэлементные операции — как простейшие арифметические (сложение, вычитание, умножение и т. д.), так и более сложные (вычисление тригонометрических, показательных и логарифмических функций и т. п.). Библиотека Pandas наследует от NumPy немалую часть этой функциональности, и ключ к ее использованию — универсальные функции, с которыми мы познакомились в главе 6.

Однако библиотека Pandas включает несколько полезных трюков: для унарных операций, например изменения знака и тригонометрических функций, при использовании ее универсальных функций в результате будут *сохранены индекс и метки столбцов*, а для бинарных операций, например сложения и умножения, библиотека Pandas автоматически совместит индексы при передаче объектов универсальной функции. Это значит, что сохранение контекста данных и объединение данных из различных источников — две задачи, потенциально чреватые ошибками при работе с исходными массивами NumPy, — оказываются надежно защищенными от ошибок благодаря библиотеке Pandas. Кроме того, в библиотеке имеются операции между одномерными структурами объектов `Series` и двумерными структурами объектов `DataFrame`.

Универсальные функции: сохранение индекса

В силу того, что библиотека Pandas предназначена для работы с библиотекой NumPy, все универсальные функции библиотеки NumPy будут работать с объектами `Series` и `DataFrame` библиотеки Pandas. Для начала давайте создадим простые объекты `Series` и `DataFrame`, которые будем использовать для демонстрации:

```
In [1]: import pandas as pd
        import numpy as np
```

```
In [2]: rng = np.random.default_rng(42)
        ser = pd.Series(rng.integers(0, 10, 4))
        ser
Out[2]: 0    0
        1    7
        2    6
        3    4
        dtype: int64

In [3]: df = pd.DataFrame(rng.integers(0, 10, (3, 4)),
                          columns=['A', 'B', 'C', 'D'])
        df
Out[3]:    A  B  C  D
        0  4  8  0  6
        1  2  0  5  9
        2  7  7  7  7
```

Если применить универсальную функцию NumPy к любому из этих объектов, результатом будет другой объект библиотеки Pandas с *сохранением индексов*:

```
In [4]: np.exp(ser)
Out[4]: 0       1.000000
        1    1096.633158
        2     403.428793
        3      54.598150
        dtype: float64
```

Как и в случае более сложных вычислений:

```
In [5]: np.sin(df * np.pi / 4)
Out[5]:               A             B         C         D
        0  1.224647e-16 -2.449294e-16  0.000000 -1.000000
        1  1.000000e+00  0.000000e+00 -0.707107  0.707107
        2 -7.071068e-01 -7.071068e-01 -0.707107 -0.707107
```

Все описанные в главе 6 универсальные функции можно использовать аналогично вышеприведенным.

Универсальные функции: согласование индексов

В бинарных операциях над двумя объектами `Series` или `DataFrame` библиотека Pandas будет согласовывать индексы в процессе их выполнения. Это очень удобно при работе с неполными данными.

Согласование индексов в объектах Series

Допустим, мы объединили два различных источника данных, чтобы найти три штата США с наибольшей *площадью* и три штата США с наибольшей *численностью населения*:

```
In [6]: area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                          'California': 423967}, name='area')
        population = pd.Series({'California': 39538223, 'Texas': 29145505,
                                'Florida': 21538187}, name='population')
```

Посмотрим, что получится, если разделить второй результат на первый для вычисления плотности населения:

```
In [7]: population / area
Out[7]: Alaska              NaN
        California    93.257784
        Florida             NaN
        Texas         41.896072
        dtype: float64
```

Получившийся в итоге массив содержит *объединение* индексов двух исходных массивов, которое можно определить посредством стандартной арифметики множеств:

```
In [8]: area.index.union(population.index)
Out[8]: Index(['Alaska', 'California', 'Florida', 'Texas'], dtype='object')
```

Ни один из относящихся к ним обоим элементов не содержит значения `NaN` («нечисловое значение»), с помощью которого библиотека Pandas отмечает отсутствующие данные (см. дальнейшее обсуждение вопроса отсутствующих данных в главе 16). Аналогичным образом реализовано сопоставление индексов для всех встроенных арифметических выражений языка Python: все отсутствующие значения заполняются по умолчанию значением `NaN`:

```
In [9]: A = pd.Series([2, 4, 6], index=[0, 1, 2])
        B = pd.Series([1, 3, 5], index=[1, 2, 3])
        A + B
Out[9]: 0    NaN
        1    5.0
        2    9.0
        3    NaN
        dtype: float64
```

Если использование значений `NaN` нежелательно, его можно заменить другим, воспользовавшись соответствующими методами объекта вместо операторов. Например, вызов метода `A.add(B)` эквивалентен вызову `A + B`, но предоставляет возможность по желанию явно задать значения заполнителей для любых потенциально отсутствующих элементов в объектах `A` или `B`:

```
In [10]: A.add(B, fill_value=0)
Out[10]: 0    2.0
         1    5.0
         2    9.0
         3    5.0
         dtype: float64
```

Согласование индексов в объектах DataFrame

При выполнении операций над объектами `DataFrame` происходит аналогичное согласование *как* для столбцов, *так* и для индексов:

```
In [11]: A = pd.DataFrame(rng.integers(0, 20, (2, 2)),
                          columns=['a', 'b'])
         A
Out[11]:     a  b
         0  10  2
         1  16  9

In [12]: B = pd.DataFrame(rng.integers(0, 10, (3, 3)),
                          columns=['b', 'a', 'c'])
         B
Out[12]:    b  a  c
         0  5  3  1
         1  9  7  6
         2  4  8  5

In [13]: A + B
Out[12]:       a     b   c
         0  13.0   7.0 NaN
         1  23.0  18.0 NaN
         2   NaN   NaN NaN
```

Обратите внимание, что индексы согласуются правильно независимо от их расположения в двух объектах и индексы в полученном результате отсортированы. Как и в случае объектов `Series`, можно использовать соответствующие арифметические методы объектов и передавать для использования вместо отсутствующих значений любое нужное значение `fill_value`. В следующем примере мы заполним отсутствующие значения средним значением всех элементов объекта `A`:

```
In [14]: A.add(B, fill_value=A.values.mean())
Out[14]:        a      b      c
         0  13.00   7.00  10.25
         1  23.00  18.00  15.25
         2  17.25  13.25  14.25
```

В табл. 15.1 перечислены операторы языка Python и эквивалентные им методы объектов библиотеки Pandas.

Таблица 15.1. Операторы языка Python и эквивалентные им методы в библиотеке Pandas

Оператор языка Python	Метод(ы) библиотеки Pandas
`+`	`add`
`-`	`sub, subtract`
`*`	`mul, multiply`

Оператор языка Python	Метод(ы) библиотеки Pandas
`/`	`truediv, div, divide`
`//`	`floordiv`
`%`	`mod`
`**`	`pow`

Универсальные функции: операции между объектами DataFrame и Series

При выполнении операций между объектами `DataFrame` и `Series` согласование столбцов и индексов осуществляется аналогичным образом. Операции между объектами `DataFrame` и `Series` подобны операциям между двумерным и одномерным массивами NumPy. Рассмотрим одну из часто встречающихся операций — вычисление разности двумерного массива и одной из его строк:

```
In [15]: A = rng.integers(10, size=(3, 4))
         A
Out[15]: array([[4, 4, 2, 0],
                [5, 8, 0, 8],
                [8, 2, 6, 1]])

In [16]: A - A[0]
Out[16]: array([[ 0,  0,  0,  0],
                [ 1,  4, -2,  8],
                [ 4, -2,  4,  1]])
```

В соответствии с правилами транслирования библиотеки NumPy (глава 8) вычитание из двумерного массива одной из его строк выполняется построчно.

В библиотеке Pandas вычитание по умолчанию также происходит построчно:

```
In [17]: df = pd.DataFrame(A, columns=['Q', 'R', 'S', 'T'])
         df - df.iloc[0]
Out[17]:    Q  R  S  T
         0  0  0  0  0
         1  1  4 -2  8
         2  4 -2  4  1
```

Чтобы выполнить эту операцию по столбцам, воспользуйтесь упомянутыми выше методами объектов, передав именованный аргумент `axis`:

```
In [18]: df.subtract(df['R'], axis=0)
Out[18]:    Q  R  S  T
         0  0  0 -2 -4
         1 -3  0 -8  0
         2  6  0  4 -1
```

Обратите внимание, что операции `DataFrame`/`Series`, аналогично обсуждавшимся ранее операциям, будут автоматически выполнять согласование индексов между двумя их элементами:

```
In [19]: halfrow = df.iloc[0, ::2]
         halfrow
Out[19]: Q    4
         S    2
         Name: 0, dtype: int64

In [20]: df - halfrow
Out[20]:      Q   R    S   T
         0  0.0 NaN  0.0 NaN
         1  1.0 NaN -2.0 NaN
         2  4.0 NaN  4.0 NaN
```

Подобное сохранение и согласование индексов и столбцов означает, что операции над данными в библиотеке Pandas всегда сохраняют контекст данных, предотвращая возможные ошибки при работе с неоднородными и/или неправильно согласованными данными в исходных массивах NumPy.

ГЛАВА 16

Обработка отсутствующих данных

Реальные данные редко бывают очищенными и однородными. В частности, во многих интересных наборах данных некоторое количество данных отсутствует. Еще более затрудняет работу то, что в различных источниках данных отсутствующие данные могут отмечаться по-разному.

В этой главе мы обсудим общие соображения, касающиеся отсутствующих данных, рассмотрим способы представления их библиотекой Pandas и продемонстрируем встроенные инструменты Pandas для обработки отсутствующих данных. Здесь и далее мы будем называть отсутствующие данные *null*, *NaN* или *NA*-значениями.

Компромиссы при обозначении отсутствующих данных

Разработано несколько схем обозначения отсутствующих данных в таблицах или объектах `DataFrame`. Они основываются на одной из двух стратегий: использование *маски*, отмечающей глобально отсутствующие значения, или выбор специального *значения-индикатора* (sentinel value), обозначающего отсутствующее значение.

Маска может быть или совершенно отдельным булевым массивом, или включать выделение одного бита представления данных для локальной индикации отсутствия значения.

Значение-индикатор может быть особым условным обозначением, подходящим для конкретных данных, например указывать на отсутствующее целое число с помощью значения –9999 или какой-то редкой комбинации битов. Или же оно может быть более глобальным обозначением, например обозначать отсутствующее значение с плавающей точкой с помощью `NaN` — специального значения, включенного в спецификацию IEEE, определяющую формат представления чисел с плавающей точкой.

Каждый из подходов имеет свои достоинства и недостатки: использование отдельного массива-маски требует выделения памяти под дополнительный булев массив, что увеличивает накладные расходы в смысле как оперативной памяти, так и процессорного времени. Значение-индикатор сокращает диапазон доступных для представления допустимых значений и может потребовать выполнения дополнительной (зачастую неоптимизированной) логики при арифметических операциях на CPU и GPU. Общепринятые специальные значения, такие как `NaN`, доступны не для всех типов данных.

Как и в большинстве случаев, где нет универсального оптимального варианта, разные языки программирования и системы используют различные обозначения. Например, в качестве значений-индикаторов для отсутствующих данных в языке R используются зарезервированные комбинации битов. А система SciDB использует для индикации состояния NA дополнительный байт, присоединяемый к каждой ячейке.

Отсутствующие данные в Pandas

Способ обработки отсутствующих данных библиотекой Pandas определяется ее зависимостью от пакета NumPy, в котором отсутствует встроенное понятие NA-значений для всех типов данных, кроме данных с плавающей точкой.

Библиотека Pandas могла бы последовать примеру языка R и задать комбинации битов для каждого конкретного типа данных, соответствующие отсутствующим значениям, но этот подход слишком громоздкий. Ведь если в языке R насчитывается всего четыре базовых типа данных, то NumPy поддерживает их *намного* больше. Например, в языке R есть только один целочисленный тип, а библиотека NumPy поддерживает 14 простых целочисленных типов с учетом различной точности, знаковости/беззнаковости и порядка байтов. Резервирование специальной комбинации битов во всех доступных в библиотеке NumPy типах данных привело бы к громадным накладным расходам в разнообразных частных случаях операций для различных типов и, вероятно, потребовало бы даже отдельной ветви пакета NumPy. Кроме того, для небольших типов данных (например, 8-битных целых чисел) потеря одного бита на маску существенно сузит диапазон значений, которые сможет представлять этот тип.

По этой причине в библиотеке Pandas реализовано два «режима» хранения отсутствующих значений и манипуляций с ними:

- в режиме по умолчанию для обозначения отсутствующих данных используются индикаторы `NaN` и `None` в зависимости от типа данных;

- альтернативное решение заключается в выборе типов данных, поддерживающих пустое (`null`) значение (dtypes), которые предлагает Pandas (обсуждаются далее в этой главе), и создании сопутствующего массива масок для отслеживания отсутствующих значений. Эти отсутствующие значения затем представляются пользователю как специальное значение `pd.NA`.

В любом случае операции с данными, предоставляемые библиотекой Pandas, будут обрабатывать и распространять эти отсутствующие значения предсказуемым образом. Но чтобы лучше понять, *когда и почему* следует выбирать тот или иной вариант, кратко рассмотрим достоинства и недостатки, присущие `None`, `NaN` и `NA`. Как обычно, начнем с импорта NumPy и Pandas:

```
In [1]: import numpy as np
        import pandas as pd
```

None как значение-индикатор

В роли значения-индикатора для некоторых типов данных Pandas использует `None`. `None` — это объект Python, то есть любой массив, содержащий `None`, должен иметь `dtype=object` или, проще говоря, должен быть последовательностью объектов Python.

Например, посмотрите, что произойдет, если передать `None` в массив NumPy:

```
In [2]: vals1 = np.array([1, None, 2, 3])
        vals1
Out[2]: array([1, None, 2, 3], dtype=object)
```

`dtype=object` означает, что наилучший возможный вывод об общем типе элементов содержимого данного массива, который только смогла сделать библиотека NumPy, — то, что они все являются объектами Python. Хотя такая разновидность массивов полезна для определенных целей, все операции над ними будут выполняться на уровне языка Python, с накладными расходами, значительно превышающими расходы на выполнение быстрых операций над массивами с данными простых типов:

```
In [3]: %timeit np.arange(1E6, dtype=int).sum()
Out[3]: 2.73 ms ± 288 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [4]: %timeit np.arange(1E6, dtype=object).sum()
Out[4]: 92.1 ms ± 3.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Кроме того, поскольку Python не поддерживает арифметические операции с None, вызов функций агрегирования, таких как sum или min, обычно приводит к ошибке:

```
In [5]: vals1.sum()
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

По этой причине Pandas не использует None как значение-индикатор в числовых массивах.

NaN: отсутствующие числовые данные

Еще одно представление отсутствующих данных, NaN, представляет собой специальное значение с плавающей точкой, распознаваемое всеми системами, использующими стандартное IEEE-представление чисел с плавающей точкой:

```
In [6]: vals2 = np.array([1, np.nan, 3, 4])
        vals2
Out[6]: array([ 1., nan,  3.,  4.])
```

Обратите внимание, что библиотека NumPy выбрала для этого массива стандартный тип с плавающей точкой: это значит, что, в отличие от вышеупомянутого массива объектов, этот массив поддерживает быстрые операции, выполняемые скомпилированным кодом. Вы должны отдавать себе отчет, что значение NaN в чем-то подобно «вирусу данных»: оно «заражает» любой объект, к которому «прикасается».

Вне зависимости от операции результат арифметического действия с участием NaN будет равен NaN:

```
In [7]: 1 + np.nan
Out[7]: nan

In [8]: 0 * np.nan
Out[8]: nan
```

Это значит, что операция агрегирования значений определена (то есть ее результатом не будет ошибка), но не всегда приносит пользу:

```
In [9]: vals2.sum(), vals2.min(), vals2.max()
Out[9]: (nan, nan, nan)
```

Библиотека NumPy предоставляет специализированные агрегирующие функции, игнорирующие эти пропущенные значения:

```
In [10]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out[10]: (8.0, 1.0, 4.0)
```

Не забывайте, что значение NaN предназначено для представления отсутствующих значений с плавающей точкой, похожего аналога для целочисленных значений, строковых и других типов не существует.

Значения NaN и None в библиотеке Pandas

Оба значения, NaN и None, имеют свое предназначение, и библиотека Pandas делает их практически взаимозаменяемыми, преобразуя одно в другое в определенных случаях:

```
In [11]: pd.Series([1, np.nan, 2, None])
Out[11]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         dtype: float64
```

Библиотека Pandas автоматически выполняет преобразование при обнаружении NA-значений для тех типов, у которых отсутствует значение-индикатор. Например, если задать значение элемента целочисленного массива равным np.nan, для соответствия типу отсутствующего значения будет автоматически выполнено повышающее приведение типа этого массива к типу с плавающей точкой:

```
In [12]: x = pd.Series(range(2), dtype=int)
         x
Out[12]: 0    0
         1    1
         dtype: int64

In [13]: x[0] = None
         x
Out[13]: 0    NaN
         1    1.0
         dtype: float64
```

Обратите внимание, что, помимо приведения типа целочисленного массива к типу массива значений с плавающей точкой, библиотека Pandas автоматически преобразует значение None в NaN.

Хотя подобный подход со значениями-индикаторами/приведением типов, реализованный в библиотеке Pandas, может показаться несколько вычурным по сравнению с более унифицированным подходом к NA-значениям в таких предметно-ориентированных языках, как R, на практике он прекрасно работает и на моей памяти лишь изредка вызывал проблемы.

В табл. 16.1 перечислены правила повышающего приведения типов в библиотеке Pandas в случае наличия NA-значений.

Таблица 16.1. Правила повышающего приведения типов в библиотеке Pandas

Класс типов	Преобразование при хранении NA-значений	Значение-индикатор NA
С плавающей точкой	Без изменений	`np.nan`
Объект (`object`)	Без изменений	`None` или `np.nan`
Целое число	Приводится к float64	`np.nan`
Булево значение	Приводится к object	`None` или `np.nan`

Имейте в виду, что строковые данные в библиотеке Pandas всегда хранятся с типом данных (dtype) `object`.

Типы данных с поддержкой пустых значений в Pandas

В первых версиях библиотеки Pandas значения `None` и `NaN` были единственными значениями-индикаторами, используемыми для представления отсутствующих значений. Основная трудность, которая из-за этого возникала, заключалась в неявном приведении типов: например, не было никакого способа представить истинный целочисленный массив с отсутствующими данными.

Для решения этой проблемы позже в Pandas были добавлены типы *dtype с поддержкой пустых значений*, которые отличаются от обычных типов dtype именами, начинающимися с заглавных букв (например, `pd.Int32` вместо `np.int32`). Для обратной совместимости типы dtype с поддержкой пустых значений используются, только если их применение затребовано явно.

Например, вот объект `Series` с набором целочисленных данных, имеющих отсутствующие значения, созданный из списка, содержащего все три доступных индикатора отсутствующих данных:

```
In [14]: pd.Series([1, np.nan, 2, None, pd.NA], dtype='Int32')
Out[14]: 0       1
         1    <NA>
         2       2
         3    <NA>
         4    <NA>
         dtype: Int32
```

Это представление можно использовать взаимозаменяемо с другими во всех операциях, которые мы исследуем в оставшейся части этой главы.

Операции над пустыми значениями

Библиотека Pandas рассматривает значения None, NaN и NA как взаимозаменяемые индикаторы отсутствующих или пустых значений. Существует несколько удобных методов для обнаружения, удаления и замены пустых значений в структурах данных библиотеки Pandas, призванных упростить работу с ними.

- isnull — генерирует булеву маску для отсутствующих значений.
- notnull — противоположность метода isnull.
- dropna — возвращает отфильтрованный вариант данных.
- fillna — возвращает копию данных, в которой отсутствующие значения заполнены или восстановлены.

Завершим эту главу кратким знакомством и демонстрацией этих методов.

Выявление пустых значений

У структур данных в библиотеке Pandas имеются два удобных метода для выявления пустых значений: isnull и notnull. Каждый из них возвращает булеву маску для данных. Например:

```
In [15]: data = pd.Series([1, np.nan, 'hello', None])
In [16]: data.isnull()
Out[16]: 0    False
         1     True
         2    False
         3     True
         dtype: bool
```

Как упоминалось в главе 14, булевы маски можно использовать непосредственно в качестве индекса объектов Series и DataFrame:

```
In [17]: data[data.notnull()]
Out[17]: 0        1
         2    hello
         dtype: object
```

Аналогичные булевы результаты дают методы isnull и notnull объектов DataFrame.

Удаление пустых значений

Помимо продемонстрированного выше маскирования, существуют удобные методы: `dropna` (отбрасывающий NA-значения) и `fillna` (заполняющий NA-значения). Для объекта `Series` результат вполне однозначен:

```
In [18]: data.dropna()
Out[18]: 0        1
         2    hello
         dtype: object
```

Методы объектов `DataFrame` поддерживают несколько дополнительных параметров. Рассмотрим следующий объект `DataFrame`:

```
In [19]: df = pd.DataFrame([[1,      np.nan, 2],
                            [2,      3,      5],
                            [np.nan, 4,      6]])
         df
Out[19]:      0    1  2
         0  1.0  NaN  2
         1  2.0  3.0  5
         2  NaN  4.0  6
```

Из `DataFrame` нельзя выбросить отдельные значения, только целые строки или столбцы. В зависимости от приложения может понадобиться тот или иной вариант, поэтому функция `dropna` предоставляет для объектов `DataFrame` еще несколько параметров.

По умолчанию `dropna` отбрасывает все строки, в которых присутствует *хотя бы одно* пустое значение:

```
In [20]: df.dropna()
Out[20]:      0    1  2
         1  2.0  3.0  5
```

В качестве альтернативы можно отбрасывать NA-значения по разным осям: с параметром `axis=1` или `axis='columns'` функция `dropna` отбрасывает все столбцы, содержащие хотя бы одно пустое значение:

```
In [21]: df.dropna(axis='columns')
Out[21]:    2
         0  2
         1  5
         2  6
```

Однако при этом отбрасываются также и некоторые действительные данные. Возможно, вам захочется отбросить строки или столбцы, *все* значения (или большинство) в которых представлены значениями NA. Такое поведение можно задать с помощью параметров `how` и `thresh`, обеспечивающих точное управление допустимым количеством пустых значений.

По умолчанию `how='any'`, то есть отбрасываются все строки или столбцы (в зависимости от аргумента `axis`), содержащие хотя бы одно пустое значение. Можно также передать параметр `how='all'`, и в таком случае будут отбрасываться только строки/столбцы, *все* значения в которых пустые:

```
In [22]: df[3] = np.nan
         df
Out[22]:      0    1  2   3
         0  1.0  NaN  2 NaN
         1  2.0  3.0  5 NaN
         2  NaN  4.0  6 NaN

In [23]: df.dropna(axis='columns', how='all')
Out[23]:      0    1  2
         0  1.0  NaN  2
         1  2.0  3.0  5
         2  NaN  4.0  6
```

Для более точного управления можно задать в параметре `thresh` минимальное количество непустых значений, при котором строки/столбцы сохраняются:

```
In [24]: df.dropna(axis='rows', thresh=3)
Out[24]:      0    1  2   3
         1  2.0  3.0  5 NaN
```

В данном случае отбрасываются первая и последняя строки, поскольку в них содержится только по два непустых значения.

Заполнение пустых значений

Иногда предпочтительнее не отбрасывать пустые значения, а заполнять их каким-то допустимым значением. Это значение может быть фиксированным, например нулем, или интерполированным, или восстановленным на основе действительных данных. Это можно сделать, используя результат метода `isnull` в качестве маски. Но это настолько распространенная операция, что библиотека Pandas предоставляет метод `fillna`, возвращающий копию массива с замененными пустыми значениями.

Рассмотрим следующий объект `Series`:

```
In [25]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'),
                          dtype='Int32')
         data
Out[25]: a       1
         b    <NA>
         c       2
         d    <NA>
         e       3
         dtype: Int32
```

Можно заменить NA-элементы одним фиксированным значением, например нулями:

```
In [26]: data.fillna(0)
Out[26]: a    1
         b    0
         c    2
         d    0
         e    3
         dtype: Int32
```

Можно потребовать заменить любое отсутствующее значение предыдущим действительным значением:

```
In [27]: # заменить предыдущим действительным значением
         data.fillna(method='ffill')
Out[27]: a    1
         b    1
         c    2
         d    2
         e    3
         dtype: Int32
```

Аналогично можно потребовать заменить любое отсутствующее значение следующим за ним действительным значением:

```
In [28]: # заменить последующим действительным значением
         data.fillna(method='bfill')
Out[28]: a    1
         b    2
         c    2
         d    3
         e    3
         dtype: Int32
```

При работе с объектами `DataFrame` применяются аналогичные параметры, но дополнительно можно задать ось, по которой будет выполняться заполнение:

```
In [29]: df
Out[29]:      0    1  2   3
         0  1.0  NaN  2 NaN
         1  2.0  3.0  5 NaN
         2  NaN  4.0  6 NaN

In [30]: df.fillna(method='ffill', axis=1)
Out[30]:      0    1    2    3
         0  1.0  1.0  2.0  2.0
         1  2.0  3.0  5.0  5.0
         2  NaN  4.0  6.0  6.0
```

Обратите внимание, что если при заполнении по направлению «вперед» предыдущего действительного значения нет, то NA-значение остается незаполненным.

ГЛАВА 17

Иерархическая индексация

До сих пор мы рассматривали главным образом одномерные и двумерные данные, находящиеся в объектах `Series` и `DataFrame` библиотеки Pandas. Часто бывает удобно выйти за пределы двух измерений и хранить многомерные данные, то есть данные, индексированные по более чем двум ключам. Ранние версии Pandas предоставляли объекты `Panel` и `Panel4D`, позволяющие хранить трехмерные и четырехмерные данные, аналоги двухмерного `DataFrame`, но они были несколько неудобны для практического использования. Гораздо чаще использовалась *иерархическая индексация* (hierarchical indexing) для включения в один индекс нескольких *уровней*. При этом многомерные данные могут быть компактно представлены в уже привычных нам одномерных объектах `Series` и двумерных объектах `DataFrame`. (Если вас интересуют настоящие многомерные массивы с гибкими индексами в стиле Pandas, то обратите внимание на замечательный пакет Xarray (https://xarray.pydata.org/).)

В этой главе мы рассмотрим создание объектов `MultiIndex` напрямую, приведем соображения относительно индексации, получения срезов и вычисления статистических показателей по мультииндексированным данным, а также полезные методы для преобразования между простым и иерархически индексированным представлением данных.

Начнем с обычных инструкций импорта:

```
In [1]: import pandas as pd
        import numpy as np
```

Мультииндексированный объект Series

Рассмотрим, как можно представить двумерные данные в одномерном объекте `Series`. Для конкретики изучим ряд данных, в котором у каждой точки имеются символьный и числовой ключи.

Плохой способ

Пускай нам требуется проанализировать данные о штатах за два разных года. Вам может показаться соблазнительным, воспользовавшись инструментами из библиотеки Pandas, применить в качестве ключей кортежи Python:

```
In [2]: index = [('California', 2010), ('California', 2020),
                 ('New York', 2010), ('New York', 2020),
                 ('Texas', 2010), ('Texas', 2020)]
        populations = [37253956, 39538223,
                       19378102, 20201249,
                       25145561, 29145505]
        pop = pd.Series(populations, index=index)
        pop
Out[2]: (California, 2010)    37253956
        (California, 2020)    39538223
        (New York, 2010)      19378102
        (New York, 2020)      20201249
        (Texas, 2010)         25145561
        (Texas, 2020)         29145505
        dtype: int64
```

При подобной схеме индексации появляется возможность непосредственно индексировать или получать срезы на основе такого мультииндекса:

```
In [3]: pop[('California', 2020):('Texas', 2010)]
Out[3]: (California, 2020)    39538223
        (New York, 2010)      19378102
        (New York, 2020)      20201249
        (Texas, 2010)         25145561
        dtype: int64
```

Однако на этом удобство заканчивается. Например, чтобы выбрать все значения из 2010 года, придется проделать громоздкую (и потенциально медленную) очистку данных:

```
In [4]: pop[[i for i in pop.index if i[1] == 2010]]
Out[4]: (California, 2010)    37253956
        (New York, 2010)      19378102
        (Texas, 2010)         25145561
        dtype: int64
```

Хотя этот способ и приводит к желаемому результату, но он не такой изящный (и далеко не такой эффективный), как синтаксис срезов, столь полюбившийся нам в библиотеке Pandas.

Лучший способ: объект MultiIndex

Библиотека Pandas предлагает более удобный способ выполнения таких операций. Наша индексация, основанная на кортежах, по сути, является примитивным мультииндексом, и тип `MultiIndex` в библиотеке Pandas как раз обеспечивает необходимые нам операции. Создать мультииндекс из кортежей можно следующим образом:

```
In [5]: index = pd.MultiIndex.from_tuples(index)
```

Обратите внимание, что `MultiIndex` содержит несколько *уровней* (levels) индексации. В данном случае — названия штатов и годы, а также несколько меток (labels), кодирующих эти уровни для каждой точки данных.

Проиндексировав заново наши ряды данных с помощью `MultiIndex`, мы увидим иерархическое представление данных:

```
In [6]: pop = pop.reindex(index)
        pop
Out[6]: California  2010    37253956
                    2020    39538223
        New York    2010    19378102
                    2020    20201249
        Texas       2010    25145561
                    2020    29145505
        dtype: int64
```

Первые два столбца в этом представлении объекта `Series` отражают значения мультииндекса, а третий столбец — данные. Обратите внимание, что в первом столбце отсутствуют некоторые элементы: в этом мультииндексном представлении все пропущенные элементы означают то же значение, что и строкой выше.

Теперь для выбора всех данных, второй индекс которых равен 2020, можно просто воспользоваться синтаксисом срезов библиотеки Pandas:

```
In [7]: pop[:, 2020]
Out[7]: California    39538223
        New York      20201249
        Texas         29145505
        dtype: int64
```

Результат представляет собой массив с одиночным индексом и только теми ключами, которые нас интересуют. Такой синтаксис намного удобнее (а операция выполняется гораздо быстрее!), чем мультииндексное решение на основе кортежей, с которого мы начали. Сейчас мы обсудим подробнее подобные операции индексации над иерархически индексированными данными.

Мультииндекс как дополнительное измерение

Мы могли с легкостью сохранить те же данные в простом объекте `DataFrame` с индексом и метками столбцов. На самом деле библиотека Pandas создана с учетом этой равнозначности. Метод `unstack` может быстро преобразовать мультииндексный объект `Series` в индексированный обычным образом объект `DataFrame`:

```
In [8]: pop_df = pop.unstack()
        pop_df
Out[8]:                 2010      2020
        California  37253956  39538223
        New York    19378102  20201249
        Texas       25145561  29145505
```

Как и следовало ожидать, метод `stack` выполняет противоположную операцию:

```
In [9]: pop_df.stack()
Out[9]: California  2010    37253956
                    2020    39538223
        New York    2010    19378102
                    2020    20201249
        Texas       2010    25145561
                    2020    29145505
        dtype: int64
```

Почему вообще имеет смысл возиться с иерархической индексацией? Причина проста: аналогично тому, как мы использовали мультииндексацию для представления двумерных данных в одномерном объекте `Series`, можно использовать ее для представления данных с тремя или более измерениями в объектах `Series` или `DataFrame`. Каждый новый уровень в мультииндексе представляет дополнительное измерение данных. Благодаря использованию этого свойства мы получаем намного больше свободы в представлении типов данных. Например, нам может понадобиться добавить в демографические данные по каждому штату за каждый год еще один столбец (допустим, количество населения младше 18 лет). Благодаря типу `MultiIndex` это сводится к добавлению еще одного столбца в объект `DataFrame`:

```
In [10]: pop_df = pd.DataFrame({'total': pop,
                                'under18': [9284094, 8898092,
                                            4318033, 4181528,
                                            6879014, 7432474]})
         pop_df
Out[10]:                     total  under18
         California 2010  37253956  9284094
                    2020  39538223  8898092
         New York   2010  19378102  4318033
                    2020  20201249  4181528
         Texas      2010  25145561  6879014
                    2020  29145505  7432474
```

Помимо этого, все универсальные функции и остальная функциональность, обсуждавшаяся в главе 15, также прекрасно работают с иерархическими индексами. Следующий фрагмент кода вычисляет по годам долю населения младше 18 лет на основе вышеприведенных данных:

```
In [11]: f_u18 = pop_df['under18'] / pop_df['total']
         f_u18.unstack()
Out[11]:                 2010      2020
         California  0.249211  0.225050
         New York    0.222831  0.206994
         Texas       0.273568  0.255013
```

Это дает нам возможность легко и быстро манипулировать даже многомерными данными и исследовать их.

Методы создания объектов MultiIndex

Наиболее простой метод создания мультииндексированного объекта `Series` или `DataFrame` — передать в конструктор список из двух или более индексных массивов. Например:

```
In [12]: df = pd.DataFrame(np.random.rand(4, 2),
                           index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                           columns=['data1', 'data2'])
         df
Out[12]:         data1     data2
         a 1  0.748464  0.561409
           2  0.379199  0.622461
         b 1  0.701679  0.687932
           2  0.436200  0.950664
```

Вся работа по созданию экземпляра `MultiIndex` выполняется в фоновом режиме.

Если передать словарь с соответствующими кортежами в качестве ключей, то библиотека Pandas автоматически распознает такой синтаксис и по умолчанию будет использовать объект `MultiIndex`:

```
In [13]: data = {('California', 2010): 37253956,
                 ('California', 2020): 39538223,
                 ('New York', 2010): 19378102,
                 ('New York', 2020): 20201249,
                 ('Texas', 2010): 25145561,
                 ('Texas', 2020): 29145505}
         pd.Series(data)
Out[13]: California  2010    37253956
                     2020    39538223
```

```
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
dtype: int64
```

Тем не менее иногда удобно создавать объекты `MultiIndex` явно. Далее мы рассмотрим несколько методов, служащих этой цели.

Явные конструкторы MultiIndex

При формировании индекса для большей гибкости можно воспользоваться имеющимися в классе `pd.MultiIndex` конструкторами. Например, объект `MultiIndex` можно сформировать из простого списка массивов, задающих значения индекса на каждом из уровней:

```
In [14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
Out[14]: MultiIndex([('a', 1),
                     ('a', 2),
                     ('b', 1),
                     ('b', 2)],
                    )
```

Или из списка кортежей, задающих значения индекса в каждой точке:

```
In [15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
Out[15]: MultiIndex([('a', 1),
                     ('a', 2),
                     ('b', 1),
                     ('b', 2)],
                    )
```

И даже из декартова произведения обычных индексов:

```
In [16]: pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
Out[16]: MultiIndex([('a', 1),
                     ('a', 2),
                     ('b', 1),
                     ('b', 2)],
                    )
```

Объект `MultiIndex` можно создать непосредственно с помощью его внутреннего представления, передав в конструктор аргумент `levels` (список списков, содержащих имеющиеся значения индекса для каждого уровня) и аргумент `codes` (список списков меток):

```
In [17]: pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
                       codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

```
Out[17]: MultiIndex([('a', 1),
                     ('a', 2),
                     ('b', 1),
                     ('b', 2)],
                    )
```

Любой из этих объектов можно передать в аргументе `index` конструкторам объектов `Series` или `DataFrame` или методу `reindex` уже существующих объектов `Series` или `DataFrame`.

Названия уровней мультииндексов

Иногда бывает удобно задать названия для уровней объекта `MultiIndex`. Для этого можно передать аргумент `names` любому из вышеперечисленных конструкторов класса `MultiIndex` или задать значения атрибута `names` постфактум:

```
In [18]: pop.index.names = ['state', 'year']
         pop
Out[18]: state       year
         California  2010    37253956
                     2020    39538223
         New York    2010    19378102
                     2020    20201249
         Texas       2010    25145561
                     2020    29145505
         dtype: int64
```

При обработке более сложных наборов данных такой способ дает возможность не терять из виду, что означают различные значения индекса.

Мультииндекс для столбцов

В объектах `DataFrame` строки и столбцы полностью симметричны, и у столбцов, и у строк могут быть многоуровневые индексы. Рассмотрим следующий пример, имитирующий некоторые (в чем-то достаточно реалистичные) медицинские данные:

```
In [19]: # Иерархические индексы и столбцы
         index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                            names=['year', 'visit'])
         columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'],
                                               ['HR', 'Temp']],
                                               names=['subject', 'type'])

         # Создаем имитационные данные
         data = np.round(np.random.randn(4, 6), 1)
         data[:, ::2] *= 10
         data += 37
```

```
        # Создаем объект DataFrame
        health_data = pd.DataFrame(data, index=index, columns=columns)
        health_data
Out[19]: subject      Bob       Guido         Sue
        type          HR  Temp    HR  Temp    HR  Temp
        year visit
        2013 1      30.0  38.0  56.0  38.3  45.0  35.8
             2      47.0  37.1  27.0  36.0  37.0  36.4
        2014 1      51.0  35.9  24.0  36.7  32.0  36.2
             2      49.0  36.3  48.0  39.2  31.0  35.7
```

По сути это четырехмерные данные со следующими измерениями: субъект, измеряемый параметр[1], год и номер посещения. При наличии этого мы можем, например, индексировать столбец верхнего уровня по имени человека и получить объект `DataFrame`, содержащий информацию только об этом человеке:

```
In [20]: health_data['Guido']
Out[20]: type          HR  Temp
        year visit
        2013 1      56.0  38.3
             2      27.0  36.0
        2014 1      24.0  36.7
             2      48.0  39.2
```

Индексация и срезы по мультииндексу

Объект `MultiIndex` спроектирован так, чтобы индексация и срезы по мультииндексу были интуитивно понятны, особенно если думать об индексах как о дополнительных измерениях. Изучим сначала индексацию мультииндексированного объекта `Series`, а затем мультииндексированного объекта `DataFrame`.

Мультииндексация объектов Series

Рассмотрим мультииндексированный объект `Series`, содержащий количество населения по штатам:

```
In [21]: pop
Out[21]: state       year
        California  2010    37253956
                    2020    39538223
        New York    2010    19378102
                    2020    20201249
        Texas       2010    25145561
                    2020    29145505
        dtype: int64
```

[1] Пульс (HR, от *англ.* heart rate — «частота сердцебиений») и температура (temp).

Обращаться к отдельным элементам можно путем индексации по нескольким индексам:

```
In [22]: pop['California', 2010]
Out[22]: 37253956
```

Объект `MultiIndex` поддерживает также *частичную индексацию* (partial indexing), то есть индексацию только по одному из уровней. Результат — тоже объект `Series` с более низкоуровневыми индексами:

```
In [23]: pop['California']
Out[23]: year
         2010    37253956
         2020    39538223
         dtype: int64
```

Возможно также получение частичных срезов, если мультииндекс отсортирован (см. обсуждение в разделе «Отсортированные и неотсортированные индексы» ниже):

```
In [24]: poploc['california':'new york']
Out[24]: state       year
         california  2010    37253956
                     2020    39538223
         new york    2010    19378102
                     2020    20201249
         dtype: int64
```

С помощью отсортированных индексов можно выполнять частичную индексацию по нижним уровням, указав пустой срез в первом индексе:

```
In [25]: pop[:, 2010]
Out[25]: state
         california    37253956
         new york      19378102
         texas         25145561
         dtype: int64
```

Другие типы индексации и выборки (обсуждаемые в главе 14) тоже работают. Выборка данных на основе булевой маски:

```
In [26]: pop[pop > 22000000]
Out[26]: state       year
         California  2010    37253956
                     2020    39538223
         Texas       2010    25145561
                     2020    29145505
         dtype: int64
```

Выборка на основе «прихотливой» индексации:

```
In [27]: pop[['California', 'Texas']]
Out[27]: state       year
         California  2010    37253956
                     2020    39538223
         Texas       2010    25145561
                     2020    29145505
         dtype: int64
```

Мультииндексация объектов DataFrame

Мультииндексированные объекты `DataFrame` ведут себя аналогично. Рассмотрим наш модельный медицинский объект `DataFrame`:

```
In [28]: health_data
Out[28]: subject      Bob        Guido         Sue
         type          HR  Temp    HR  Temp    HR  Temp
         year visit
         2013 1      30.0  38.0  56.0  38.3  45.0  35.8
              2      47.0  37.1  27.0  36.0  37.0  36.4
         2014 1      51.0  35.9  24.0  36.7  32.0  36.2
              2      49.0  36.3  48.0  39.2  31.0  35.7
```

Не забывайте, что в объектах `DataFrame` основными являются столбцы, и используемый для мультииндексированных `Series` синтаксис применяется тоже к столбцам. Например, можно узнать пульс Гвидо (Guido) с помощью простой операции:

```
In [29]: health_data['Guido', 'HR']
Out[29]: year  visit
         2013  1        56.0
               2        27.0
         2014  1        24.0
               2        48.0
         Name: (Guido, HR), dtype: float64
```

Как и в случае с одиночным индексом, можно использовать индексаторы `loc` и `iloc`, описанные в главе 14. Например:

```
In [30]: health_data.iloc[:2, :2]
Out[30]: subject      Bob
         type          HR  Temp
         year visit
         2013 1      30.0  38.0
              2      47.0  37.1
```

Эти индексаторы возвращают массивоподобное представление исходных двумерных данных, но в каждом отдельном индексе в `loc` и `iloc` можно указать кортеж из нескольких индексов. Например:

```
In [31]: health_data.loc[:, ('Bob', 'HR')]
Out[31]: year  visit
         2013  1        30.0
               2        47.0
         2014  1        51.0
               2        49.0
         Name: (Bob, HR), dtype: float64
```

Работать со срезами в подобных кортежах индексов не очень удобно: попытка создать срез кортежа может привести к синтаксической ошибке:

```
In [32]: health_data.loc[(:, 1), (:, 'HR')]
SyntaxError: invalid syntax (3311942670.py, line 1)
```

Избежать этого можно, сформировав срез явно с помощью встроенной функции Python `slice`, но в данном случае лучше использовать объект `IndexSlice`, предназначенный библиотекой Pandas как раз для подобной ситуации. Например:

```
In [33]: idx = pd.IndexSlice
         health_data.loc[idx[:, 1], idx[:, 'HR']]
Out[33]: subject      Bob Guido   Sue
         type          HR    HR    HR
         year visit
         2013 1      30.0  56.0  45.0
         2014 1      51.0  24.0  32.0
```

Существует множество способов взаимодействия с данными в мультииндексированных объектах `Series` и `DataFrame`, и лучший способ освоить их — начать с ними экспериментировать!

Перегруппировка мультииндексов

Один из ключей к эффективной работе с мультииндексированными данными — умение эффективно преобразовывать данные. Существует немало операций, сохраняющих всю информацию из набора данных, но преобразующих ее ради удобства проведения различных вычислений. Мы рассмотрели небольшой пример с методами `stack` и `unstack`, но есть и другие способы точного управления перегруппировкой данных между иерархическими индексами и столбцами.

Отсортированные и неотсортированные индексы

Выше я уже отмечал, но должен повторить еще раз. *Большинство операций получения срезов с мультииндексами завершается ошибкой, если индекс не отсортирован.* Рассмотрим этот вопрос.

Начнем с создания простых мультииндексированных данных, индексы в которых *не отсортированы лексикографически*:

```
In [34]: index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
         data = pd.Series(np.random.rand(6), index=index)
         data.index.names = ['char', 'int']
         data
Out[34]: char  int
         a     1      0.280341
               2      0.097290
         c     1      0.206217
               2      0.431771
         b     1      0.100183
               2      0.015851
         dtype: float64
```

При попытке получить частичный срез этого индекса код сгенерирует ошибку:

```
In [35]: try:
             data['a':'b']
         except KeyError as e:
             print("KeyError", e)
KeyError 'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Хотя из сообщения об ошибке[1] это не вполне понятно, ошибка генерируется, потому что объект `MultiIndex` не отсортирован. По различным причинам частичные срезы и другие подобные операции требуют, чтобы уровни мультииндекса были отсортированы (лексикографически упорядочены). Библиотека Pandas предоставляет множество удобных инструментов для выполнения подобной сортировки, таких как методы `sort_index` и `sortlevel` объекта `DataFrame`. Мы воспользуемся простейшим из них — методом `sort_index`:

```
In [36]: data = data.sort_index()
         data
Out[36]: char  int
         a     1      0.280341
               2      0.097290
         b     1      0.100183
               2      0.015851
         c     1      0.206217
               2      0.431771
         dtype: float64
```

После подобной сортировки индекса частичный срез будет выполняться как положено:

```
In [37]: data['a':'b']
Out[37]: char  int
```

[1] «Длина ключа больше глубины лексикографической сортировки объекта MultiIndex».

```
        a     1      0.280341
              2      0.097290
        b     1      0.100183
              2      0.015851
        dtype: float64
```

Выполнение операций stack и unstack над индексами

Существует возможность преобразовывать набор данных из вертикального мультииндексированного в простое двумерное представление, при необходимости указывая требуемый уровень:

```
In [38]: pop.unstack(level=0)
Out[38]: year            2010      2020
         state
         California  37253956  39538223
         New York    19378102  20201249
         Texas       25145561  29145505

In [39]: pop.unstack(level=1)
Out[39]: state       year
         California  2010    37253956
                     2020    39538223
         New York    2010    19378102
                     2020    20201249
         Texas       2010    25145561
                     2020    29145505
         dtype: int64
```

Метод `stack` действует противоположным образом, и его можно использовать, чтобы получить обратно исходный ряд данных:

```
In [40]: pop.unstack().stack()
Out[40]: state       year
         California  2010    37253956
                     2020    39538223
         New York    2010    19378102
                     2020    20201249
         Texas       2010    25145561
                     2020    29145505
         dtype: int64
```

Создание и перестройка индексов

Еще один способ перегруппировки иерархических данных — преобразовать метки индекса в столбцы с помощью метода `reset_index`. Результатом применения этого метода к нашему демографическому словарю будет объект `DataFrame` со столбцами `state` (штат) и `year` (год), содержащими информацию, ранее

находившуюся в индексе. Для большей ясности можно задать название для представленных в виде столбцов данных:

```
In [41]: pop_flat = pop.reset_index(name='population')
         pop_flat
Out[41]:         state  year  population
         0  California  2010    37253956
         1  California  2020    39538223
         2    New York  2010    19378102
         3    New York  2020    20201249
         4       Texas  2010    25145561
         5       Texas  2020    29145505
```

На практике часто удобнее создать объект `MultiIndex` из значений столбцов. Это можно сделать с помощью метода `set_index` объекта `DataFrame`, возвращающего мультииндексированный объект `DataFrame`:

```
In [42]: pop_flat.set_index(['state', 'year'])
Out[42]:                  population
         state      year
         California 2010    37253956
                    2020    39538223
         New York   2010    19378102
                    2020    20201249
         Texas      2010    25145561
                    2020    29145505
```

Я считаю такой способ перестройки индекса одним из самых удобных при работе с реальными наборами данных.

ГЛАВА 18

Объединение наборов данных: конкатенация и добавление в конец

Некоторые наиболее интересные исследования выполняются благодаря объединению различных источников данных. Эти операции могут включать в себя что угодно, начиная с простейшей конкатенации двух различных наборов данных и заканчивая более сложными соединениями и слияниями в стиле баз данных, корректно обрабатывающих все возможные частичные совпадения наборов. Объекты `Series` и `DataFrame` созданы в расчете на подобные операции, и библиотека Pandas содержит функции и методы для быстрого и удобного выполнения таких манипуляций.

Мы рассмотрим простую конкатенацию объектов `Series` и `DataFrame` с помощью функции `pd.concat`, углубимся в реализованные в библиотеке Pandas более запутанные слияния и соединения, выполняемые в оперативной памяти.

Начнем с обычных инструкций импорта:

```
In [1]: import pandas as pd
        import numpy as np
```

Для удобства опишем следующую функцию, создающую объект `DataFrame` определенной формы, которая нам пригодится в дальнейшем:

```
In [2]: def make_df(cols, ind):
            """Быстро создает объект DataFrame"""
            data = {c: [str(c) + str(i) for i in ind]
                    for c in cols}
            return pd.DataFrame(data, ind)

        # Экземпляр DataFrame
        make_df('ABC', range(3))
```

```
Out[2]:     A   B   C
        0  A0  B0  C0
        1  A1  B1  C1
        2  A2  B2  C2
```

Дополнительно создадим маленький класс, который позволит нам отображать несколько объектов `DataFrame` бок о бок. В классе используется специальный метод `_repr_html_`, который в IPython/Jupyter предназначен для реализации расширенного отображения объектов:

```
In [3]: class display(object):
            """Отображает несколько объектов в формате HTML"""
            template = """<div style="float: left; padding: 10px;">
            <p style='font-family:"Courier New", Courier, monospace'>{0}{1}"""
            def __init__(self, *args):
                self.args = args

            def _repr_html_(self):
                return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                                 for a in self.args)

            def __repr__(self):
                return '\n\n'.join(a + '\n' + repr(eval(a))
                                   for a in self.args)
```

Порядок использования этого класса станет яснее, когда мы продолжим обсуждение в следующем разделе.

Напоминание: конкатенация массивов NumPy

Конкатенация объектов `Series` и `DataFrame` очень похожа на конкатенацию массивов библиотеки NumPy, которую можно осуществить посредством функции `np.concatenate`, обсуждавшейся в главе 5. Напомним, что таким образом можно объединять содержимое двух или более массивов в один:

```
In [4]: x = [1, 2, 3]
        y = [4, 5, 6]
        z = [7, 8, 9]
        np.concatenate([x, y, z])
Out[4]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Первый аргумент этой функции — список или кортеж объединяемых массивов. Кроме того, она принимает именованный аргумент `axis`, позволяющий задавать ось, по которой будет выполняться конкатенация:

```
In [5]: x = [[1, 2],
             [3, 4]]
        np.concatenate([x, x], axis=1)
Out[5]: array([[1, 2, 1, 2],
               [3, 4, 3, 4]])
```

Простая конкатенация с помощью метода pd.concat

Функция `pd.concat` предлагает синтаксис, аналогичный синтаксису функции `np.concatenate`, но принимает множество разных параметров, которые мы обсудим далее:

```
# Сигнатура в Pandas v1.3.5
pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None,
          levels=None, names=None, verify_integrity=False,
          sort=False, copy=True)
```

Функцию `pd.concat` можно использовать для простой конкатенации объектов `Series` или `DataFrame` подобно тому, как функцию `np.concatenate` можно применять для простой конкатенации массивов:

```
In [6]: ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
        ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
        pd.concat([ser1, ser2])
Out[6]: 1    A
        2    B
        3    C
        4    D
        5    E
        6    F
        dtype: object
```

Она также подходит для конкатенации многомерных объектов, таких как `DataFrame`:

```
In [7]: df1 = make_df('AB', [1, 2])
        df2 = make_df('AB', [3, 4])
        display('df1', 'df2', 'pd.concat([df1, df2])')
Out[7]: df1           df2           pd.concat([df1, df2])
            A   B         A   B          A   B
        1  A1  B1     3  A3  B3     1  A1  B1
        2  A2  B2     4  A4  B4     2  A2  B2
                                    3  A3  B3
                                    4  A4  B4
```

По умолчанию конкатенация объектов `DataFrame` производится построчно (то есть `axis=0`). Подобно `np.concatenate`, функция `pd.concat` позволяет указать ось, по которой будет выполняться конкатенация. Рассмотрим следующий пример:

```
In [8]: df3 = make_df('AB', [0, 1])
        df4 = make_df('CD', [0, 1])
        display('df3', 'df4', "pd.concat([df3, df4], axis='columns')")
Out[8]: df3           df4           pd.concat([df3, df4], axis='columns')
            A   B         C   D          A   B   C   D
        0  A0  B0     0  C0  D0     0  A0  B0  C0  D0
        1  A1  B1     1  C1  D1     1  A1  B1  C1  D1
```

Мы могли задать ось явно, `axis=1`, но в этом примере использован более интуитивно понятный вариант `axis='columns'`.

Дублирование индексов

Важное различие между функциями `np.concatenate` и `pd.concat` состоит в том, что конкатенация из библиотеки Pandas *сохраняет индексы*, даже если в результате некоторые индексы будут повторяться. Рассмотрим следующий пример:

```
In [9]: x = make_df('AB', [0, 1])
        y = make_df('AB', [2, 3])
        y.index = x.index  # дублируем индексы
        display('x', 'y', 'pd.concat([x, y])')
Out[9]: x              y              pd.concat([x, y])
            A   B          A   B          A   B
        0  A0  B0      0  A2  B2      0  A0  B0
        1  A1  B1      1  A3  B3      1  A1  B1
                                      0  A2  B2
                                      1  A3  B3
```

Обратите внимание на повторяющиеся индексы в результате. Хотя в объектах `DataFrame` это допустимо, подобный результат часто может быть нежелателен. Функция `pd.concat` предлагает несколько способов решения этой проблемы.

Интерпретация повторяющихся индексов как ошибки

Если нужно гарантировать уникальность индексов в результате, возвращаемом функцией `pd.concat`, то ей можно передать флаг `verify_integrity` со значением `True`. В этом случае, обнаружив повторяющиеся индексы, операция конкатенации завершится ошибкой. Вот пример, который перехватывает ошибку и выводит сообщение в консоль:

```
In [10]: try:
             pd.concat([x, y], verify_integrity=True)
         except ValueError as e:
             print("ValueError:", e)
ValueError: Indexes have overlapping values:
            Int64Index([0, 1], dtype='int64')
```

Игнорирование индекса

Иногда индекс не имеет особого значения и его можно просто проигнорировать. Для этого достаточно передать в вызов `pd.concat` флаг `ignore_index` со значением `True`. В этом случае операция конкатенации создаст новый целочисленный индекс для итогового объекта `DataFrame`:

```
In [11]: display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
Out[11]: x              y              pd.concat([x, y], ignore_index=True)
             A   B          A   B          A   B
         0  A0  B0      0  A2  B2      0  A0  B0
         1  A1  B1      1  A3  B3      1  A1  B1
                                       2  A2  B2
                                       3  A3  B3
```

Добавление ключей мультииндекса

Еще один вариант — передать параметр `keys`, чтобы задать разные метки для разных источников данных. В результате получатся иерархически индексированные ряды, содержащие данные:

```
In [12]: display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
Out[12]: x              y              pd.concat([x, y], keys=['x', 'y'])
             A   B          A   B           A   B
         0  A0  B0      0  A2  B2      x 0  A0  B0
         1  A1  B1      1  A3  B3        1  A1  B1
                                       y 0  A2  B2
                                         1  A3  B3
```

Полученный в результате мультииндексированный объект `DataFrame` можно преобразовать в требуемое представление с помощью инструментов, описанных в главе 17.

Конкатенация с использованием соединений

В примерах, представленных выше, в основном объединяются объекты `DataFrame` с общими названиями столбцов. Но на практике данные из разных источников могут иметь разные наборы имен столбцов. На этот случай функция `pd.concat` предлагает несколько дополнительных параметров. Рассмотрим пример конкатенации двух объектов `DataFrame`, в которых некоторые столбцы имеют одинаковые имена:

```
In [13]: df5 = make_df('ABC', [1, 2])
         df6 = make_df('BCD', [3, 4])
         display('df5', 'df6', 'pd.concat([df5, df6])')
Out[13]: df5                 df6                 pd.concat([df5, df6])
             A   B   C           B   C   D          A   B   C    D
         1  A1  B1  C1       3  B3  C3  D3       1   A1  B1  C1  NaN
         2  A2  B2  C2       4  B4  C4  D4       2   A2  B2  C2  NaN
                                                 3  NaN  B3  C3   D3
                                                 4  NaN  B4  C4   D4
```

По умолчанию элементы с отсутствующими данными заполняются NA-значениями. Чтобы поменять это поведение, можно передать в вызов `concat` параметр `join`. По умолчанию соединение выполняется как объединение входных столбцов (`join='outer'`), но есть возможность выполнить соединение как пересечение, передав параметр `join='inner'`:

```
In [14]: display('df5', 'df6',
                 "pd.concat([df5, df6], join='inner')")
Out[14]: df5                 df6
             A   B   C           B   C   D
         1  A1  B1  C1       3  B3  C3  D3
         2  A2  B2  C2       4  B4  C4  D4
```

```
        pd.concat([df5, df6], join='inner')
            B   C
        1  B1  C1
        2  B2  C2
        3  B3  C3
        4  B4  C4
```

Другой интересный способ — вызвать метод `reindex` перед конкатенацией, чтобы точно указать, какие столбцы должны отбрасываться:

```
In [15]: pd.concat([df5, df6.reindex(df5.columns, axis=1)])
Out[15]:      A   B   C
        1   A1  B1  C1
        2   A2  B2  C2
        3  NaN  B3  C3
        4  NaN  B4  C4
```

Метод append()

Непосредственная конкатенация массивов настолько распространена, что в объекты `Series` и `DataFrame` был включен метод `append`, позволяющий выполнить то же самое с меньшими усилиями. Например, вместо вызова `pd.concat([df1, df2])` можно вызвать `df1.append(df2)`:

```
In [16]: display('df1', 'df2', 'df1.append(df2)')
Out[16]: df1            df2           df1.append(df2)
            A   B          A   B          A   B
        1  A1  B1      3  A3  B3      1  A1  B1
        2  A2  B2      4  A4  B4      2  A2  B2
                                      3  A3  B3
                                      4  A4  B4
```

Не забывайте, что, в отличие от методов `append` и `extend` обычных списков, метод `append` в библиотеке Pandas не изменяет исходный объект, а создает новый, включающий объединенные данные, что делает этот метод не слишком эффективным, поскольку означает создание нового индекса и буфера данных. Поэтому если потребуется выполнить несколько операций `append`, то лучше создать список объектов `DataFrame` и передать их все сразу функции `concat`.

В следующей главе мы рассмотрим другой подход к объединению данных из нескольких источников: слияние/соединение в стиле баз данных с помощью функции `pd.merge`. Дополнительную информацию о методах `concat`, `append` и их возможностях см. в разделе «Merge, Join and Concatenate» в документации библиотеки Pandas (https://oreil.ly/cY16c).

ГЛАВА 19

Объединение наборов данных: слияние и соединение

Одна из важных особенностей библиотеки Pandas — ее высокопроизводительные операции соединения и слияния, выполняемые в оперативной памяти, которые могут показаться знакомыми тем, кто когда-либо работал с базами данных. Основной интерфейс к этим операциям — функция `pd.merge`. Далее мы рассмотрим несколько примеров ее применения.

Для удобства мы снова определим функцию `display` из предыдущей главы, попутно выполнив обычные инструкции импорта:

```
In [1]: import pandas as pd
        import numpy as np

        class display(object):
            """Отображает несколько объектов в формате HTML"""
            template = """<div style="float: left; padding: 10px;">
            <p style='font-family:"Courier New", Courier, monospace'>{0}{1}
            """
            def __init__(self, *args):
                self.args = args

            def _repr_html_(self):
                return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                                 for a in self.args)

            def __repr__(self):
                return '\n\n'.join(a + '\n' + repr(eval(a))
                                   for a in self.args)
```

Реляционная алгебра

Реализованное в методе `pd.merge` поведение представляет собой подмножество того, что известно под названием «реляционная алгебра» (relational algebra). Реляционная алгебра — формальный набор правил манипуляции реляционными данными, формирующий теоретическую основу для операций, поддерживаемых в большинстве баз данных. Сила реляционного подхода состоит в предоставлении нескольких простейших операций — своеобразных «кирпичиков» для построения более сложных операций над любым набором данных. При наличии эффективно реализованного в базе данных или другой программе подобного базового набора операций можно выполнять широкий диапазон весьма сложных составных операций.

Библиотека Pandas реализует несколько из этих базовых «кирпичиков» в функции `pd.merge` и родственном ей методе `join` объектов `Series` и `DataFrame`. Они позволяют эффективно связывать данные из различных источников.

Виды соединений

Функция `pd.merge` реализует множество типов соединений: «один-к-одному», «многие-к-одному» и «многие-ко-многим». Все эти три типа соединений доступны через один и тот же интерфейс `pd.merge`, тип выполняемого соединения зависит от формы входных данных. Ниже мы рассмотрим примеры этих трех типов слияний и обсудим более подробно имеющиеся параметры.

Соединения «один-к-одному»

Простейший тип слияния — соединение «один-к-одному», во многом напоминающее конкатенацию по столбцам, которую мы изучили в главе 18. В качестве примера рассмотрим следующие два объекта `DataFrame`, содержащие информацию о нескольких служащих компании:

```
In [2]: df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'group': ['Accounting', 'Engineering',
                                      'Engineering', 'HR']})
        df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                            'hire_date': [2004, 2008, 2012, 2014]})
        display('df1', 'df2')
Out[2]: df1                            df2
          employee        group          employee  hire_date
        0      Bob   Accounting        0     Lisa       2004
        1     Jake  Engineering        1      Bob       2008
        2     Lisa  Engineering        2     Jake       2012
        3      Sue           HR        3      Sue       2014
```

Чтобы объединить эту информацию в один объект `DataFrame`, воспользуемся функцией `pd.merge`:

```
In [3]: df3 = pd.merge(df1, df2)
        df3
Out[3]:   employee        group  hire_date
        0      Bob   Accounting       2008
        1     Jake  Engineering       2012
        2     Lisa  Engineering       2004
        3      Sue           HR       2014
```

Функция `pd.merge` распознает присутствие столбца `employee` в обоих объектах `DataFrame` и автоматически выполняет соединение, используя этот столбец в качестве ключа. Результатом слияния становится новый объект `DataFrame`, объединяющий информацию из двух входных объектов. Обратите внимание, что порядок следования записей в столбцах может не сохраняться: в данном случае записи в `df1` и `df2` по-разному отсортированы по столбцу `employee`, но функция `pd.merge` корректно обрабатывает эту ситуацию. Кроме того, не забывайте, что слияние игнорирует индекс, за исключением особого случая слияния по индексу (см. обсуждение именованных аргументов `left_index` и `right_index` ниже).

Соединения «многие-к-одному»

Соединения «многие-к-одному» применяются, когда один из двух ключевых столбцов содержит повторяющиеся значения. Соединения этого вида сохраняют повторяющиеся записи в итоговом объекте `DataFrame`. Рассмотрим следующий пример соединения «многие-к-одному»:

```
In [4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                            'supervisor': ['Carly', 'Guido', 'Steve']})
        display('df3', 'df4', 'pd.merge(df3, df4)')
Out[4]: df3                                  df4
          employee        group  hire_date             group supervisor
        0      Bob   Accounting       2008       0   Accounting      Carly
        1     Jake  Engineering       2012       1  Engineering      Guido
        2     Lisa  Engineering       2004       2           HR      Steve
        3      Sue           HR       2014

        pd.merge(df3, df4)
          employee        group  hire_date supervisor
        0      Bob   Accounting       2008      Carly
        1     Jake  Engineering       2012      Guido
        2     Lisa  Engineering       2004      Guido
        3      Sue           HR       2014      Steve
```

В итоговом объекте `DataFrame` имеется дополнительный столбец с информацией о руководителе (`supervisor`), информация в котором повторяется в соответствии с входными данными.

Соединения «многие-ко-многим»

Соединения «многие-ко-многим» семантически несколько сложнее, но тем не менее четко определены. Если столбец ключа в обоих массивах содержит повторяющиеся значения, то результат окажется слиянием типа «многие-ко-многим». Рассмотрим следующий пример, в котором объект DataFrame отражает один или несколько навыков, соответствующих конкретной группе.

Выполнив соединение «многие-ко-многим», можно выяснить навыки каждого конкретного человека:

```
In [5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                      'Engineering', 'Engineering', 'HR', 'HR'],
                            'skills': ['math', 'spreadsheets', 'software', 'math',
                                       'spreadsheets', 'organization']})
        display('df1', 'df5', "pd.merge(df1, df5)")
Out[5]: df1                         df5
        employee        group               group        skills
      0      Bob   Accounting     0   Accounting          math
      1     Jake  Engineering     1   Accounting  spreadsheets
      2     Lisa  Engineering     2  Engineering      software
      3      Sue           HR     3  Engineering          math
                                  4           HR  spreadsheets
                                  5           HR  organization

      pd.merge(df1, df5)
        employee        group        skills
      0      Bob   Accounting          math
      1      Bob   Accounting  spreadsheets
      2     Jake  Engineering      software
      3     Jake  Engineering          math
      4     Lisa  Engineering      software
      5     Lisa  Engineering          math
      6      Sue           HR  spreadsheets
      7      Sue           HR  organization
```

Эти три типа соединений можно использовать совместно с другими инструментами библиотеки Pandas для реализации разных функциональных возможностей. Но на практике наборы данных редко оказываются такими же «чистыми», как те, с которыми мы имели дело. В следующем разделе мы рассмотрим параметры метода pd.merge, позволяющие более точно описать желаемое поведение операций соединения.

Задание ключа слияния

Мы рассмотрели поведение метода pd.merge по умолчанию: он отыскивает в двух входных объектах столбцы с совпадающими именами и использует их в качестве ключа. Однако зачастую имена столбцов не совпадают точно, и в методе pd.merge имеется немало параметров для разрешения такой ситуации.

Именованный аргумент on

Самый простой путь — явно передать методу именованный аргумент `on` и указать в нем имя или список имен столбцов:

```
In [6]: display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
Out[6]: df1                          df2
          employee        group        employee  hire_date
        0      Bob   Accounting     0     Lisa       2004
        1     Jake  Engineering     1      Bob       2008
        2     Lisa  Engineering     2     Jake       2012
        3      Sue           HR     3      Sue       2014

        pd.merge(df1, df2, on='employee')
          employee        group  hire_date
        0      Bob   Accounting       2008
        1     Jake  Engineering       2012
        2     Lisa  Engineering       2004
        3      Sue           HR       2014
```

Этот параметр применим, только когда в левом и правом объектах `DataFrame` имеется столбец с указанным именем.

Именованные аргументы left_on и right_on

Иногда приходится выполнять слияние двух наборов данных со столбцами с различными именами. Например, у нас может быть набор данных, в котором столбец с именами служащих называется `Name`, а не `Employee`. В этом случае можно воспользоваться именованными аргументами `left_on` и `right_on`, чтобы указать имена двух нужных столбцов:

```
In [7]: df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'salary': [70000, 80000, 120000, 90000]})
        display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee",
                 right_on="name")')
Out[7]: df1                          df3
          employee        group         name  salary
        0      Bob   Accounting     0   Bob   70000
        1     Jake  Engineering     1  Jake   80000
        2     Lisa  Engineering     2  Lisa  120000
        3      Sue           HR     3   Sue   90000

        pd.merge(df1, df3, left_on="employee", right_on="name")
          employee        group  name  salary
        0      Bob   Accounting   Bob   70000
        1     Jake  Engineering  Jake   80000
        2     Lisa  Engineering  Lisa  120000
        3      Sue           HR   Sue   90000
```

Результат этой операции содержит избыточный столбец, который можно при желании удалить. Например, с помощью метода `drop`, имеющегося в объектах `DataFrame`:

```
In [8]: pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
Out[8]:   employee        group  salary
        0      Bob   Accounting   70000
        1     Jake  Engineering   80000
        2     Lisa  Engineering  120000
        3      Sue           HR   90000
```

Именованные аргументы left_index и right_index

Иногда удобнее выполнять соединение по индексу, а не по столбцу. Допустим, у нас имеются следующие данные:

```
In [9]: df1a = df1.set_index('employee')
        df2a = df2.set_index('employee')
        display('df1a', 'df2a')
Out[9]: df1a                      df2a
                         group              hire_date
        employee                  employee
        Bob         Accounting    Lisa           2004
        Jake       Engineering    Bob            2008
        Lisa       Engineering    Jake           2012
        Sue                 HR    Sue            2014
```

В таких случаях в роли ключа соединения можно использовать индексы, передав их методу `pd.merge` в именованных аргументах `left_index` и/или `right_index`:

```
In [10]: display('df1a', 'df2a',
                 "pd.merge(df1a, df2a, left_index=True, right_index=True)")
Out[10]: df1a                      df2a
                          group              hire_date
         employee                  employee
         Bob         Accounting    Lisa           2004
         Jake       Engineering    Bob            2008
         Lisa       Engineering    Jake           2012
         Sue                 HR    Sue            2014

         pd.merge(df1a, df2a, left_index=True, right_index=True)
                          group  hire_date
         employee
         Bob         Accounting       2008
         Jake       Engineering       2012
         Lisa       Engineering       2004
         Sue                 HR       2014
```

Для удобства в объектах `DataFrame` реализован метод `join`, по умолчанию выполняющий соединение по индексам:

```
In [11]: df1a.join(df2a)
Out[11]:                  group  hire_date
         employee
         Bob        Accounting       2008
         Jake      Engineering       2012
         Lisa      Engineering       2004
         Sue                HR       2014
```

Если требуется комбинация соединения по столбцам и индексам, то для достижения нужного поведения можно воспользоваться сочетанием именованных аргументов `left_index` и `right_on` или `left_on` и `right_index`:

```
In [12]: display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True,
                  right_on='name')")
Out[12]: df1a                       df3
                         group      name  salary
         employee                   0   Bob   70000
         Bob        Accounting      1  Jake   80000
         Jake      Engineering      2  Lisa  120000
         Lisa      Engineering      3   Sue   90000
         Sue                HR

         pd.merge(df1a, df3, left_index=True, right_on='name')
                  group  name  salary
         0   Accounting   Bob   70000
         1  Engineering  Jake   80000
         2  Engineering  Lisa  120000
         3           HR   Sue   90000
```

Все эти способы можно также использовать и в случае нескольких индексов и/или столбцов; синтаксис реализации такого поведения интуитивно понятен. Более подробную информацию по этому вопросу вы найдете в разделе «Merge, Join and Concatenate» («Слияние, соединение и конкатенация») в документации Pandas (https://oreil.ly/ffyAp).

Применение операций над множествами для соединений

Во всех предыдущих примерах мы игнорировали один важный нюанс выполнения соединения — вид используемой при соединении операции алгебры множеств. Это выбор играет важную роль, когда какое-либо значение присутствует в одном ключевом столбце, но отсутствует в другом. Рассмотрим следующий пример:

```
In [13]: df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                             'food': ['fish', 'beans', 'bread']},
                            columns=['name', 'food'])
```

```
        df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                            'drink': ['wine', 'beer']},
                           columns=['name', 'drink'])
        display('df6', 'df7', 'pd.merge(df6, df7)')
Out[13]: df6                    df7
             name   food            name drink
         0  Peter   fish       0    Mary  wine
         1   Paul  beans       1  Joseph  beer
         2   Mary  bread

         pd.merge(df6, df7)
            name   food drink
         0  Mary  bread  wine
```

Здесь выполняется соединение двух наборов данных, у которых совпадает только одна запись `name: Mary`. По умолчанию результат будет содержать *пересечение* двух входных множеств — *внутреннее соединение* (inner join). Тип соединения можно указать явно с помощью именованного аргумента `how`, имеющего по умолчанию значение `'inner'`:

```
In [14]: pd.merge(df6, df7, how='inner')
Out[14]:    name   food drink
         0  Mary  bread  wine
```

Другие возможные значения аргумента `how`: `'outer'`, `'left'` и `'right'`. *Внешнее соединение* (outer join) возвращает объединение входных столбцов и заполняет значениями NA все отсутствующие значения:

```
In [15]: display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
Out[15]: df6                    df7
             name   food            name drink
         0  Peter   fish       0    Mary  wine
         1   Paul  beans       1  Joseph  beer
         2   Mary  bread

         pd.merge(df6, df7, how='outer')
              name   food drink
         0   Peter   fish   NaN
         1    Paul  beans   NaN
         2    Mary  bread  wine
         3  Joseph    NaN  beer
```

Левое соединение (left join) и *правое соединение* (right join) возвращают соединение по записям слева и справа соответственно. Например:

```
In [16]: display('df6', 'df7', "pd.merge(df6, df7, how='left')")
Out[16]: df6                    df7
             name   food            name drink
         0  Peter   fish       0    Mary  wine
         1   Paul  beans       1  Joseph  beer
         2   Mary  bread
```

```
        pd.merge(df6, df7, how='left')
            name   food drink
        0  Peter   fish   NaN
        1   Paul  beans   NaN
        2   Mary  bread  wine
```

Записи в результате теперь соответствуют записям в левом входном объекте. Аргумент `how='right'` работает аналогично.

Все эти параметры можно непосредственно применять ко всем вышеописанным типам соединений.

Пересекающиеся имена столбцов: именованный аргумент suffixes

Иногда может встретиться случай, когда в двух входных объектах `DataFrame` присутствуют столбцы с конфликтующими именами. Рассмотрим следующий пример:

```
In [17]: df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                             'rank': [1, 2, 3, 4]})
         df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                             'rank': [3, 1, 4, 2]})
         display('df8', 'df9', 'pd.merge(df8, df9, on="name")')
Out[17]: df8                 df9
            name  rank          name  rank
         0   Bob     1       0   Bob     3
         1  Jake     2       1  Jake     1
         2  Lisa     3       2  Lisa     4
         3   Sue     4       3   Sue     2

         pd.merge(df8, df9, on="name")
            name  rank_x  rank_y
         0   Bob       1       3
         1  Jake       2       1
         2  Lisa       3       4
         3   Sue       4       2
```

Поскольку в результате должны были быть два столбца с конфликтующими именами, функция `merge` автоматически добавила в названия суффиксы `_x` и `_y`, чтобы обеспечить уникальность имен столбцов. Если подобное поведение по умолчанию неуместно, то можно задать свои суффиксы с помощью именованного аргумента `suffixes`:

```
In [18]: pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
Out[18]:    name  rank_L  rank_R
         0   Bob       1       3
         1  Jake       2       1
         2  Lisa       3       4
         3   Sue       4       2
```

Эти суффиксы подходят для всех возможных вариантов соединений, в том числе и при наличии нескольких столбцов с пересекающимися именами.

В главе 20 мы глубже погрузимся в реляционную алгебру. Более подробную информацию по этому вопросу вы найдете в разделе «Merge, Join, Concatenate and Compare» («Слияние, соединение, конкатенация и сравнение») в документации Pandas (https://oreil.ly/l8zZ1).

Пример: данные по штатам США

Операции слияния и соединения чаще используются для объединения данных из различных источников. Здесь мы рассмотрим пример с определенными данными по штатам США и их населению (https://oreil.ly/aq6Xb).

```
In [19]: # Ниже приводятся команды для скачивания данных
         # repo = "https://raw.githubusercontent.com/jakevdp/data-USstates/master"
         # !cd data && curl -O {repo}/state-population.csv
         # !cd data && curl -O {repo}/state-areas.csv
         # !cd data && curl -O {repo}/state-abbrevs.csv
```

Посмотрим на эти три набора данных с помощью функции `read_csv` библиотеки Pandas:

```
In [20]: pop = pd.read_csv('data/state-population.csv')
         areas = pd.read_csv('data/state-areas.csv')
         abbrevs = pd.read_csv('data/state-abbrevs.csv')

         display('pop.head()', 'areas.head()', 'abbrevs.head()')
Out[20]: pop.head()
           state/region     ages  year  population
         0           AL  under18  2012   1117489.0
         1           AL    total  2012   4817528.0
         2           AL  under18  2010   1130966.0
         3           AL    total  2010   4785570.0
         4           AL  under18  2011   1125763.0

         areas.head()
                 state  area (sq. mi)
         0     Alabama          52423
         1      Alaska         656425
         2     Arizona         114006
         3    Arkansas          53182
         4  California         163707

         abbrevs.head()
                 state abbreviation
         0     Alabama           AL
```

```
        1      Alaska          AK
        2     Arizona          AZ
        3    Arkansas          AR
        4  California          CA
```

Допустим, нам нужно на основе этой информации отсортировать штаты и территории США по плотности населения в 2010 году. Информации для этого у нас достаточно, но для достижения цели придется объединить наборы данных.

Начнем со слияния «многие-к-одному», чтобы поместить полные имена штатов в объект `DataFrame` со значениями плотности населения. Выполнить слияние нужно на основе столбца `state/region` объекта `pop` и столбца `abbreviation` объекта `abbrevs`. Воспользуемся аргументом `how='outer'`, чтобы гарантировать, что никакие данные не будут упущены из-за несовпадения меток:

```
In [21]: merged = pd.merge(pop, abbrevs, how='outer',
                           left_on='state/region', right_on='abbreviation')
         merged = merged.drop('abbreviation', axis=1) # удалить повторяющуюся
                                                      # информацию
         merged.head()
Out[21]:   state/region     ages  year  population    state
         0           AL  under18  2012   1117489.0  Alabama
         1           AL    total  2012   4817528.0  Alabama
         2           AL  under18  2010   1130966.0  Alabama
         3           AL    total  2010   4785570.0  Alabama
         4           AL  under18  2011   1125763.0  Alabama
```

Давайте проверим, не было ли каких-то несовпадений. Сделать это можно путем поиска строк с пустыми значениями:

```
In [22]: merged.isnull().any()
Out[22]: state/region    False
         ages            False
         year            False
         population       True
         state            True
         dtype: bool
```

Часть информации о населении отсутствует, выясним, какая именно:

```
In [23]: merged[merged['population'].isnull()].head()
Out[23]:      state/region     ages  year  population state
         2448           PR  under18  1990         NaN   NaN
         2449           PR    total  1990         NaN   NaN
         2450           PR    total  1991         NaN   NaN
         2451           PR  under18  1991         NaN   NaN
         2452           PR    total  1993         NaN   NaN
```

Похоже, что источник пустых значений по населению — Пуэрто-Рико до 2000 года. Вероятно, это произошло из-за того, что необходимых данных не было в первоисточнике.

Что еще более важно, некоторые из новых записей имеют пустые значения в столбце `state`, а это означает, что в объекте `abbrevs` нет соответствующих записей! Выясним, для каких территорий отсутствуют требуемые значения:

```
In [24]: merged.loc[merged['state'].isnull(), 'state/region'].unique()
Out[24]: array(['PR', 'USA'], dtype=object)
```

Все ясно: данные по численности населения включают записи для Пуэрто-Рико (PR) и США в целом (USA), отсутствующие в `abbrevs`. Исправим это, вставив соответствующие записи:

```
In [25]: merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
         merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
         merged.isnull().any()
Out[25]: state/region    False
         ages            False
         year            False
         population       True
         state           False
         dtype: bool
```

Готово! Теперь в столбце `state` нет пустых значений.

Далее можно объединить полученный результат с данными по площади штатов, выполнив аналогичную процедуру. После исследования имеющихся результатов нетрудно догадаться, что нужно выполнить соединение обоих объектов по столбцу `state`:

```
In [26]: final = pd.merge(merged, areas, on='state', how='left')
         final.head()
Out[26]:   state/region     ages  year  population    state  area (sq. mi)
         0           AL  under18  2012   1117489.0  Alabama        52423.0
         1           AL    total  2012   4817528.0  Alabama        52423.0
         2           AL  under18  2010   1130966.0  Alabama        52423.0
         3           AL    total  2010   4785570.0  Alabama        52423.0
         4           AL  under18  2011   1125763.0  Alabama        52423.0
```

И снова проверим на пустые значения, чтобы узнать, были ли какие-то несовпадения:

```
In [27]: final.isnull().any()
Out[27]: state/region     False
         ages             False
         year             False
         population        True
         state            False
         area (sq. mi)     True
         dtype: bool
```

В столбце `area` имеются пустые значения. Посмотрим, какие территории не были учтены:

```
In [28]: final['state'][final['area (sq. mi)'].isnull()].unique()
Out[28]: array(['United States'], dtype=object)
```

Как видите, наш объект `areas` не содержит площади США в целом. Мы могли бы вставить это значение (например, сложив площади всех штатов), но в данном случае мы просто удалим пустые значения, поскольку плотность населения США в целом нас сейчас не интересует:

```
In [29]: final.dropna(inplace=True)
         final.head()
Out[29]:   state/region     ages  year  population    state  area (sq. mi)
         0           AL  under18  2012   1117489.0  Alabama        52423.0
         1           AL    total  2012   4817528.0  Alabama        52423.0
         2           AL  under18  2010   1130966.0  Alabama        52423.0
         3           AL    total  2010   4785570.0  Alabama        52423.0
         4           AL  under18  2011   1125763.0  Alabama        52423.0
```

Теперь у нас есть все необходимые данные. Чтобы ответить на интересующий вопрос, сначала выберем часть данных, соответствующих 2010 году и всему населению. Воспользуемся функцией `query` (для этого должен быть установлен пакет NumExpr, см. главу 24):

```
In [30]: data2010 = final.query("year == 2010 & ages == 'total'")
         data2010.head()
Out[30]:     state/region   ages  year  population       state  area (sq. mi)
         3             AL  total  2010   4785570.0     Alabama        52423.0
         91            AK  total  2010    713868.0      Alaska       656425.0
         101           AZ  total  2010   6408790.0     Arizona       114006.0
         189           AR  total  2010   2922280.0    Arkansas        53182.0
         197           CA  total  2010  37333601.0  California       163707.0
```

Теперь вычислим плотность населения и выведем данные в соответствующем порядке. Начнем с переиндексации наших данных по штату, после чего вычислим результат:

```
In [31]: data2010.set_index('state', inplace=True)
         density = data2010['population'] / data2010['area (sq. mi)']

In [32]: density.sort_values(ascending=False, inplace=True)
         density.head()
Out[32]: state
         District of Columbia    8898.897059
         Puerto Rico             1058.665149
         New Jersey              1009.253268
         Rhode Island             681.339159
         Connecticut              645.600649
         dtype: float64
```

Результат — список штатов США плюс Вашингтон (округ Колумбия) и Пуэрто-Рико, упорядоченный по плотности населения в 2010 году, в жителях на квадратную милю. Как видите, самая густонаселенная территория в этом наборе данных — Вашингтон (округ Колумбия); среди штатов же самый густонаселенный — Нью-Джерси.

Можно также вывести последние записи в списке:

```
In [33]: density.tail()
Out[33]: state
         South Dakota    10.583512
         North Dakota     9.537565
         Montana          6.736171
         Wyoming          5.768079
         Alaska           1.087509
         dtype: float64
```

Как видите, наименьшая плотность населения, причем с большим отрывом от остальных, наблюдается в штате Аляска: в среднем она равна одному жителю на квадратную милю.

Подобное слияние данных — распространенная задача при исследовании данных, получаемых из реальных источников. Надеюсь, что этот пример дал вам представление, как можно комбинировать вышеописанные инструменты, чтобы почерпнуть полезную информацию из данных!

ГЛАВА 20

Агрегирование и группировка

Важная часть анализа больших данных — эффективное обобщение: вычисление сводных показателей, например `sum`, `mean`, `median`, `min` и `max`, в которых одно число позволяет понять природу, возможно, огромного набора данных. В этой главе мы займемся изучением приемов вычисления сводных показателей, предлагаемых библиотекой Pandas, начиная с простых операций, подобных тем, с которыми мы уже имели дело при работе с массивами NumPy, и заканчивая более сложными операциями на основе понятия `groupby`.

Для удобства мы используем ту же функцию `display` из предыдущих глав:

```
In [1]: import pandas as pd
        import numpy as np

        class display(object):
            """Отображает несколько объектов в формате HTML"""
            template = """<div style="float: left; padding: 10px;">
            <p style='font-family:"Courier New", Courier, monospace'>{0}{1}
            """
            def __init__(self, *args):
                self.args = args

            def _repr_html_(self):
                return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                                 for a in self.args)

            def __repr__(self):
                return '\n\n'.join(a + '\n' + repr(eval(a))
                                   for a in self.args)
```

Данные о планетах

Воспользуемся набором данных «Планеты» (Planets), доступным в пакете Seaborn (http://seaborn.pydata.org/; см. главу 36). Он включает информацию о планетах, открытых астрономами и вращающихся вокруг других звезд, известных как *внесолнечные планеты* или *экзопланеты*. Скачать его можно с помощью самого пакета Seaborn:

```
In [2]: import seaborn as sns
        planets = sns.load_dataset('planets')
        planets.shape
Out[2]: (1035, 6)
In [3]: planets.head()
Out[3]:            method  number  orbital_period   mass  distance  year
        0  Radial Velocity       1         269.300   7.10     77.40  2006
        1  Radial Velocity       1         874.774   2.21     56.95  2008
        2  Radial Velocity       1         763.000   2.60     19.84  2011
        3  Radial Velocity       1         326.030  19.40    110.62  2007
        4  Radial Velocity       1         516.220  10.50    119.47  2009
```

Этот набор данных содержит определенную информацию о более чем 1000 экзопланет, открытых до 2014 года.

Простое агрегирование в библиотеке Pandas

В главе 7 мы рассмотрели некоторые возможности агрегирования данных, доступные для массивов NumPy. Как и в случае одномерных массивов NumPy, объекты `Series` в библиотеке Pandas имеют агрегирующие функции, которые возвращают скалярные значения:

```
In [4]: rng = np.random.RandomState(42)
        ser = pd.Series(rng.rand(5))
        ser
Out[4]: 0    0.374540
        1    0.950714
        2    0.731994
        3    0.598658
        4    0.156019
        dtype: float64

In [5]: ser.sum()
Out[5]: 2.811925491708157

In [6]: ser.mean()
Out[6]: 0.5623850983416314
```

Агрегирующие функции объекта `DataFrame` по умолчанию возвращают сводные показатели по каждому столбцу:

```
In [7]: df = pd.DataFrame({'A': rng.rand(5),
                           'B': rng.rand(5)})
        df
Out[7]:           A         B
        0  0.155995  0.020584
        1  0.058084  0.969910
        2  0.866176  0.832443
        3  0.601115  0.212339
        4  0.708073  0.181825

In [8]: df.mean()
Out[8]: A    0.477888
        B    0.443420
        dtype: float64
```

Однако есть возможность агрегировать по строкам, задав аргумент `axis`:

```
In [9]: df.mean(axis='columns')
Out[9]: 0    0.088290
        1    0.513997
        2    0.849309
        3    0.406727
        4    0.444949
        dtype: float64
```

Объекты `Series` и `DataFrame` имеют методы, соответствующие всем распространенным агрегирующим функциям, упомянутым в главе 7. Кроме того, они имеют удобный метод `describe`, вычисляющий сразу несколько самых распространенных сводных показателей для каждого столбца и возвращающий результат. Опробуем его на наборе данных «Планеты», пока удалив строки с отсутствующими значениями:

```
In [10]: planets.dropna().describe()
Out[10]:           number  orbital_period        mass    distance         year
         count  498.00000      498.000000  498.000000  498.000000   498.000000
         mean     1.73494      835.778671    2.509320   52.068213  2007.377510
         std      1.17572     1469.128259    3.636274   46.596041     4.167284
         min      1.00000        1.328300    0.003600    1.350000  1989.000000
         25%      1.00000       38.272250    0.212500   24.497500  2005.000000
         50%      1.00000      357.000000    1.245000   39.940000  2009.000000
         75%      2.00000      999.600000    2.867500   59.332500  2011.000000
         max      6.00000    17337.500000   25.000000  354.000000  2014.000000
```

Эта возможность очень удобна для первоначального знакомства с общими характеристиками набора данных. Например, как можно заметить в столбце `year`, первая экзопланета была открыта еще в 1989 году, однако половина всех известных экзопланет открыта после 2009 года. В значительной степени мы обязаны запуску космического телескопа «Кеплер», специально созданного для поиска затмений от планет, вращающихся вокруг других звезд.

В табл. 20.1 перечислены основные агрегирующие методы, реализованные в библиотеке Pandas.

Таблица 20.1. Список агрегирующих методов библиотеки Pandas

Агрегирующая функция	Описание
`count`	Общее количество элементов
`first`, `last`	Первый и последний элементы
`mean`, `median`	Среднее значение и медиана
`min`, `max`	Минимум и максимум
`std`, `var`	Стандартное отклонение и дисперсия
`mad`	Среднее абсолютное отклонение
`prod`	Произведение всех элементов
`sum`	Сумма всех элементов

Это все методы объектов `DataFrame` и `Series`.

Для более глубокого исследования данных простых сводных показателей часто недостаточно. Следующий уровень обобщения данных — операция `groupby`, позволяющая быстро и эффективно вычислять сводные показатели по подмножествам данных.

groupby: разбиение, применение, объединение

Простые агрегирующие функции помогают «прочувствовать» набор данных, но зачастую бывает нужно выполнить условное агрегирование по какой-либо метке или индексу. Это действие реализовано в так называемой операции `groupby`. Название *group by* («сгруппировать по») ведет начало от одноименной команды в языке SQL баз данных, но, возможно, будет понятнее говорить о ней в терминах, придуманных Хэдли Викхэмом, более известным своими библиотеками для языка R: *разбиение*, *применение* и *объединение*.

Разбиение, применение и объединение

Канонический пример операции «разбить, применить, объединить», в которой «применить» — обобщающее агрегирование, показан на рис. 20.1.

Диаграмма на рис. 20.1 демонстрирует, что именно делает операция `groupby`.

- Шаг *разбиения* включает разделение на части и группировку данных в объекте `DataFrame` на основе значений заданного ключа.

- Шаг *применения* включает вычисление какой-либо функции, обычно агрегирование, преобразование или фильтрацию в пределах отдельных групп.
- На шаге *объединения* выполняется слияние результатов этих операций в выходной массив.

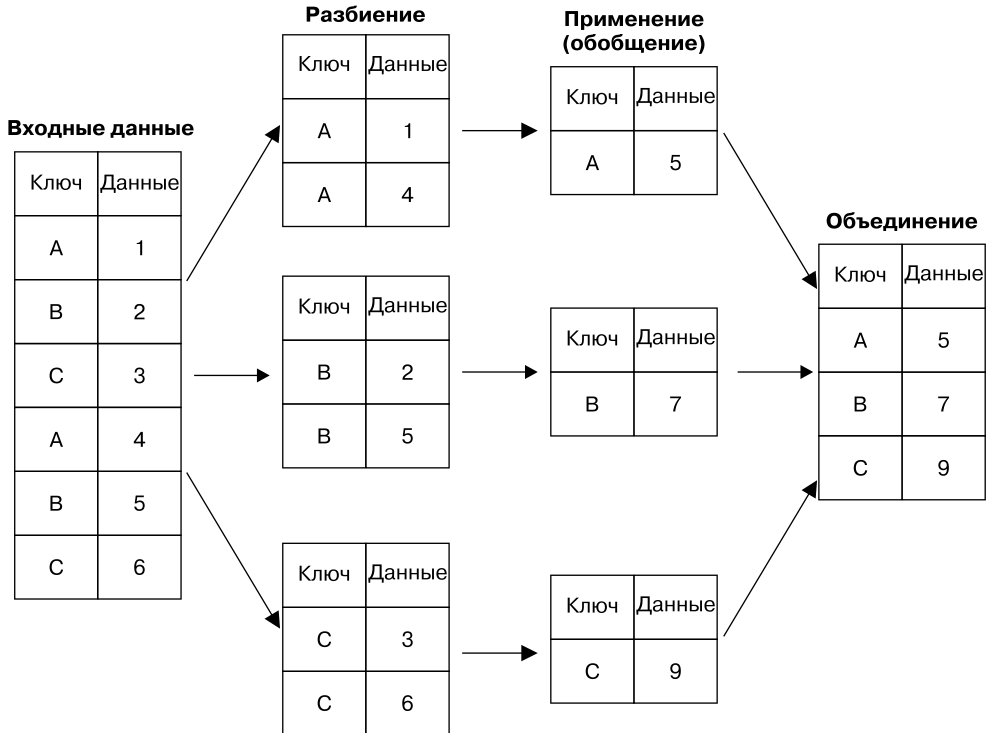


Рис. 20.1. Визуальное представление операции groupby[1]

Мы, конечно, могли бы сделать это вручную с помощью какого-либо сочетания описанных выше команд маскирования, агрегирования и слияния, но важно понимать, что *не обязательно создавать объекты для промежуточных разбиений*. Операция `groupby` может проделать все это за один проход по данным, вычисляя сумму, среднее значение, количество, минимум и другие сводные показатели для каждой группы. Мощь операции `groupby` заключается в абстрагировании этих шагов: пользователю не нужно заботиться о том, *как* фактически выполняются вычисления, вместо этого он может думать об *операции в целом*.

[1] Код, генерирующий эту диаграмму, можно найти в онлайн-приложении по адресу: https://oreil.ly/zHqzu.

В качестве примера рассмотрим использование библиотеки Pandas для выполнения показанных на рис. 20.1 вычислений. Начнем с создания входного объекта `DataFrame`:

```
In [11]: df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                            'data': range(6)}, columns=['key', 'data'])
         df
Out[11]:  key  data
         0   A     0
         1   B     1
         2   C     2
         3   A     3
         4   B     4
         5   C     5
```

Простейшую операцию «разбить, применить, объединить» можно реализовать с помощью метода `groupby` объекта `DataFrame`, передав ему имя желаемого ключевого столбца:

```
In [12]: df.groupby('key')
Out[12]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x11d241e20>
```

Обратите внимание, что он возвращает не набор объектов `DataFrame`, а объект `DataFrameGroupBy`. Этот объект особенный, его можно рассматривать как специальное представление объекта `DataFrame`, готовое к группировке, но не выполняющее никаких фактических вычислений до этапа применения агрегирования. Подобный метод «отложенных вычислений» дает возможность реализовать очень эффективные агрегирующие функции, причем практически прозрачным для пользователя образом.

Для получения результата можно вызвать один из агрегирующих методов этого объекта `DataFrameGroupBy`, что приведет к выполнению соответствующих шагов применения/объединения:

```
In [13]: df.groupby('key').sum()
Out[13]:      data
         key
         A       3
         B       5
         C       7
```

Применение метода `sum` — лишь один из возможных вариантов. Здесь можно использовать практически любую распространенную агрегирующую функцию из библиотек Pandas и NumPy, равно как и практически любую корректную операцию с объектом `DataFrame`.

Объект GroupBy

Объект `GroupBy` — очень гибкая абстракция. Во многом с ним можно обращаться как с коллекцией объектов `DataFrame`, и вся сложность будет скрыта от пользователя. Рассмотрим примеры на основе набора данных «Планеты».

Вероятно, самые важные из операций, доступных благодаря объекту GroupBy, — *агрегирование*, *фильтрация*, *преобразование* и *применение*. Мы обсудим каждую из них более подробно в следующем разделе, но сначала познакомимся еще с одной особенностью, которую можно использовать вместе с базовой операцией `GroupBy`.

Индексация по столбцам

Объект `GroupBy` поддерживает операцию индексации по столбцам, аналогичную похожей операции в объекте `DataFrame`, и возвращает модифицированный объект `GroupBy`. Например:

```
In [14]: planets.groupby('method')
Out[14]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x11d1bc820>

In [15]: planets.groupby('method')['orbital_period']
Out[15]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x11d1bcd60>
```

Здесь мы выбрали конкретную группу `Series` из исходной группы `DataFrame`, сославшись на соответствующее имя столбца. Как и в случае с объектом `GroupBy`, никаких вычислений не происходит до вызова какого-нибудь агрегирующего метода для этого объекта:

```
In [16]: planets.groupby('method')['orbital_period'].median()
Out[16]: method
         Astrometry                         631.180000
         Eclipse Timing Variations         4343.500000
         Imaging                          27500.000000
         Microlensing                      3300.000000
         Orbital Brightness Modulation        0.342887
         Pulsar Timing                       66.541900
         Pulsation Timing Variations       1170.000000
         Radial Velocity                    360.200000
         Transit                              5.714932
         Transit Timing Variations           57.011000
         Name: orbital_period, dtype: float64
```

Результат дает нам общее представление о масштабе чувствительности каждого из методов к периодам обращения (в днях).

Цикл по группам

Объект `GroupBy` поддерживает непосредственное выполнение циклов по группам с возвратом каждой группы в виде объекта `Series` или `DataFrame`:

```
In [17]: for (method, group) in planets.groupby('method'):
             print("{0:30s} shape={1}".format(method, group.shape))
Out[17]: Astrometry                     shape=(2, 6)
         Eclipse Timing Variations      shape=(9, 6)
         Imaging                        shape=(38, 6)
         Microlensing                   shape=(23, 6)
         Orbital Brightness Modulation  shape=(3, 6)
         Pulsar Timing                  shape=(5, 6)
         Pulsation Timing Variations    shape=(1, 6)
         Radial Velocity                shape=(553, 6)
         Transit                        shape=(397, 6)
         Transit Timing Variations      shape=(4, 6)
```

Это может пригодиться для исследования групп вручную в процессе отладки, хотя обычно проще воспользоваться встроенной функциональностью `apply`, которую мы обсудим ниже.

Методы диспетчеризации

Благодаря определенной магии классов языка Python все методы, не реализованные объектом `GroupBy` явно, будут передаваться далее и выполняться для групп вне зависимости от того, являются ли они объектами `Series` или `DataFrame`. Например, вот как можно использовать метод `describe` объекта `DataFrame` для вычисления набора сводных показателей, описывающих каждую группу в данных:

```
In [18]: planets.groupby('method')['year'].describe().unstack()
Out[18]:        method
         count  Astrometry                          2.0
                Eclipse Timing Variations           9.0
                Imaging                            38.0
                Microlensing                       23.0
                Orbital Brightness Modulation       3.0
                                                  ...
         max    Pulsar Timing                    2011.0
                Pulsation Timing Variations      2007.0
                Radial Velocity                  2014.0
                Transit                          2014.0
                Transit Timing Variations        2014.0
         Length: 80, dtype: float64
```

Эта таблица позволяет получить более полное представление о данных. Например, большинство планет было открыто методом измерения лучевой скорости (radial velocity method) и транзитным методом (transit method), хотя последний стал распространенным благодаря новым более точным телескопам только в последнее

десятилетие. Похоже, что новейшими методами являются метод вариации времени транзитов (transit timing variation method) и метод модуляции орбитальной яркости (orbital brightness modulation method), которые до 2011 года не использовались для открытия новых планет.

Это всего лишь один пример полезности методов диспетчеризации. Обратите внимание, что они применяются *к каждой отдельной группе*, после чего результаты объединяются в объект `GroupBy` и возвращаются. Для объекта `GroupBy` можно вызывать любые допустимые методы объектов `Series`/`DataFrame`.

Агрегирование, фильтрация, преобразование, применение

Предыдущее обсуждение касалось агрегирования применительно к операции объединения, но доступны и другие возможности. В частности, у объектов `GroupBy` имеются методы `aggregate`, `filter`, `transform` и `apply`, эффективно выполняющие множество полезных операций до объединения сгруппированных данных.

В следующих подразделах мы будем использовать объект `DataFrame`:

```
In [19]: rng = np.random.RandomState(0)
         df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                            'data1': range(6),
                            'data2': rng.randint(0, 10, 6)},
                            columns = ['key', 'data1', 'data2'])
         df
Out[19]:   key  data1  data2
         0   A      0      5
         1   B      1      0
         2   C      2      3
         3   A      3      3
         4   B      4      7
         5   C      5      9
```

Агрегирование

Мы уже знаем, что объект `GroupBy` предлагает методы агрегирования `sum`, `median` и т. п., но метод `aggregate` обеспечивает еще большую гибкость. Он может принимать строку, функцию или список и вычислять все сводные показатели сразу. Вот пример, включающий все вышеупомянутое:

```
In [20]: df.groupby('key').aggregate(['min', np.median, max])
Out[20]:     data1            data2
               min median max   min median max
         key
         A       0    1.5   3     3    4.0   5
         B       1    2.5   4     0    3.5   7
         C       2    3.5   5     3    6.0   9
```

Еще один удобный паттерн — передача словаря, отображающего имена столбцов в операции, которые должны быть применены к этим столбцам:

```
In [21]: df.groupby('key').aggregate({'data1': 'min',
                                      'data2': 'max'})
Out[21]:      data1  data2
         key
         A        0      5
         B        1      7
         C        2      9
```

Фильтрация

Операция фильтрации позволяет отбрасывать данные в зависимости от свойств группы. Например, нам может понадобиться оставить в результате все группы со стандартным отклонением, превышающим некоторое критическое значение:

```
In [22]: def filter_func(x):
             return x['data2'].std() > 4

         display('df', "df.groupby('key').std()",
                 "df.groupby('key').filter(filter_func)")
Out[22]: df                            df.groupby('key').std()
           key  data1  data2              data1     data2
         0   A      0      5           key
         1   B      1      0           A    2.12132  1.414214
         2   C      2      3           B    2.12132  4.949747
         3   A      3      3           C    2.12132  4.242641
         4   B      4      7
         5   C      5      9

         df.groupby('key').filter(filter_func)
           key  data1  data2
         1   B      1      0
         2   C      2      3
         4   B      4      7
         5   C      5      9
```

Функция `filter` возвращает булево значение, определяющее, прошла ли группа фильтрацию. В данном случае, поскольку стандартное отклонение группы `A` превышает 4, она удаляется из итогового результата.

Преобразование

Агрегирующая функция должна возвращать сокращенную версию данных, однако преобразование может вернуть версию полного набора данных, преобразованных и подготовленных для дальнейшей перекомпоновки. При подобном преобразовании формы входных и выходных данных совпадают.

Распространенный пример — центрирование данных путем вычитания среднего значения по группам:

```
In [23]: def center(x):
             return x - x.mean()
         df.groupby('key').transform(center)
Out[23]:    data1  data2
         0   -1.5    1.0
         1   -1.5   -3.5
         2   -1.5   -3.0
         3    1.5   -1.0
         4    1.5    3.5
         5    1.5    3.0
```

Метод apply()

Метод `apply` дает возможность применить произвольную функцию к результатам группировки. В качестве параметра эта функция должна принимать объект `DataFrame` и возвращать объект библиотеки Pandas (например, `DataFrame` или `Series`) или скалярное значение. В зависимости от возвращаемого значения будет вызвана соответствующая операция объединения.

Например, функция `apply`, нормирующая первый столбец на сумму значений второго:

```
In [24]: def norm_by_data2(x):
             # x -- объект DataFrame сгруппированных значений
             x['data1'] /= x['data2'].sum()
             return x

         df.groupby('key').apply(norm_by_data2)
Out[24]:   key     data1  data2
         0   A  0.000000      5
         1   B  0.142857      0
         2   C  0.166667      3
         3   A  0.375000      3
         4   B  0.571429      7
         5   C  0.416667      9
```

Функция `apply` в `GroupBy` достаточно гибкая. Единственное требование — она должна принимать объект `DataFrame` и возвращать объект библиотеки Pandas или скалярное значение; внутренняя реализация остается на ваше усмотрение!

Задание ключа разбиения

В представленных ранее простых примерах мы разбивали объект `DataFrame` по одному столбцу. Это лишь один из многих вариантов определения способа группировки, и далее мы рассмотрим некоторые другие возможности.

Список, массив, объект Series и индекс как ключи группировки

Ключ может быть любым рядом или списком такой же длины, что и объект `DataFrame`. Например:

```
In [25]: L = [0, 1, 0, 1, 2, 0]
         df.groupby(L).sum()
Out[25]:    data1  data2
         0      7     17
         1      4      3
         2      4      7
```

Разумеется, это значит, что есть еще один, несколько более длинный способ выполнить вышеприведенную операцию `df.groupby('key')`:

```
In [26]: df.groupby(df['key']).sum()
Out[26]:      data1  data2
         key
         A        3      8
         B        5      7
         C        7     12
```

Словарь или объект Series, связывающий индекс и группу

Еще один метод — указать словарь, отображающий значения индексов в ключи группировки:

```
In [27]: df2 = df.set_index('key')
         mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
         display('df2', 'df2.groupby(mapping).sum()')
Out[27]: df2                    df2.groupby(mapping).sum()
             data1  data2                  data1  data2
         key                    key
         A        0      5      consonant     12     19
         B        1      0      vowel          3      8
         C        2      3
         A        3      3
         B        4      7
         C        5      9
```

Любая функция на Python

В вызов `groupby` можно также передать любую функцию, принимающую значение индекса и возвращающую группу:

```
In [28]: df2.groupby(str.lower).mean()
Out[28]:      data1  data2
         key
         a      1.5    4.0
         b      2.5    3.5
         c      3.5    6.0
```

Список допустимых ключей

Любые из предыдущих вариантов ключей можно комбинировать для группировки по мультииндексу:

```
In [29]: df2.groupby([str.lower, mapping]).mean()
Out[29]:                  data1  data2
         key key
         a   vowel         1.5    4.0
         b   consonant     2.5    3.5
         c   consonant     3.5    6.0
```

Пример группировки

В качестве примера соберем все это вместе в нескольких строках кода на языке Python и подсчитаем количество открытых планет по методу открытия и десятилетию:

```
In [30]: decade = 10 * (planets['year'] // 10)
         decade = decade.astype(str) + 's'
         decade.name = 'decade'
         planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
Out[30]: decade                         1980s  1990s  2000s  2010s
         method
         Astrometry                       0.0    0.0    0.0    2.0
         Eclipse Timing Variations        0.0    0.0    5.0   10.0
         Imaging                          0.0    0.0   29.0   21.0
         Microlensing                     0.0    0.0   12.0   15.0
         Orbital Brightness Modulation    0.0    0.0    0.0    5.0
         Pulsar Timing                    0.0    9.0    1.0    1.0
         Pulsation Timing Variations      0.0    0.0    1.0    0.0
         Radial Velocity                  1.0   52.0  475.0  424.0
         Transit                          0.0    0.0   64.0  712.0
         Transit Timing Variations        0.0    0.0    0.0    9.0
```

Этот пример демонстрирует возможности комбинирования нескольких операций при анализе реальных наборов данных. Мы мгновенно получили представление о том, когда и как открывались экзопланеты в последние несколько десятилетий!

Теперь я предлагаю углубиться в эти несколько строк кода и выполнить их по шагам, чтобы убедиться, что вы действительно понимаете, какой вклад в результат они вносят. Это в чем-то непростой пример, но благодаря хорошему пониманию кода у вас появятся средства для исследования собственных данных.

ГЛАВА 21

Сводные таблицы

Мы уже видели, какие возможности по исследованию отношений в наборе данных предоставляет абстракция `groupby`. *Сводная таблица* (pivot table) — схожая операция, часто встречающаяся в электронных таблицах и других программах, работающих с табличными данными. Сводная таблица получает на входе простые данные в виде столбцов и группирует записи в двумерную таблицу, обеспечивающую многомерное представление данных. Различие между сводными таблицами и операцией `groupby` иногда неочевидно. Лично мне помогает представлять сводные таблицы как *многомерную* версию агрегирующей функции `groupby`. То есть вы выполняете операцию «разбить, применить, объединить», но как разбиение, так и объединение происходят не на одномерном индексе, а на двумерной координатной сетке.

Примеры для изучения приемов работы со сводными таблицами

В качестве примера в этой главе мы используем базу данных пассажиров парохода «Титаник», доступную в пакете Seaborn (см. главу 36):

```
In [1]: import numpy as np
        import pandas as pd
        import seaborn as sns
        titanic = sns.load_dataset('titanic')

In [2]: titanic.head()
Out[2]:    survived  pclass     sex   age  sibsp  parch     fare embarked  class  \
        0         0       3    male  22.0      1      0   7.2500        S  Third
        1         1       1  female  38.0      1      0  71.2833        C  First
        2         1       3  female  26.0      0      0   7.9250        S  Third
        3         1       1  female  35.0      1      0  53.1000        S  First
        4         0       3    male  35.0      0      0   8.0500        S  Third
```

```
     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
```

Этот набор данных содержит информацию о каждом пассажире злополучного рейса, включая пол, возраст, класс, стоимость билета и многое другое.

Сводные таблицы «вручную»

Чтобы узнать о данных больше, можно начать с группировки пассажиров по полу, информации о том, выжил ли пассажир, или какой-то их комбинации. Если вы читали предыдущую главу, то можете воспользоваться операцией `groupby`. Например, посмотрим на коэффициент выживаемости в зависимости от пола:

```
In [3]: titanic.groupby('sex')[['survived']].mean()
Out[3]:         survived
        sex
        female  0.742038
        male    0.188908
```

Эти результаты сразу же дают нам некоторое представление о наборе данных: в целом три четверти находившихся на борту женщин выжило, в то время как из мужчин выжил только каждый пятый!

Однако хотелось бы заглянуть немного глубже и увидеть распределение выживших по полу и классу. Говоря языком `groupby`, можно пойти следующим путем: *сгруппировать* по классу и полу, *выбрать* выживших, *применить* агрегирующую функцию вычисления среднего значения, *объединить* получившиеся группы, после чего выполнить операцию *unstack* иерархического индекса, чтобы обнажить скрытую многомерность. Вот все это в виде кода:

```
In [4]: titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
Out[4]: class      First    Second     Third
        sex
        female  0.968085  0.921053  0.500000
        male    0.368852  0.157407  0.135447
```

Эти результаты дают нам еще более полное представление о том, как пол и класс повлияли на выживаемость, но код начинает выглядеть несколько запутанным. Каждый шаг этого конвейера выглядит вполне осмысленным в свете ранее рассмотренных инструментов, но такая длинная строка кода не особенно удобна для чтения или использования. Двумерная операция `groupby` встречается настолько часто, что в библиотеку Pandas был включен удобный метод `pivot_table`, позволяющий более кратко описать данную разновидность многомерного агрегирования.

Синтаксис сводных таблиц

Вот код, эквивалентный вышеприведенной операции, использующий метод `pivot_table` объекта `DataFrame`:

```
In [5]: titanic.pivot_table('survived', index='sex', columns='class',
aggfunc='mean')
Out[5]: class      First    Second     Third
        sex
        female  0.968085  0.921053  0.500000
        male    0.368852  0.157407  0.135447
```

Такая запись несравненно удобнее для чтения, чем подход с `groupby`, при том же результате. Как и следовало ожидать от трансатлантического круиза начала XX века, судьба благоприятствовала женщинам, путешествовавшим первым классом. Женщины из первого класса выжили практически все (привет, Роуз!), из мужчин, путешествовавших третьим классом, выжила только десятая часть (извини, Джек!).

Многоуровневые сводные таблицы

Группировку в сводных таблицах, как и при операции `groupby`, можно задавать на нескольких уровнях и со множеством параметров. Например, интересно взглянуть на возраст в качестве третьего измерения. Разобьем данные на интервалы по возрасту с помощью функции `pd.cut`:

```
In [6]: age = pd.cut(titanic['age'], [0, 18, 80])
        titanic.pivot_table('survived', ['sex', age], 'class')
Out[6]: class               First    Second     Third
        sex    age
        female (0, 18]   0.909091  1.000000  0.511628
               (18, 80]  0.972973  0.900000  0.423729
        male   (0, 18]   0.800000  0.600000  0.215686
               (18, 80]  0.375000  0.071429  0.133663
```

Ту же стратегию можно применить при работе со столбцами. Добавим сюда информацию о стоимости билета, воспользовавшись функцией `pd.qcut` для автоматического вычисления квантилей:

```
In [7]: fare = pd.qcut(titanic['fare'], 2)
        titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
Out[7]: fare            (-0.001, 14.454]                     (14.454, 512.329]  \
        class                      First    Second     Third             First
        sex    age
        female (0, 18]               NaN  1.000000  0.714286          0.909091
               (18, 80]              NaN  0.880000  0.444444          0.972973
        male   (0, 18]               NaN  0.000000  0.260870          0.800000
               (18, 80]              0.0  0.098039  0.125000          0.391304
```

```
fare
class              Second     Third
sex    age
female (0, 18]   1.000000  0.318182
       (18, 80]  0.914286  0.391304
male   (0, 18]   0.818182  0.178571
       (18, 80]  0.030303  0.192308
```

В результате получается четырехмерная сводная таблица с иерархическими индексами (см. главу 17), показанная в сетке, демонстрирующей отношения между значениями.

Дополнительные параметры сводных таблиц

Полная сигнатура метода `DataFrame.pivot_table` выглядит следующим образом:

```
# сигнатура в версии Pandas 1.3.5
DataFrame.pivot_table(data, values=None, index=None, columns=None,
                      aggfunc='mean', fill_value=None, margins=False,
                      dropna=True, margins_name='All', observed=False,
                      sort=True)
```

Мы уже видели примеры первых трех аргументов, в данном подразделе рассмотрим остальные. Параметры `fill_value` и `dropna` определяют, как обрабатывать отсутствующие значения, и интуитивно понятны, примеры их использования мы приводить не будем.

Именованный аргумент `aggfunc` задает тип агрегирования — по умолчанию среднее значение. Как и в `groupby`, агрегирующая функция может задаваться строкой с одним из обычных вариантов (`'sum'`, `'mean'`, `'count'`, `'min'`, `'max'` и т. д.) или именем функции, реализующей агрегирование (`np.sum`, `min`, `sum` и т. п.). Кроме того, агрегирование можно задать в виде словаря, отображающего столбец в любой из вышеперечисленных вариантов:

```
In [8]: titanic.pivot_table(index='sex', columns='class',
                            aggfunc={'survived':sum, 'fare':'mean'})
Out[8]:              fare                         survived
        class        First     Second      Third    First Second Third
        sex
        female  106.125798  21.970121  16.118810       91     70    72
        male     67.226127  19.741782  12.661633       45     17    47
```

Обратите внимание, что, задав аргумент `aggfunc`, мы опустили аргумент `values`, так как в этом случае он определяется автоматически.

Иногда бывает полезно вычислять итоги по каждой группе. Это можно сделать, передав аргумент `margins`:

```
In [9]: titanic.pivot_table('survived', index='sex', columns='class', margins=True)
Out[9]: class      First    Second     Third       All
        sex
        female  0.968085  0.921053  0.500000  0.742038
        male    0.368852  0.157407  0.135447  0.188908
        All     0.629630  0.472826  0.242363  0.383838
```

Такие итоги автоматически дают нам информацию о выживаемости вне зависимости от класса, коэффициенте выживаемости по классу вне зависимости от пола и общем коэффициенте выживаемости 38 %. Метки для этих итогов можно задать с помощью аргумента `margins_name`, по умолчанию имеющего значение `"All"`.

Пример: данные о рождаемости

В качестве еще одного примера взглянем на находящиеся в открытом доступе данные о рождаемости в США (https://oreil.ly/2NWnk), предоставляемые центрами по контролю заболеваний (Centers for Disease Control, CDC). Этот набор данных довольно широко исследовался Эндрю Гелманом и его группой (см., например, сообщение в блоге https://oreil.ly/5EqEp)[1]:

```
In [10]: # команды для скачивания данных:
         # !cd data && curl -O \
         # https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv

In [11]: births = pd.read_csv('data/births.csv')
```

Взглянув на эти данные, можно обнаружить их относительную простоту — они содержат количество новорожденных, сгруппированных по дате и полу:

```
In [12]: births.head()
Out[12]:    year  month  day gender  births
         0  1969      1  1.0      F    4046
         1  1969      1  1.0      M    4440
         2  1969      1  2.0      F    4454
         3  1969      1  2.0      M    4548
         4  1969      1  3.0      F    4548
```

Мы начнем понимать эти данные немного лучше, воспользовавшись сводной таблицей. Добавим в нее столбец для десятилетия и взглянем на рождение девочек и мальчиков как функцию от десятилетия:

[1] В наборе данных CDC используется пол, указанный при рождении, где он называется «гендером» (gender). Гендер может иметь только одно из двух значений: мужской (male) и женский (female). Несмотря на то что пол — это целый спектр, не зависящий от биологии, для согласованности и ясности при обсуждении этого набора данных я буду придерживаться привычной терминологии.

```
In [13]: births['decade'] = 10 * (births['year'] // 10)
         births.pivot_table('births', index='decade', columns='gender',
                            aggfunc='sum')
Out[13]: gender         F         M
         decade
         1960     1753634   1846572
         1970    16263075  17121550
         1980    18310351  19243452
         1990    19479454  20420553
         2000    18229309  19106428
```

Сразу же видим, что в каждом десятилетии мальчиков рождается больше, чем девочек. Воспользуемся встроенными средствами построения графиков в библиотеке Pandas для визуализации общего количества новорожденных в зависимости от года, как показано на рис. 21.1 (см. обсуждение построения графиков с помощью библиотеки Matplotlib в части IV):

```
In [14]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')
         births.pivot_table(
             'births', index='year', columns='gender', aggfunc='sum').plot()
         plt.ylabel('total births per year');
         # Общее количество новорожденных за год
```

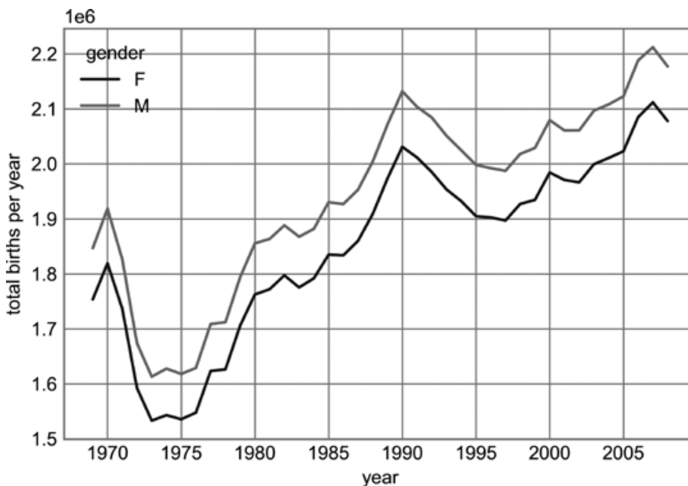


Рис. 21.1. Общее количество новорожденных в США в зависимости от года и пола

Благодаря простой сводной таблице и методу `plot` можно сразу же увидеть ежегодный тренд новорожденных по полу. В последние 50 с лишним лет мальчиков рождалось больше, чем девочек, примерно на 5 %.

Хотя это, возможно, и не имеет отношения к сводным таблицам, но в библиотеке Pandas есть еще несколько интересных инструментов, позволяющих извлечь дополнительную информацию из этого набора данных. Для начала немного почистим данные, удалив аномальные значения, возникшие из-за неправильно набранных дат (например, 31 июня) или отсутствующих значений (например, 99 июня). Простой способ убрать их все сразу — отсечь аномальные значения. Мы сделаем это с помощью надежного алгоритма сигма-отсечения (sigma-clipping):

```
In [15]: quartiles = np.percentile(births['births'], [25, 50, 75])
         mu = quartiles[1]
         sig = 0.74 * (quartiles[2] - quartiles[0])
```

В последней строке вычисляется грубая оценка среднего значения по выборке, где 0,74 — межквартильный размах гауссова распределения (узнать больше об операциях сигма-отсечения можно в книге, которую я написал совместно с Желько Ивечичем, Эндрю Дж. Конолли и Александром Греем: *Statistics, Data Mining and Machine Learning in Astronomy: A Practical Python Guide for the Analysis of Survey Data* (Princeton University Press, 2014)).

Теперь можно воспользоваться методом `query` (обсуждается в главе 24) для фильтрации строк, в которых количество новорожденных выходит за пределы этих значений:

```
In [16]: births = births.query('(births > @mu - 5 * @sig) &
                                (births < @mu + 5 * @sig)')
```

Далее установим целочисленный тип для столбца `day`. Ранее он был строчным, потому что некоторые столбцы в наборе данных содержат значение `'null'`:

```
In [17]: # делаем тип столбца 'day' целочисленным;
         # изначально он был строчным из-за пустых значений
         births['day'] = births['day'].astype(int)
```

Наконец, создадим индекс для даты, объединив день, месяц и год (см. главу 23). Это даст нам возможность быстро вычислять день недели для каждой строки:

```
In [18]: # создаем индекс для даты из года, месяца и дня
         births.index = pd.to_datetime(10000 * births.year +
                                       100 * births.month +
                                       births.day, format='%Y%m%d')

         births['dayofweek'] = births.index.dayofweek
```

С помощью следующего кода построим график дней рождения в зависимости от дня недели за несколько десятилетий (рис. 21.2):

```
In [19]: import matplotlib.pyplot as plt
         import matplotlib as mpl
```

```
births.pivot_table('births', index='dayofweek',
                    columns='decade', aggfunc='mean').plot()
plt.gca().set(xticks=range(7),
              xticklabels=['Mon', 'Tues', 'Wed', 'Thurs',
                           'Fri', 'Sat', 'Sun'])
plt.ylabel('mean births by day');
# Среднее количество новорожденных за день
```

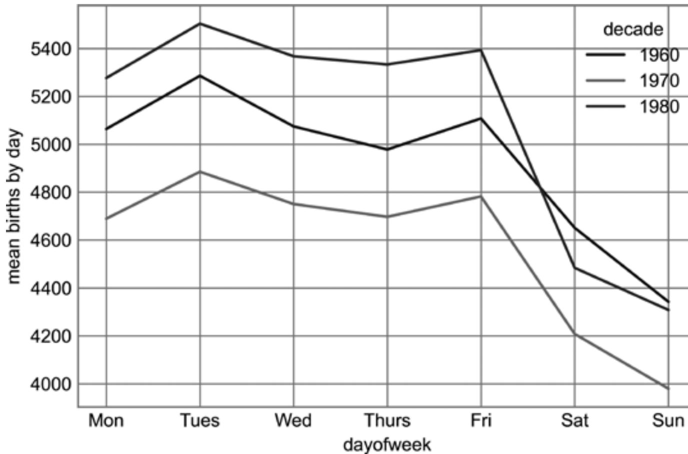


Рис. 21.2. Среднее количество новорожденных за день в зависимости от дня недели и десятилетия

Очевидно, что в выходные дни число рождений меньше, чем в будни! Обратите внимание, что 1990-е и 2000-е годы отсутствуют на графике, потому что начиная с 1989 года данные CDC содержат только месяц рождения.

Еще одно интересное представление этих данных можно получить, построив график рождений в зависимости от дня года. Сначала сгруппируем данные отдельно по месяцу и дню:

```
In [20]: births_by_date = births.pivot_table('births',
                        [births.index.month, births.index.day])
         births_by_date.head()
Out[20]:        births
         1 1  4009.225
           2  4247.400
           3  4500.900
           4  4571.350
           5  4603.625
```

Результат представляет собой мультииндекс по месяцам и дням. Чтобы упростить построение графика, преобразуем месяцы и дни в даты, связав их с фиктивным годом (обязательно выбирайте високосный год, чтобы корректно обработать 29 февраля!):

```
In [21]: from datetime import datetime
         births_by_date.index = [datetime(2012, month, day)
                                 for (month, day) in births_by_date.index]
         births_by_date.head()
Out[21]:              births
         2012-01-01  4009.225
         2012-01-02  4247.400
         2012-01-03  4500.900
         2012-01-04  4571.350
         2012-01-05  4603.625
```

Сфокусировавшись на дне и месяце, мы получаем временной ряд, отражающий среднее количество новорожденных в зависимости от дня года. Исходя из этого, можно с помощью метода plot построить график (рис. 21.3). В нем, как видите, обнаруживаются некоторые любопытные тренды.

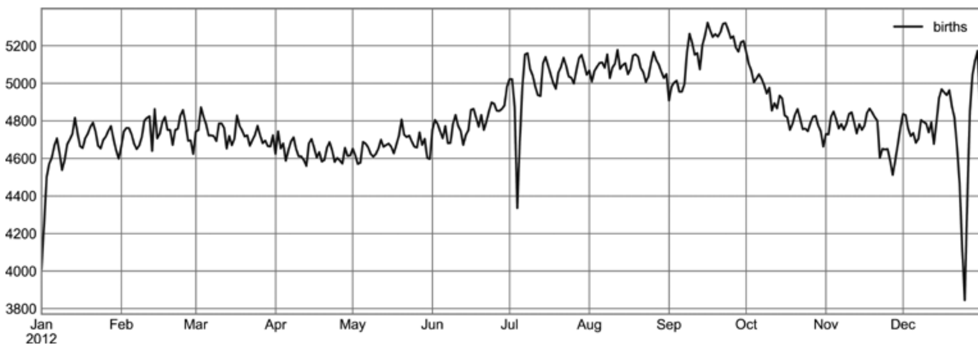


Рис. 21.3. Среднее количество новорожденных по датам

```
In [22]: # Визуализируем результат
            fig, ax = plt.subplots(figsize=(12, 4))
            births_by_date.plot(ax=ax);
```

В частности, удивительным выглядит резкое падение количества рождений в государственные праздники США (например, День независимости, День труда, День благодарения, Рождество, Новый год). Хотя оно отражает скорее тенденции, относящиеся к заранее запланированным/искусственным родам, а не глубокое психосоматическое влияние на естественные роды. Дальнейшее обсуждение данной тенденции, ее анализ и ссылки на эту тему вы найдете в статье (https://oreil.ly/ugVHI) в блоге Эндрю Гелмана. Мы вернемся к этому графику в главе 32, где используем инструменты из библиотеки Matplotlib для добавления меток на график.

Рассматривая этот краткий пример, вы могли заметить, что многие из представленных инструментов Python и Pandas можно комбинировать между собой и использовать, чтобы почерпнуть полезную информацию из наборов данных. В следующих главах мы увидим более сложные манипуляции с данными!

ГЛАВА 22

Векторизованные операции над строками

Одна из сильных сторон языка Python — относительная простота работы со строковыми данными. Библиотека Pandas тоже вносит свою лепту и предоставляет набор *векторизованных операций над строками*, широко используемых для очистки данных, необходимой при работе с реальными данными. В этой главе мы перечислим некоторые строковые операции, реализованные в библиотеке Pandas, а затем рассмотрим их применение для частичной очистки очень зашумленного набора данных рецептов, собранных в Интернете.

Знакомство со строковыми операциями в библиотеке Pandas

В предыдущих главах мы видели, как библиотеки NumPy и Pandas обобщают арифметические операции, позволяя легко и быстро выполнять одну и ту же операцию над множеством элементов массива. Например:

```
In [1]: import numpy as np
        x = np.array([2, 3, 5, 7, 11, 13])
        x * 2
Out[1]: array([ 4,  6, 10, 14, 22, 26])
```

Векторизация упрощает синтаксис операций с массивами данных, избавляя от необходимости беспокоиться о размере или форме массива и позволяя сосредоточиться только на нужной операции. Библиотека NumPy не предоставляет такого простого способа доступа для массивов строк, поэтому приходится использовать более длинный синтаксис циклов:

```
In [2]: data = ['peter', 'Paul', 'MARY', 'gUIDO']
        [s.capitalize() for s in data]
Out[2]: ['Peter', 'Paul', 'Mary', 'Guido']
```

Для работы с некоторыми данными этого достаточно, но если в наборе могут отсутствовать какие-то значения, то приходится добавлять дополнительные проверки:

```
In [3]: data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
        [s if s is None else s.capitalize() for s in data]
Out[3]: ['Peter', 'Paul', None, 'Mary', 'Guido']
```

Такой подход не только влечет за собой написание лишнего кода, но и чреват ошибками.

Библиотека Pandas включает средства и для поддержки векторизованных строковых операций, и для корректной обработки отсутствующих значений в виде атрибута `str` объектов `Series` и объектов `Index` со строками. Так, например, создав объект `Series` с такими данными, вы сможете вызвать один-единственный метод `str.capitalize` для преобразования строчных букв в заглавные, который будет игнорировать любые отсутствующие значения:

```
In [4]: import pandas as pd
        names = pd.Series(data)
        names.str.capitalize()
Out[4]: 0    Peter
        1     Paul
        2     None
        3     Mary
        4    Guido
        dtype: object
```

Таблица строковых методов в библиотеке Pandas

Если вы хорошо разбираетесь в манипуляциях строковыми данными в языке Python, то львиная доля синтаксиса операций со строками в библиотеке Pandas будет вам интуитивно понятна настолько, что достаточно, наверное, просто привести таблицу имеющихся методов. С этого и начнем, прежде чем углубимся в некоторые нюансы. Примеры в этом разделе используют следующий объект `Series`:

```
In [5]: monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                           'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

Методы, аналогичные строковым методам языка Python

Практически для всех встроенных строковых методов Python в библиотеке Pandas имеются векторизованные эквиваленты. Вот список методов атрибута `str` в библиотеке Pandas, дублирующих строковые методы языка Python:

```
len        lower      translate  islower  ljust
upper      startswith isupper    rjust    find
endswith   isnumeric  center     rfind    isalnum
isdecimal  zfill      index      isalpha  split
strip      rindex     isdigit    rsplit   rstrip
capitalize isspace    partition  lstrip   swapcase
```

Обратите внимание, что они возвращают разные значения. Некоторые, например `lower`, возвращают ряды строк:

```
In [6]: monte.str.lower()
Out[6]: 0    graham chapman
        1       john cleese
        2     terry gilliam
        3         eric idle
        4       terry jones
        5     michael palin
        dtype: object
```

Другие — числовые значения:

```
In [7]: monte.str.len()
Out[7]: 0    14
        1    11
        2    13
        3     9
        4    11
        5    13
        dtype: int64
```

Третьи — булевы значения:

```
In [8]: monte.str.startswith('T')
Out[8]: 0    False
        1    False
        2     True
        3    False
        4     True
        5    False
        dtype: bool
```

А четвертые — списки и другие составные значения для каждого элемента:

```
In [9]: monte.str.split()
Out[9]: 0    [Graham, Chapman]
        1       [John, Cleese]
        2     [Terry, Gilliam]
        3         [Eric, Idle]
        4       [Terry, Jones]
        5     [Michael, Palin]
        dtype: object
```

Далее мы рассмотрим приемы работы с подобными объектами типа «ряды списков».

Методы, использующие регулярные выражения

Помимо перечисленных выше, есть несколько методов, принимающих регулярные выражения, проверяющих содержимое всех строковых элементов и следующих некоторым соглашениям по API встроенного модуля `re` языка Python (табл. 22.1).

Таблица 22.1. Соответствие между методами библиотеки Pandas и функциями модуля re языка Python

Метод	Описание
`match`	Вызывает функцию `re.match` для каждого элемента, возвращая булево значение
`extract`	Вызывает функцию `re.match` для каждого элемента, возвращая подходящие группы в виде строк
`findall`	Вызывает функцию `re.findall` для каждого элемента
`replace`	Заменяет вхождения шаблона какой-либо другой строкой
`contains`	Вызывает функцию `re.search` для каждого элемента, возвращая булево значение
`count`	Подсчитывает вхождения шаблона
`split`	Эквивалент функции `str.split`, но принимающий на входе регулярные выражения
`rsplit`	Эквивалент функции `str.rsplit`, но принимающий на входе регулярные выражения

С помощью этих функций можно выполнять массу интересных операций. Например, можно извлечь имя из каждого элемента, выполнив поиск непрерывной группы символов в начале каждого из них:

```
In [10]: monte.str.extract('([A-Za-z]+)', expand=False)
Out[10]: 0     Graham
         1       John
         2      Terry
         3       Eric
         4      Terry
         5    Michael
         dtype: object
```

Или, например, найти все имена, начинающиеся и заканчивающиеся согласным звуком, воспользовавшись символами регулярных выражений «начало строки» (`^`) и «конец строки» (`$`):

```
In [11]: monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
Out[11]: 0    [Graham Chapman]
         1                  []
         2     [Terry Gilliam]
         3                  []
```

```
        4       [Terry Jones]
        5     [Michael Palin]
        dtype: object
```

Такой сжатый синтаксис применения регулярных выражений для обработки записей в объектах `Series` и `DataFrame` открывает массу возможностей для анализа и очистки данных.

Прочие методы

Наконец, существуют и прочие методы, пригодные для разных удобных операций (табл. 22.2).

Таблица 22.2. Прочие методы для работы со строками библиотеки Pandas

Метод	Описание
`get`	Индексирует все элементы
`slice`	Вырезает подстроку из каждого элемента
`slice_replace`	Заменяет в каждом элементе вырезанную подстроку заданным значением
`cat`	Выполняет конкатенацию строк
`repeat`	Повторяет значения (указанное число раз)
`normalize`	Возвращает версию строки в кодировке Unicode
`pad`	Добавляет пробелы слева, справа или с обеих сторон строки
`wrap`	Разбивает длинные строковые значения на строки с длиной, не превышающей заданную
`join`	Объединяет строки из всех элементов в `Series` с использованием заданного разделителя
`get_dummies`	Извлекает значения переменных-индикаторов в виде объекта `DataFrame`

Векторизованный доступ к элементам и вырезание подстрок

Операции `get` и `slice`, в частности, предоставляют возможность векторизованного доступа к элементам из каждого массива. Например, вызовом `str.slice(0, 3)` можно вырезать первые три символа из каждого элемента массива. Обратите внимание, что такая возможность доступна и с помощью обычного синтаксиса индексации языка Python, например, вызову `df.str.slice(0, 3)` эквивалентно обращение `df.str[0:3]`:

```
In [12]: monte.str[0:3]
Out[12]: 0    Gra
         1    Joh
         2    Ter
```

```
        3    Eri
        4    Ter
        5    Mic
        dtype: object
```

Индексация посредством `df.str.get(i)` и `df.str[i]` происходит аналогично.

Методы с поддержкой индексирования также дают возможность обращаться к элементам, возвращаемым методом `split` массивов. Например, вот как с помощью `split` и `str` с индексом можно извлечь фамилии из каждой записи:

```
In [13]: monte.str.split().str[-1]
Out[13]: 0    Chapman
         1     Cleese
         2    Gilliam
         3       Idle
         4      Jones
         5      Palin
         dtype: object
```

Индикаторные переменные

Еще один метод, требующий дополнительных пояснений, — `get_dummies`. Его удобно использовать, когда в данных имеется столбец, содержащий кодированный индикатор. Например, у нас есть набор данных, содержащий информацию в виде кодов, таких как `A` = «родился в США», `B` = «родился в Великобритании», `C` = «любит сыр», `D` = «любит мясные консервы»:

```
In [14]: full_monte = pd.DataFrame({'name': monte,
                                    'info': ['B|C|D', 'B|D', 'A|C',
                                             'B|D', 'B|C', 'B|C|D']})
         full_monte
Out[14]:              name   info
         0  Graham Chapman  B|C|D
         1     John Cleese    B|D
         2   Terry Gilliam    A|C
         3       Eric Idle    B|D
         4     Terry Jones    B|C
         5   Michael Palin  B|C|D
```

Метод `get_dummies` дает возможность быстро выполнить разбивку по всем индикаторным переменным, преобразовав их в объект `DataFrame`:

```
In [15]: full_monte['info'].str.get_dummies('|')
Out[15]:    A  B  C  D
         0  0  1  1  1
         1  0  1  0  1
         2  1  0  1  0
         3  0  1  0  1
         4  0  1  1  0
         5  0  1  1  1
```

Используя эти операции как «строительные блоки», можно создать бесчисленное множество строковых процедур для очистки данных.

Мы не будем углубляться в эти методы, но я рекомендую прочитать раздел *Working with Text Data* («Работа с текстовыми данными») в онлайн-документации библиотеки Pandas (https://oreil.ly/oYgWA) или заглянуть в раздел «Дополнительные источники информации» главы 24.

Пример: база данных рецептов

Описанные векторизованные строковые операции, как оказывается, с успехом можно использовать для очистки сильно зашумленных данных. В этом разделе мы рассмотрим пример такой очистки базы данных рецептов, собранной из множества различных источников в Интернете. Наша цель — преобразовать рецепты в списки ингредиентов, чтобы можно было быстро находить рецепты по списку имеющихся продуктов. Сценарии, что использовались для сбора рецептов, можно найти на GitHub (https://oreil.ly/3S0Rg), как и ссылку на актуальную версию базы.

Размер базы данных составляет около 30 Мбайт, и ее можно скачать и разархивировать с помощью следующих команд:

```
In [16]: # repo = "https://raw.githubusercontent.com/jakevdp/open-recipe-data/master"
         # !cd data && curl -O {repo}/recipeitems.json.gz
         # !gunzip data/recipeitems.json.gz
```

Данные хранятся в формате JSON, так что можно попробовать воспользоваться функцией `pd.read_json` для их чтения (обязательно добавьте аргумент `lines=True`, потому что каждая строка в файле — это отдельная запись в формате JSON):

```
In [17]: recipes = pd.read_json('data/recipeitems.json', lines=True)
         recipes.shape
Out[17]: (173278, 17)
```

Как видите, в базе данных собрано почти 175 тысяч рецептов и имеется 17 столбцов. Рассмотрим одну из строк, чтобы понять, что мы имеем:

```
In [18]: recipes.iloc[0]
Out[18]: _id                                {'$oid': '5160756b96cc62079cc2db15'}
         name                                    Drop Biscuits and Sausage Gravy
         ingredients          Biscuits\n3 cups All-purpose Flour\n2 Tablespo...
         url                  http://thepioneerwoman.com/cooking/2013/03/dro...
         image                http://static.thepioneerwoman.com/cooking/file...
         ts                                             {'$date': 1365276011104}
         cookTime                                                          PT30M
         source                                                  thepioneerwoman
         recipeYield                                                          12
         datePublished                                                2013-03-11
```

```
        prepTime                                                              PT10M
        description          Late Saturday afternoon, after Marlboro Man ha...
        totalTime                                                               NaN
        creator                                                                 NaN
        recipeCategory                                                          NaN
        dateModified                                                            NaN
        recipeInstructions                                                      NaN
        Name: 0, dtype: object
```

Здесь мы видим немалый объем информации, но большая ее часть хранится неупорядоченно, что очень характерно для данных, извлекаемых из Интернета. В частности, список ингредиентов имеет строковый формат, и нам придется проявить осторожность и настойчивость, чтобы извлечь интересующую нас информацию. Для начала взглянем поближе на ингредиенты:

```
In [19]: recipes.ingredients.str.len().describe()
Out[19]: count    173278.000000
         mean        244.617926
         std         146.705285
         min           0.000000
         25%         147.000000
         50%         221.000000
         75%         314.000000
         max        9067.000000
         Name: ingredients, dtype: float64
```

Средняя длина списка ингредиентов составляет 250 символов при минимальной длине 0 и максимальной — почти 10 000 символов!

Из любопытства посмотрим, какой рецепт имеет самый длинный список ингредиентов:

```
In [20]: recipes.name[np.argmax(recipes.ingredients.str.len())]
Out[20]: 'Carrot Pineapple Spice &amp; Brownie Layer Cake with Whipped Cream &amp;
          > Cream Cheese Frosting and Marzipan Carrots'
```

Этот рецепт выглядит довольно сложным.

Извлекая сводные показатели, можно сделать и другие открытия. Например, посмотрим, сколько рецептов описывают блюда на завтрак (используем синтаксис регулярных выражений, чтобы обеспечить совпадение с заглавными буквами и капителью):

```
In [21]: recipes.description.str.contains('[Bb]reakfast').sum()
Out[21]: 3524
```

А вот количество рецептов, содержащих корицу (cinnamon) в списке ингредиентов:

```
In [22]: recipes.ingredients.str.contains('[Cc]innamon').sum()
Out[22]: 10526
```

Можно даже посмотреть, есть ли рецепты, в которых название этого ингредиента написано с орфографической ошибкой, как *cinamon*:

```
In [23]: recipes.ingredients.str.contains('[Cc]inamon').sum()
Out[23]: 11
```

Такое предварительное знакомство с данными возможно благодаря строковым инструментам, имеющимся в библиотеке Pandas. Именно в сфере такой очистки данных Python действительно силен.

Простая рекомендательная система для рецептов

Немного углубимся в этот пример и начнем создавать простую рекомендательную систему для выбора рецептов, которая будет получать список ингредиентов и отыскивать рецепты, использующие их все. Эта концептуально простая задача усложняется неоднородностью данных: не существует удобной операции, которая позволила бы извлечь из каждой строки очищенный список ингредиентов. Поэтому мы немного сжульничаем: начнем со списка распространенных ингредиентов и будем искать, в каждом ли рецепте они содержатся. Для упрощения ограничимся травами и специями:

```
In [24]: spice_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',
                       'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']
```

Мы можем создать булев объект `DataFrame`, состоящий из значений `True` и `False`, указывающих на присутствие данного ингредиента в списке:

```
In [25]: import re
         spice_df = pd.DataFrame({
             spice: recipes.ingredients.str.contains(spice, re.IGNORECASE)
             for spice in spice_list})
         spice_df.head()
Out[25]:     salt  pepper  oregano   sage  parsley  rosemary  tarragon  thyme   \
         0  False   False    False   True    False     False     False  False
         1  False   False    False  False    False     False     False  False
         2   True    True    False  False    False     False     False  False
         3  False   False    False  False    False     False     False  False
         4  False   False    False  False    False     False     False  False

            paprika   cumin
         0    False   False
         1    False   False
         2    False    True
         3    False   False
         4    False   False
```

Теперь допустим для примера, что мы хотели бы найти рецепт, в котором используются петрушка (parsley), паприка (paprika) и эстрагон (tarragon). Это можно сделать очень быстро, воспользовавшись методом `query` объекта `DataFrame`, который мы обсудим подробнее в главе 24:

```
In [26]: selection = spice_df.query('parsley & paprika & tarragon')
         len(selection)
Out[26]: 10
```

Мы нашли всего десять рецептов с таким сочетанием ингредиентов. Воспользуемся полученным индексом, чтобы выяснить названия этих рецептов:

```
In [27]: recipes.name[selection.index]
Out[27]: 2069      All cremat with a Little Gem, dandelion and wa...
         74964                         Lobster with Thermidor butter
         93768      Burton's Southern Fried Chicken with White Gravy
         113926                     Mijo's Slow Cooker Shredded Beef
         137686                     Asparagus Soup with Poached Eggs
         140530                                 Fried Oyster Po'boys
         158475                Lamb shank tagine with herb tabbouleh
         158486                 Southern fried chicken in buttermilk
         163175              Fried Chicken Sliders with Pickles + Slaw
         165243                        Bar Tartine Cauliflower Salad
         Name: name, dtype: object
```

Теперь, сократив список рецептов со 175 000 до 10, мы можем принять более взвешенное решение: что готовить на обед.

Дальнейшая работа с рецептами

Надеемся, что этот пример позволил вам попробовать на вкус (па-ба-ба-бам!) операции по очистке данных, которые можно эффективно реализовать с помощью строковых методов из библиотеки Pandas. Создание надежной рекомендательной системы для рецептов потребовало бы *намного* больше труда! Важной частью этой задачи было бы извлечение из каждого рецепта полного списка ингредиентов. К сожалению, разнообразие используемых форматов делает этот процесс весьма трудоемким. Как подсказывает нам Капитан Очевидность, в науке о данных очистка полученных из реального мира данных часто представляет собой основную часть работы, и библиотека Pandas предоставляет инструменты для эффективного решения этой задачи.

ГЛАВА 23

Работа с временными рядами

Библиотека Pandas была разработана в расчете на построение финансовых моделей, так что, как и следовало ожидать, она содержит весьма широкий набор инструментов для работы с датой, временем и индексированными по времени данными. Данные о дате и времени могут находиться в нескольких видах, которые мы сейчас обсудим.

- *Метки даты/времени* ссылаются на конкретные моменты времени (например, 4 июля 2021 года в 07:00 утра).
- *Временные интервалы и периоды* ссылаются на отрезки времени между конкретными начальной и конечной точками (например, июнь 2021 года). Периоды обычно представляют собой особый случай с непересекающимися интервалами одинаковой длительности (например, 24-часовые периоды времени, составляющие сутки).
- *Временная дельта (она же продолжительность)* относится к отрезку времени конкретной длительности (например, 22,56 с).

В этой главе мы поговорим об инструментах в библиотеке Pandas для работы с каждым из этих типов временных данных. Глава никоим образом не претендует на звание исчерпывающего руководства по имеющимся в Python или библиотеке Pandas инструментам для работы с временными рядами. Она дает лишь общий обзор приемов работы с временными рядами. Мы начнем с краткого обсуждения инструментов для работы с датой и временем в языке Python, а затем перейдем непосредственно к обсуждению инструментов библиотеки Pandas. После перечисления источников дополнительной информации мы рассмотрим несколько кратких примеров работы с временными рядами с использованием библиотеки Pandas.

Дата и время в языке Python

В мире Python существует немало представлений дат, времени, временных дельт и интервалов времени. Для практической работы с данными наиболее удобны инструменты из библиотеки Pandas, однако нелишним будет посмотреть на другие пакеты, имеющиеся в Python.

Представление даты и времени в Python: пакеты datetime и dateutil

Базовые объекты Python для работы с датами и временем располагаются во встроенном пакете `datetime`. Его вместе со сторонним модулем `dateutil` можно использовать для быстрого выполнения множества операций над датами и временем. Например, с помощью типа `datetime` можно вручную сформировать дату:

```
In [1]: from datetime import datetime
        datetime(year=2021, month=7, day=4)
Out[1]: datetime.datetime(2021, 7, 4, 0, 0)
```

Или, воспользовавшись модулем `dateutil`, выполнить синтаксический разбор дат, находящихся во множестве строковых форматов:

```
In [2]: from dateutil import parser
        date = parser.parse("4th of July, 2021")
        date
Out[2]: datetime.datetime(2021, 7, 4, 0, 0)
```

Получив объект `datetime`, можно узнать день недели:

```
In [3]: date.strftime('%A')
Out[3]: 'Sunday'
```

Здесь мы использовали для вывода даты один из стандартных спецификаторов форматирования строк (`"%A"`), о котором можно прочитать в разделе `strftime` (https://oreil.ly/bjdsf) документации к пакету `datetime` (https://oreil.ly/AGVR9). Описание других полезных утилит для работы с датой и временем можно найти в онлайн-документации пакета `dateutil` (https://oreil.ly/Y5Rwd). Не помешает также познакомиться с родственным пакетом `pytz` (https://oreil.ly/DU9Jr), содержащим инструменты для работы с элементами временных рядов — часовыми поясами.

Сила `datetime` и `dateutil` заключается в их гибкости и удобном синтаксисе: эти объекты и их методы можно использовать для выполнения практически любой интересующей вас операции. Единственное, с чем они плохо справляются, — большие массивы дат и времени: подобно тому как стандартные списки числовых переменных в Python уступают типизированным числовым массивам NumPy, списки объектов `datetime` в Python уступают в производительности типизированным массивам дат.

Типизированные массивы значений времени: тип datetime64 библиотеки NumPy

Тип `datetime64`, реализованный в библиотеке NumPy, кодирует даты как 64-битные целые числа, поэтому массивы дат имеют очень компактное представление. Для создания экземпляра типа `datetime64` требуется передать данные в строго определенном формате:

```
In [4]: import numpy as np
        date = np.array('2021-07-04', dtype=np.datetime64)
        date
Out[4]: array('2021-07-04', dtype='datetime64[D]')
```

Но, получив данные в этом представлении, можно быстро выполнять над ними различные векторизованные операции:

```
In [5]: date + np.arange(12)
Out[5]: array(['2021-07-04', '2021-07-05', '2021-07-06', '2021-07-07',
               '2021-07-08', '2021-07-09', '2021-07-10', '2021-07-11',
               '2021-07-12', '2021-07-13', '2021-07-14', '2021-07-15'],
              dtype='datetime64[D]')
```

Поскольку массивы NumPy со значениями `datetime64` содержат данные одного типа, подобные операции выполняются намного быстрее, чем при работе непосредственно с объектами `datetime` языка Python, что особенно заметно при обработке больших массивов (мы рассматривали эту разновидность векторизации в главе 6).

Важный нюанс относительно объектов `datetime64` и `timedelta64`: они основаны на *базовой единице времени* (fundamental time unit). Поскольку объект `datetime64` ограничен точностью 64 бита, кодируемый им диапазон времени может представлять не более 2^{64} значений. Другими словами, `datetime64` навязывает компромисс между разрешающей способностью по времени и максимальным промежутком времени.

Например, если потребуется разрешающая способность 1 наносекунда, то мы сможем представить интервал протяженностью не более 2^{64} наносекунды, или чуть более 600 лет. Библиотека NumPy определяет требуемую единицу на основе входной информации; например, вот дата/время на основе единицы, равной суткам:

```
In [6]: np.datetime64('2021-07-04')
Out[6]: numpy.datetime64('2021-07-04')
```

Вот дата/время на основе единицы, равной одной минуте:

```
In [7]: np.datetime64('2021-07-04 12:00')
Out[7]: numpy.datetime64('2021-07-04T12:00')
```

Можно принудительно потребовать использовать любую базовую единицу с помощью одного из множества кодов форматирования; например, вот дата/время на основе единицы, равной одной наносекунде:

```
In [8]: np.datetime64('2021-07-04 12:59:59.50', 'ns')
Out[8]: numpy.datetime64('2021-07-04T12:59:59.500000000')
```

В табл. 23.1, взятой из описания `datetime64` в документации к библиотеке NumPy, перечислены поддерживаемые коды форматирования, а также относительные и абсолютные промежутки времени, которые можно представить при их использовании.

Таблица 23.1. Описание кодов форматирования даты и времени

Код	Значение	Промежуток времени (относительный)	Промежуток времени (абсолютный)
`Y`	Год	± 9.2e18 лет	[9.2e18 до н. э., 9.2e18 н. э.]
`M`	Месяц	± 7.6e17 лет	[7.6e17 до н. э., 7.6e17 н. э.]
`W`	Неделя	± 1.7e17 лет	[1.7e17 до н. э., 1.7e17 н. э.]
`D`	День	± 2.5e16 лет	[2.5e16 до н. э., 2.5e16 н. э.]
`h`	Час	± 1.0e15 лет	[1.0e15 до н. э., 1.0e15 н. э.]
`m`	Минута	± 1.7e13 лет	[1.7e13 до н. э., 1.7e13 н. э.]
`s`	Секунда	± 2.9e12 лет	[2.9e9 до н. э., 2.9e9 н. э.]
`ms`	Миллисекунда	± 2.9e9 лет	[2.9e6 до н. э., 2.9e6 н. э.]
`us`	Микросекунда	± 2.9e6 лет	[290301 до н. э., 294241 н. э.]
`ns`	Наносекунда	± 292 лет	[1678 до н. э., 2262 н. э.]
`ps`	Пикосекунда	± 106 дней	[1969 до н. э., 1970 н. э.]
`fs`	Фемтосекунда	± 2.6 часов	[1969 до н. э., 1970 н. э.]
`as`	Аттосекунда	± 9.2 секунды	[1969 до н. э., 1970 н. э.]

Удобное значение по умолчанию для типов данных, встречающихся в реальном мире, — `datetime64[ns]`, позволяющее кодировать достаточный диапазон современных дат с высокой точностью.

Наконец, отметим, что, хотя тип данных `datetime64` лишен некоторых недостатков встроенного типа данных `datetime` языка Python, ему недостает многих предоставляемых `datetime` и особенно `dateutil` удобных методов и функций. Больше информации можно найти в документации по типу `datetime64` библиотеки NumPy (https://oreil.ly/XDbck).

Даты и время в библиотеке Pandas: лучшее из обоих миров

Библиотека Pandas предоставляет объект `Timestamp`, основанный на всех только что обсуждавшихся инструментах и сочетающий удобство использования `datetime` и `dateutil` с эффективным хранением и векторизованным интерфейсом типа `numpy.datetime64`. Библиотека Pandas позволяет создать из нескольких объектов `Timestamp` объект класса `DatetimeIndex`, который можно использовать для индексации данных в объектах `Series` или `DataFrame`.

Инструменты библиотеки Pandas можно применить для воспроизведения вышеприведенной наглядной демонстрации. Вот пример синтаксического разбора строки с датой в гибком формате с применением кодов форматирования, чтобы вывести день недели:

```
In [9]: import pandas as pd
        date = pd.to_datetime("4th of July, 2021")
        date
Out[9]: Timestamp('2021-07-04 00:00:00')

In [10]: date.strftime('%A')
Out[10]: 'Sunday'
```

С этим объектом также можно выполнять векторизованные операции в стиле библиотеки NumPy:

```
In [11]: date + pd.to_timedelta(np.arange(12), 'D')
Out[11]: DatetimeIndex(['2021-07-04', '2021-07-05', '2021-07-06', '2021-07-07',
                        '2021-07-08', '2021-07-09', '2021-07-10', '2021-07-11',
                        '2021-07-12', '2021-07-13', '2021-07-14', '2021-07-15'],
                       dtype='datetime64[ns]', freq=None)
```

В следующем разделе мы подробнее рассмотрим манипуляции с данными временных рядов с помощью инструментов из библиотеки Pandas.

Временные ряды библиотеки Pandas: индексация по времени

Инструменты Pandas для работы с временными рядами особенно удобны при необходимости индексации данных по меткам даты/времени. Например, создадим объект `Series` с индексированными по времени данными:

```
In [12]: index = pd.DatetimeIndex(['2020-07-04', '2020-08-04',
                                   '2021-07-04', '2021-08-04'])
         data = pd.Series([0, 1, 2, 3], index=index)
         data
Out[12]: 2020-07-04    0
         2020-08-04    1
```

```
         2021-07-04    2
         2021-08-04    3
         dtype: int64
```

Теперь, когда данные находятся в объекте `Series`, можно использовать любые паттерны индексации `Series`, обсуждавшиеся в предыдущих главах, передавая значения, которые допускают приведение к типу даты:

```
In [13]: data['2020-07-04':'2021-07-04']
Out[13]: 2020-07-04    0
         2020-08-04    1
         2021-07-04    2
         dtype: int64
```

Имеются также дополнительные специальные операции индексации, предназначенные только для дат. Например, можно указать год, чтобы получить срез всех данных за этот год:

```
In [14]: data['2021']
Out[14]: 2021-07-04    2
         2021-08-04    3
         dtype: int64
```

Позднее мы рассмотрим еще примеры удобства индексации по датам. Но сначала изучим имеющиеся структуры данных для временных рядов.

Структуры данных для временных рядов библиотеки Pandas

В этом разделе мы рассмотрим основные структуры данных, предназначенные для работы с временными рядами.

- Для представления *меток даты/времени* библиотека Pandas предоставляет тип данных `Timestamp`. Этот тип является заменой для встроенного типа данных `datetime` в языке Python, он основан на более эффективном типе данных `numpy.datetime64`. Соответствующая индексная конструкция — `DatetimeIndex`.
- Для представления *периодов времени* библиотека Pandas предоставляет тип данных `Period`. Этот тип, основанный на типе данных `numpy.datetime64`, кодирует интервал времени фиксированной периодичности. Соответствующая индексная конструкция — `PeriodIndex`.
- Для представления *временных дельт (продолжительностей)* библиотека Pandas предоставляет тип данных `Timedelta`, основанный на типе `numpy.timedelta64`, более эффективном, чем встроенный тип данных `datetime.timedelta` в языке Python. Соответствующая индексная конструкция — `TimedeltaIndex`.

Самые основные из этих объектов — `Timestamp` и `DatetimeIndex`. Хотя к ним можно обращаться непосредственно, но чаще для этого используется функция `pd.to_datetime`, способная выполнять синтаксический разбор широкого диапазона форматов. Получив одну дату, функция `pd.to_datetime` возвращает `Timestamp`, а получив ряд дат — `DatetimeIndex`:

```
In [15]: dates = pd.to_datetime([datetime(2021, 7, 3), '4th of July, 2021',
                                 '2021-Jul-6', '07-07-2021', '20210708'])
         dates
Out[15]: DatetimeIndex(['2021-07-03', '2021-07-04', '2021-07-06', '2021-07-07',
                        '2021-07-08'],
                       dtype='datetime64[ns]', freq=None)
```

Любой объект `DatetimeIndex` можно с помощью функции `to_period` преобразовать в объект `PeriodIndex`, указав код для периодичности интервала. В следующем примере используется код `'D'`, означающий периодичность в одни сутки:

```
In [16]: dates.to_period('D')
Out[16]: PeriodIndex(['2021-07-03', '2021-07-04', '2021-07-06', '2021-07-07',
                      '2021-07-08'],
                     dtype='period[D]')
```

Объект `TimedeltaIndex` создается, например, при вычитании одной даты из другой:

```
In [17]: dates - dates[0]
Out[17]: TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
          > dtype='timedelta64[ns]', freq=None)
```

Регулярные последовательности: функция pd.date_range()

Чтобы облегчить создание регулярных последовательностей дат, библиотека Pandas предоставляет несколько функций: `pd.date_range` — для меток даты/времени, `pd.period_range` — для периодов времени и `pd.timedelta_range` — для временных дельт. Мы уже видели, что функции `range` языка Python и `np.arange` из библиотеки NumPy преобразуют начальную точку, конечную точку и (необязательную) величину шага в последовательность. Аналогично функция `pd.date_range` создает регулярную последовательность дат, принимая на входе начальную дату, конечную дату и необязательный код периодичности. По умолчанию период равен одним суткам:

```
In [18]: pd.date_range('2015-07-03', '2015-07-10')
Out[18]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                        '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
                       dtype='datetime64[ns]', freq='D')
```

В качестве альтернативы можно также задать диапазон дат, указав не начальную и конечную точки, а начальную точку и количество периодов времени:

```
In [19]: pd.date_range('2015-07-03', periods=8)
Out[19]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                        '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
                       dtype='datetime64[ns]', freq='D')
```

Можно изменить интервал времени, поменяв аргумент `freq`, имеющий значение по умолчанию `'D'`. Например, следующий пример создает диапазон часовых меток даты/времени:

```
In [20]: pd.date_range('2015-07-03', periods=8, freq='H')
Out[20]: DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',
                        '2015-07-03 02:00:00', '2015-07-03 03:00:00',
                        '2015-07-03 04:00:00', '2015-07-03 05:00:00',
                        '2015-07-03 06:00:00', '2015-07-03 07:00:00'],
                       dtype='datetime64[ns]', freq='H')
```

Для создания регулярных последовательностей значений периодов или временных дельт можно воспользоваться функциями `pd.period_range` и `pd.timedelta_range`, напоминающими функцию `date_range`. Вот несколько периодов времени длительностью в месяц:

```
In [21]: pd.period_range('2015-07', periods=8, freq='M')
Out[21]: PeriodIndex(['2015-07', '2015-08', '2015-09',
                      '2015-10', '2015-11', '2015-12',
                      '2016-01', '2016-02'],
                     dtype='period[M]')
```

Вот последовательность продолжительностей, увеличивающихся на час:

```
In [22]: pd.timedelta_range(0, periods=6, freq='H')
Out[22]: TimedeltaIndex(['0 days 00:00:00', '0 days 01:00:00', '0 days 02:00:00',
                         '0 days 03:00:00', '0 days 04:00:00', '0 days 05:00:00'],
                        dtype='timedelta64[ns]', freq='H')
```

Все эти операции требуют понимания кодов периодичности, описываемых в следующем разделе.

Периодичность и смещение дат

Периодичность или *смещение даты* — базовое понятие для инструментов библиотеки Pandas, необходимых для работы с временными рядами. Аналогично уже продемонстрированным кодам `D` (день) и `H` (час) можно использовать коды для задания любой требуемой периодичности. В табл. 23.2 описаны основные доступные коды.

Таблица 23.2. Список кодов периодичности библиотеки Pandas

Код	Описание	Код	Описание
D	Календарный день	B	Рабочий день
W	Неделя		
M	Конец месяца	BM	Конец отчетного месяца
Q	Конец квартала	BQ	Конец отчетного квартала
A	Конец года	BA	Конец финансового года
H	Час	BH	Рабочие часы
T	Минута		
S	Секунда		
L	Миллисекунда		
U	Микросекунда		
N	Наносекунда		

Периодичности в месяц, квартал и год определяются на конец соответствующего периода. Добавление к любому из них суффикса `S` приводит к определению их на начало периода (табл. 23.3).

Таблица 23.3. Список стартовых кодов периодичности

Код	Описание
MS	Начало месяца
BMS	Начало отчетного месяца
QS	Начало квартала
BQS	Начало отчетного квартала
AS	Начало года
BAS	Начало финансового года

Кроме этого, можно изменить месяц, используемый для определения квартала или года, с помощью добавления в конец кода месяца, состоящего из трех букв:

- `Q-JAN`, `BQ-FEB`, `QS-MAR`, `BQS-APR` и т. д.;
- `A-JAN`, `BA-FEB`, `AS-MAR`, `BAS-APR` и т. д.

Аналогично можно изменить точку разбиения для недельной периодичности, добавив состоящий из трех букв код дня недели: W-SUN, W-MON, W-TUE, W-WED и т. д.

Чтобы указать иную периодичность, можно сочетать коды с числами. Например, для периодичности в 2 часа 30 минут можно скомбинировать коды для часа (H) и минуты (T):

```
In [23]: pd.timedelta_range(0, periods=6, freq="2H30T")
Out[23]: TimedeltaIndex(['0 days 00:00:00', '0 days 02:30:00', '0 days 05:00:00',
                         '0 days 07:30:00', '0 days 10:00:00', '0 days 12:30:00'],
                        dtype='timedelta64[ns]', freq='150T')
```

Все эти короткие коды ссылаются на соответствующие экземпляры смещений даты/времени временных рядов библиотеки Pandas, которые можно найти в модуле pd.tseries.offsets. Например, можно непосредственно создать смещение в один рабочий день:

```
In [24]: from pandas.tseries.offsets import BDay
         pd.date_range('2015-07-01', periods=6, freq=BDay())
Out[24]: DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',
                        '2015-07-07', '2015-07-08'],
                       dtype='datetime64[ns]', freq='B')
```

Дальнейшее обсуждение периодичности и смещений времени можно найти в разделе DateOffset (https://oreil.ly/J6JHA) онлайн-документации библиотеки Pandas.

Передискретизация, временные сдвиги и окна

Возможность использовать дату/время в качестве индексов для интуитивно понятной организации данных и доступа к ним — немаловажная часть инструментария из библиотеки Pandas для работы с временными рядами. При этом сохраняются общие преимущества использования индексированных данных (автоматическое сопоставление в ходе выполнения операций, интуитивно понятные срезы и доступ к данным и т. д.), но библиотека Pandas предоставляет еще несколько дополнительных операций специально для временных рядов.

Мы рассмотрим некоторые из них на примере данных с курсами акций. Библиотека Pandas, будучи разработанной в значительной степени для работы с финансовыми данными, имеет для этой цели несколько весьма специфических инструментов. Например, сопутствующий Pandas пакет pandas-datareader (который можно установить командой conda install pandas-datareader) умеет импортировать финансовые данные из множества источников. В следующем примере мы импортируем историю цен акций из набора данных S&P 500:

```
In [25]: from pandas_datareader import data

         sp500 = data.DataReader('^GSPC', start='2018', end='2022',
                                 data_source='yahoo')
         sp500.head()
Out[25]:                    High          Low         Open        Close      Volume \
         Date
         2018-01-02  2695.889893  2682.360107  2683.729980  2695.810059  3367250000
         2018-01-03  2714.370117  2697.770020  2697.850098  2713.060059  3538660000
         2018-01-04  2729.290039  2719.070068  2719.310059  2723.989990  3695260000
         2018-01-05  2743.449951  2727.919922  2731.330078  2743.149902  3236620000
         2018-01-08  2748.510010  2737.600098  2742.669922  2747.709961  3242650000

                       Adj Close
         Date
         2018-01-02  2695.810059
         2018-01-03  2713.060059
         2018-01-04  2723.989990
         2018-01-05  2743.149902
         2018-01-08  2747.709961
```

Для простоты используем только цену на момент закрытия торгов:

```
In [26]: sp500 = sp500['Close']
```

Нарисовать график изменения цен можно вызовом метода `plot` после обычных инструкций настройки Matplotlib (рис. 23.1):

```
In [27]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')
         sp500.plot();
```

Рис. 23.1. График изменения цены акций S&P 500 на момент закрытия торгов с течением времени

Передискретизация и изменение периодичности интервалов

При работе с данными временных рядов часто бывает необходимо переразбить их с использованием интервалов другой периодичности. Сделать это можно с помощью метода `resample` или гораздо более простого метода `asfreq`. Основная разница между ними заключается в том, что `resample` выполняет *агрегирование данных*, а `asfreq` — *выборку данных*.

Рассмотрим, что возвращают эти два метода при обработке набора данных с ценами акций S&P 500 на момент закрытия торгов при увеличении интервала дискретизации. Здесь мы выполняем передискретизацию данных на конец финансового года (рис. 23.2).

```
In [28]: sp500.plot(alpha=0.5, style='-')
         sp500.resample('BA').mean().plot(style=':')
         sp500.asfreq('BA').plot(style='--');
         plt.legend(['input', 'resample', 'asfreq'],
                    loc='upper left');
```

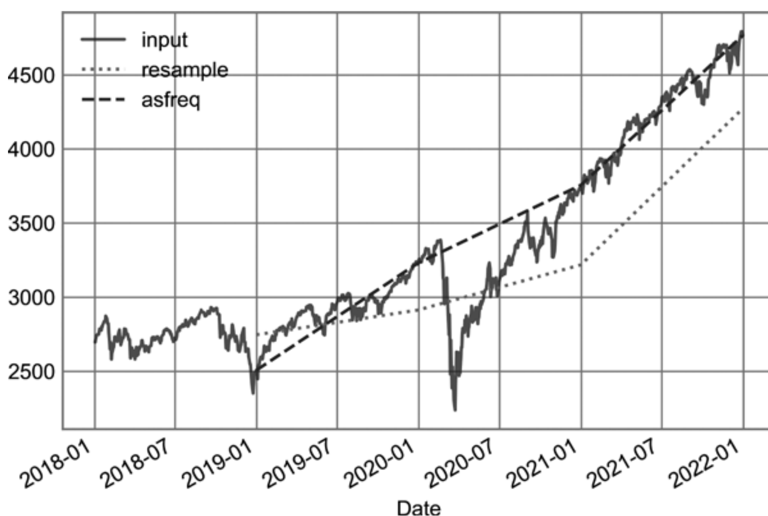


Рис. 23.2. Передискретизация цен акций S&P 500

Обратите внимание на различие: в каждой точке `resample` выдает *среднее значение за предыдущий год*, а `asfreq` — *значение на конец года*.

В случае уменьшения периода дискретизации методы `resample` и `asfreq` в значительной степени идентичны, хотя доступных для использования параметров

у `resample` гораздо больше. В данном случае оба этих метода по умолчанию оставляют интерполированные значения точек пустыми, то есть заполненными значениями NA. Аналогично функции `pd.fillna`, обсуждавшейся в главе 16, метод `asfreq` принимает аргумент `method`, определяющий, откуда будут браться значения для таких точек. Здесь мы передискретизируем данные по рабочим дням с суточной периодичностью, то есть включая выходные дни (рис. 23.3):

```
In [29]: fig, ax = plt.subplots(2, sharex=True)
         data = sp500.iloc[:20]

         data.asfreq('D').plot(ax=ax[0], marker='o')

         data.asfreq('D', method='bfill').plot(ax=ax[1], style='-o')
         data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')
         ax[1].legend(["back-fill", "forward-fill"]);
```

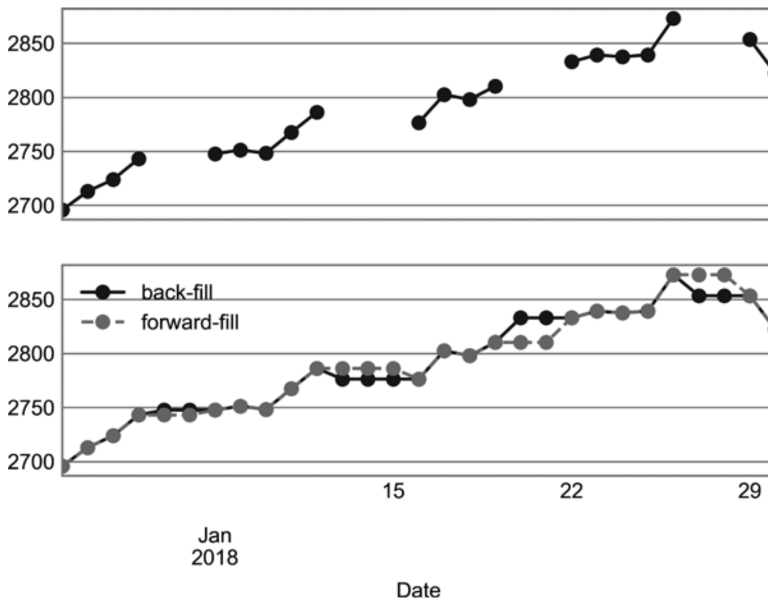


Рис. 23.3. Сравнение интерполяции вперед (forward-fill interpolation) и назад (backward-fill interpolation)

Поскольку набор данных S&P 500 содержит информацию только в рабочие дни, на верхнем графике можно видеть разрывы, соответствующие добавленным значениям NA. Нижний график демонстрирует различия между двумя методиками восстановления отсутствующих данных: интерполяцией вперед (forward-fill interpolation) и интерполяцией назад (back-fill interpolation).

Временные сдвиги

Еще одна распространенная операция с временными рядами — сдвиг данных во времени. Библиотека Pandas предоставляет для этой цели метод `shift`, позволяющий выполнять сдвиг данных на заданное количество элементов. Возможность сдвига временных рядов, отобранных с постоянным шагом, позволяет исследовать тенденции с течением времени.

Например, следующий пример передискретизирует данные, устанавливая суточную периодичность, и выполняет сдвиг на 364, чтобы вычислить годовую доходность инвестиций для S&P 500 с течением времени (рис. 23.4).

```
In [30]: sp500 = sp500.asfreq('D', method='pad')

         ROI = 100 * (sp500.shift(-365) - sp500) / sp500
         ROI.plot()
         plt.ylabel('% Return on Investment after 1 year');
```

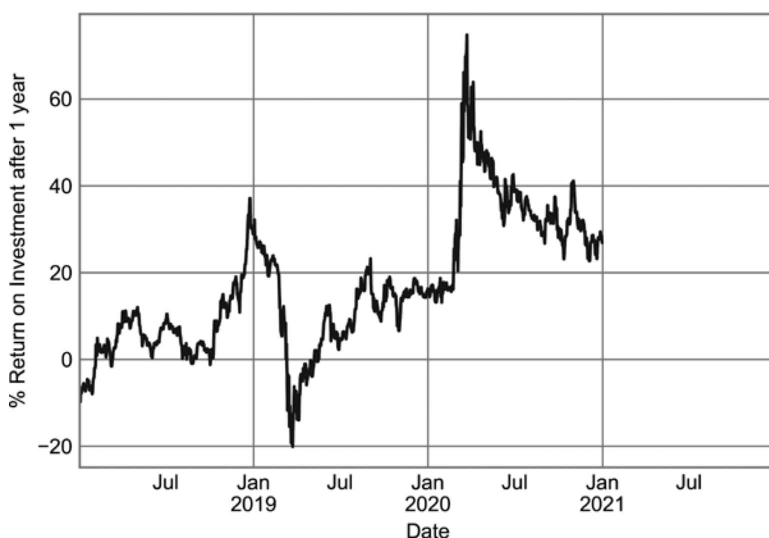


Рис. 23.4. Годовая доходность инвестиций

Худшая годовая доходность наблюдалась примерно в марте 2019 года, когда точно через год случился обвал рынка, обусловленный пандемией коронавируса. Как и следовало ожидать, лучшая годовая доходность наблюдалась в марте 2020 года, когда наиболее предусмотрительные или удачливые купили акции по дешевке.

Скользящие окна

Вычисление скользящих статистических показателей — третья из реализованных в библиотеке Pandas разновидностей операций, предназначенных для использования при анализе временных рядов. Работать с ними можно с помощью атрибута `rolling` объектов `Series` и `DataFrame`, возвращающего представление, подобное тому, с которым мы сталкивались при выполнении операции `groupby` (см. главу 20). Это скользящее представление предоставляет несколько операций агрегирования.

Например, вот годичное скользящее среднее значение и стандартное отклонение цен на акции (рис. 23.5):

```
In [31]: rolling = sp500.rolling(365, center=True)

         data = pd.DataFrame({'input': sp500,
                              'one-year rolling_mean': rolling.mean(),
                              'one-year rolling_median': rolling.median()})
         ax = data.plot(style=['-', '--', ':'])
         ax.lines[0].set_alpha(0.3)
```

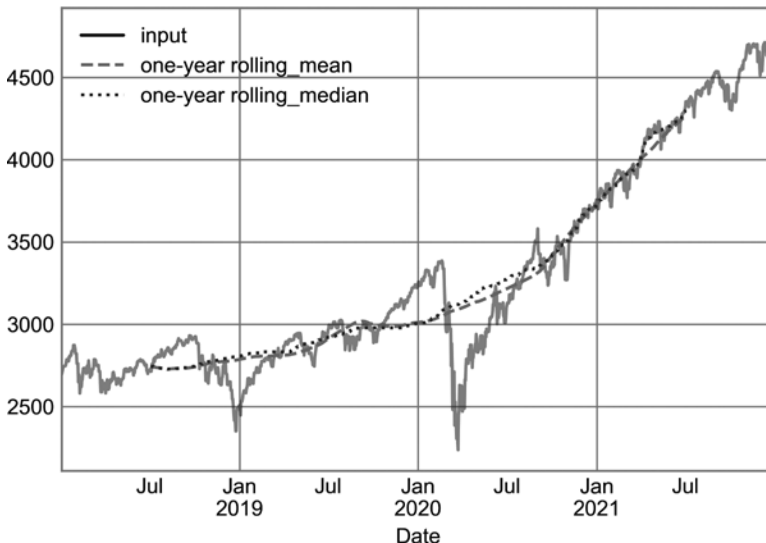


Рис. 23.5. Скользящие статистические показатели для цен на акции S&P 500

Подобно операции `groupby`, методы `aggregate` и `apply` можно использовать для вычисления скользящих показателей.

ГДЕ НАЙТИ ДОПОЛНИТЕЛЬНУЮ ИНФОРМАЦИЮ

В данной главе лишь кратко перечислены некоторые наиболее важные возможности инструментов библиотеки Pandas для работы с временными рядами. Более развернутое обсуждение этой темы можно найти в разделе Time Series/Date Functionality («Обработка временных рядов/дат») онлайн-документации библиотеки Pandas (https://oreil.ly/uC3pB).

Еще один великолепный источник информации — книга *Python for Data Analysis* издательства O'Reilly (https://oreil.ly/ik2g7)[1]. Это бесценный источник информации по использованию библиотеки Pandas. В частности, в книге сделан особый акцент на применении инструментов временных рядов в контексте бизнеса и финансов и уделено больше внимания конкретным деталям бизнес-календаря, работе с часовыми поясами и вопросам, связанным с этим.

Также вы можете воспользоваться справочной функциональностью оболочки IPython для изучения и экспериментов с другими параметрами, имеющимися у обсуждавшихся здесь функций и методов. Я считаю, что это оптимальный способ изучения какого-либо нового инструмента языка Python.

Пример: визуализация количества велосипедов в Сиэтле

В качестве более сложного примера работы с временными рядами рассмотрим подсчет количества велосипедов на Фримонтском мосту в Сиэтле (https://oreil.ly/6qVBt). Эти данные поступают из автоматического счетчика велосипедов, установленного в конце 2012 года, с индуктивными датчиками на восточной и западной боковых дорожках моста. Сведения о почасовом количестве велосипедов можно скачать по адресу http://data.seattle.gov/; набор данных Fremont Bridge Bicycle Counter доступен в категории Transportation (Транспорт).

Файл в формате CSV можно скачать так:

```
In [32]: # url = ('https://raw.githubusercontent.com/jakevdp/'
         #        'bicycle-data/main/FremontBridge.csv')
         # !curl -O {url}
```

После скачивания набора данных его можно прочитать в объект `DataFrame`. Можно указать, что в качестве индекса мы хотим видеть объекты `Date` и чтобы выполнялся автоматический синтаксический разбор этих дат:

```
In [33]: data = pd.read_csv('FremontBridge.csv', index_col='Date', parse_dates=True)
         data.head()
Out[33]:                      Fremont Bridge Total  Fremont Bridge East Sidewalk  \
         Date
         2019-11-01 00:00:00                  12.0                           7.0
```

[1] *Маккини У.* Python и анализ данных.

```
2019-11-01 01:00:00                   7.0                          0.0
2019-11-01 02:00:00                   1.0                          0.0
2019-11-01 03:00:00                   6.0                          6.0
2019-11-01 04:00:00                   6.0                          5.0

                     Fremont Bridge West Sidewalk
Date
2019-11-01 00:00:00                           5.0
2019-11-01 01:00:00                           7.0
2019-11-01 02:00:00                           1.0
2019-11-01 03:00:00                           0.0
2019-11-01 04:00:00                           1.0
```

Для удобства ограничим количество столбцов, участвующих в анализе:

```
In [34]: data.columns = ['Total', 'East', 'West']
```

Теперь рассмотрим сводные статистические показатели для этих данных:

```
In [35]: data.dropna().describe()
Out[35]:                Total           East           West
         count  147255.000000  147255.000000  147255.000000
         mean      110.341462      50.077763      60.263699
         std       140.422051      64.634038      87.252147
         min         0.000000       0.000000       0.000000
         25%        14.000000       6.000000       7.000000
         50%        60.000000      28.000000      30.000000
         75%       145.000000      68.000000      74.000000
         max      1097.000000     698.000000     850.000000
```

Визуализация данных

Мы можем почерпнуть полезную информацию из этого набора данных, визуализировав его. Начнем с построения графика исходных данных (рис. 23.6).

```
In [36]: data.plot()
         plt.ylabel('Hourly Bicycle Count'); # Количество велосипедистов по часам
```

Примерно 150 000 почасовых значений — слишком плотная дискретизация, чтобы можно было понять хоть что-то. Попробуем выполнить передискретизацию данных, увеличив шаг до одной недели (рис. 23.7):

```
In [37]: weekly = data.resample('W').sum()
         weekly.plot(style=['-', ':', '--'])
         plt.ylabel('Weekly bicycle count'); # Количество велосипедистов еженедельно
```

Полученный результат демонстрирует некоторые интересные сезонные тенденции: как и следовало ожидать, летом люди ездят на велосипедах чаще, чем зимой, и даже в пределах каждого из сезонов велосипеды используются с разной интенсивностью в разные недели (вероятно, в зависимости от погоды; см. главу 42, в которой будем

рассматривать этот вопрос). Кроме того, начиная с 2020 года явно видно влияние пандемии COVID-19 на количество поездок.

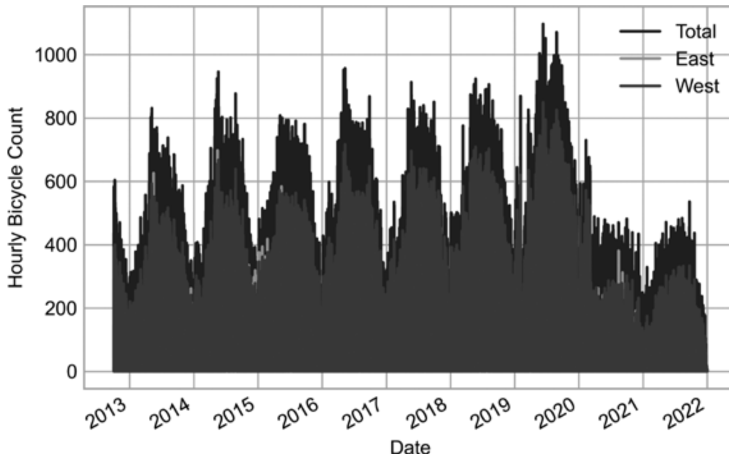


Рис. 23.6. Количество велосипедов, каждый час пересекающих Фримонтский мост в Сиэтле

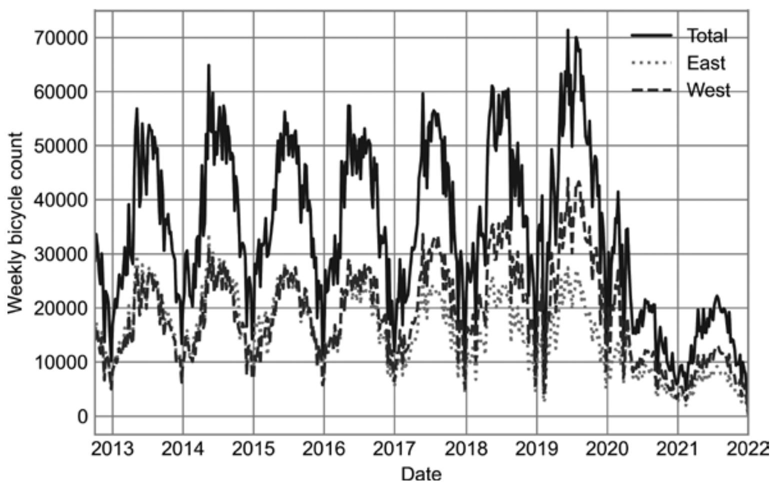


Рис. 23.7. Количество велосипедов, пересекающих Фримонтский мост в Сиэтле, с шагом в одну неделю

Еще один удобный способ агрегирования данных — вычисление скользящего среднего с помощью функции `pd.rolling_mean`. Здесь мы вычисляем для наших данных скользящее среднее за 30 дней с центрированием по окну (рис. 23.8):

```
In [38]: daily = data.resample('D').sum()
         daily.rolling(30, center=True).sum().plot(style=['-', ':', '--'])
         plt.ylabel('mean hourly count');  # Среднее количество по часам
```

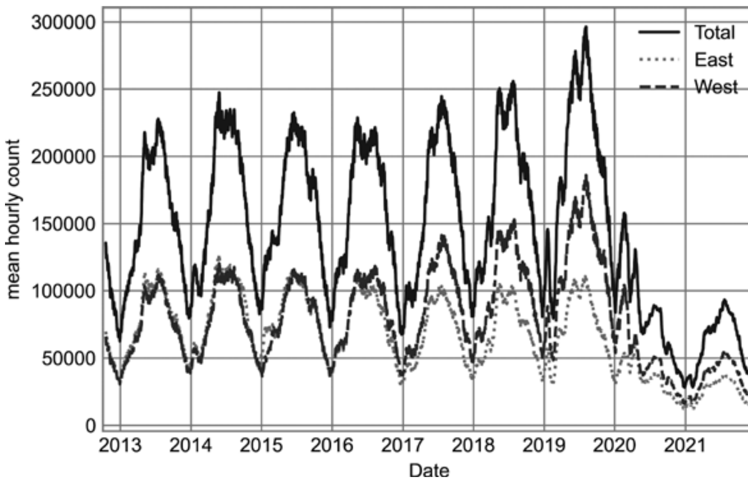


Рис. 23.8. Скользящее среднее значение еженедельного количества велосипедов

Причина зубчатого вида графика — в резкой границе окна. Более гладкую версию скользящего среднего можно получить, воспользовавшись оконной функцией, например гауссовым окном, как показано на рис. 23.9. Следующий код задает как ширину окна (в нашем случае 50 дней), так и ширину гауссовой функции внутри окна (в нашем случае 10 дней):

```
In [39]: daily.rolling(50, center=True,
                       win_type='gaussian').sum(std=10).plot(style=['-', ':', '--']);
```

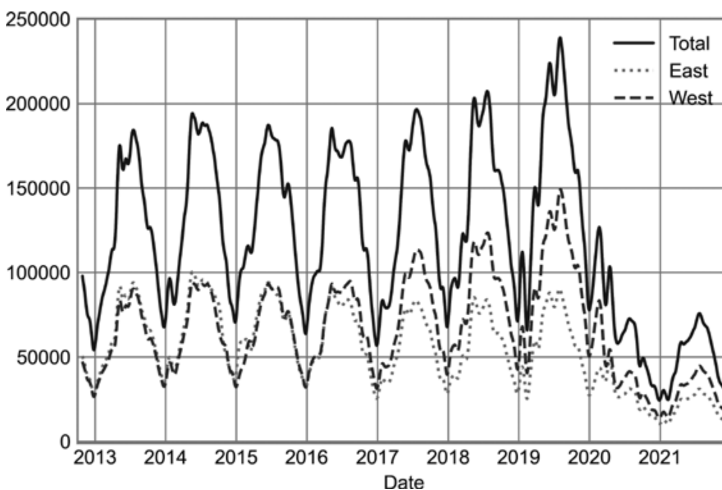


Рис. 23.9. Сглаженная гауссова функция еженедельного количества велосипедов

Углубленное изучение данных

Хотя с помощью сглаженных представлений данных можно получить общее представление о тенденциях, они скрывают от нас многие интересные нюансы их структуры. Например, нам может понадобиться увидеть усредненное движение велосипедного транспорта как функцию от времени суток. Это можно сделать с помощью функциональности `groupby`, обсуждавшейся в главе 20 (рис. 23.10):

```
In [40]: by_time = data.groupby(data.index.time).mean()
         hourly_ticks = 4 * 60 * 60 * np.arange(6)
         by_time.plot(xticks=hourly_ticks, style=['-', ':', '--']);
```

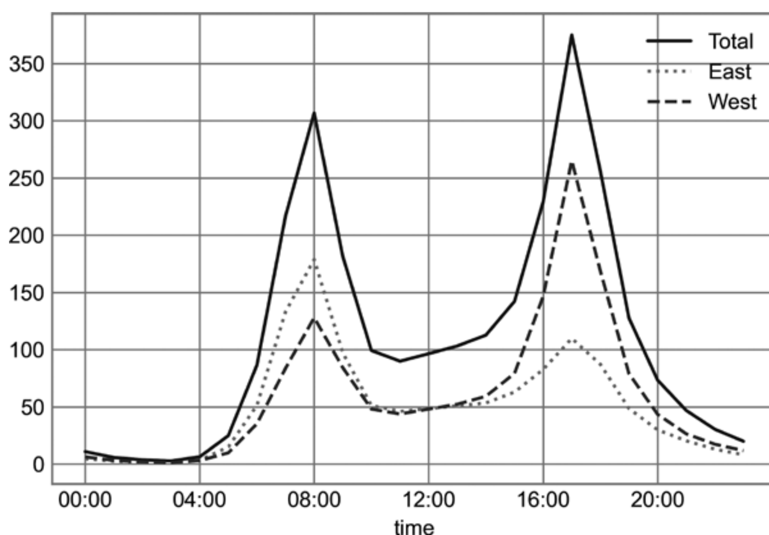


Рис. 23.10. Среднее почасовое количество велосипедов

График почасового трафика имеет двугорбый вид с максимумами в 08:00 утра и 05:00 вечера. Вероятно, это свидетельствует о существенном вкладе маятниковой миграции[1] через мост. В пользу этого говорят и различия между значениями с западной боковой дорожки (обычно используемой при движении в деловой центр Сиэтла) с более выраженными утренними максимумами и значениями с восточной боковой дорожки (обычно используемой при движении из делового центра Сиэтла) с более выраженными вечерними максимумами.

[1] См.: https://ru.wikipedia.org/wiki/Маятниковая_миграция.

Нас могут также интересовать изменения по дням недели. Эти изменения тоже можно выявить с помощью операции groupby (рис. 23.11):

```
In [41]: by_weekday = data.groupby(data.index.dayofweek).mean()
         by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']
         by_weekday.plot(style=['-', ':', '--']);
```

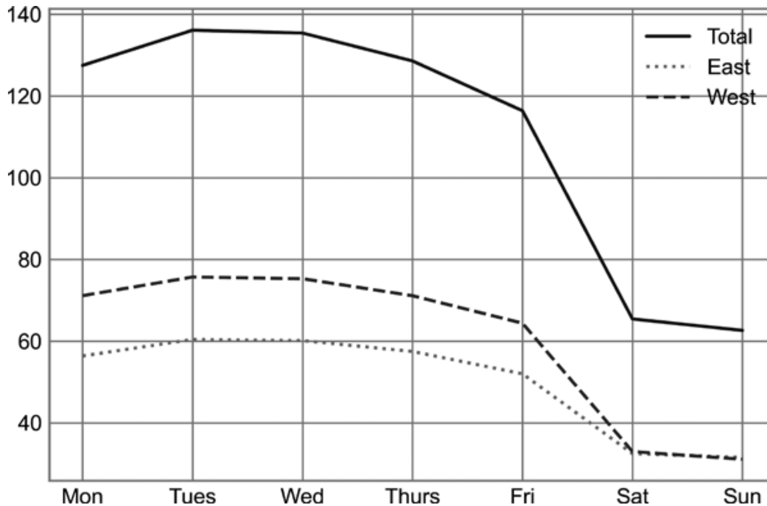


Рис. 23.11. Среднее количество велосипедов по дням недели

Этот график демонстрирует существенное различие между количеством велосипедов в будние и выходные дни: с понедельника по пятницу мост пересекает в среднем вдвое больше велосипедистов, чем в субботу и воскресенье.

С учетом этого выполним составную операцию groupby и посмотрим на почасовой тренд в будни по сравнению с выходными. Начнем с группировки как по признаку выходного дня, так и по времени суток:

```
In [42]: weekend = np.where(data.index.weekday < 5, 'Weekday', 'Weekend')
         by_time = data.groupby([weekend, data.index.time]).mean()
```

Теперь воспользуемся некоторыми инструментами, которые описываются в главе 31, чтобы нарисовать два графика бок о бок (рис. 23.12):

```
In [43]: import matplotlib.pyplot as plt
         fig, ax = plt.subplots(1, 2, figsize=(14, 5))
         by_time.loc['Weekday'].plot(ax=ax[0], title='Weekdays',
                                xticks=hourly_ticks, style=['-', ':', '--'])
         by_time.loc['Weekend'].plot(ax=ax[1], title='Weekends',
                                xticks=hourly_ticks, style=['-', ':', '--']);
```

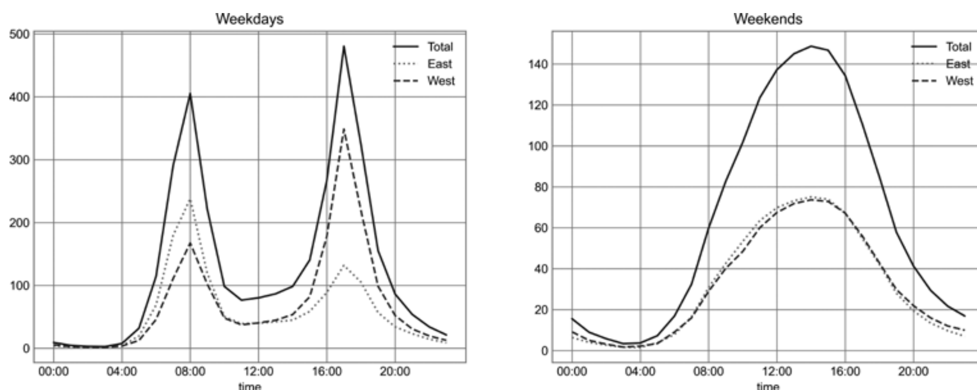


Рис. 23.12. Среднее количество велосипедов по часам в рабочие и выходные дни

Результат оказался очень интересным: мы видим двугорбый паттерн в рабочие дни, обусловленный поездками на работу в город, и одногорбый, обусловленный досугом/отдыхом в выходные. Было бы интересно дальше покопаться в этих данных и изучить влияние погоды, температуры, времени года и других факторов на паттерны поездок в город на велосипедах. Дальнейшее обсуждение этих вопросов вы найдете в моем блоге, в статье «Действительно ли в Сиэтле наблюдается оживление поездок на велосипедах?» (https://oreil.ly/j5oEI), иллюстрирующей анализ подмножества этих данных. Мы еще раз вернемся к этому набору данных в контексте моделирования в главе 42.

ГЛАВА 24

Увеличение производительности библиотеки Pandas: eval() и query()

Как мы видели в предыдущих главах, основные возможности стека PyData основываются на возможности библиотек NumPy и Pandas передавать выполнение простых операций низкоуровневому машинному коду посредством интуитивно понятного высокоуровневого синтаксиса: примерами могут послужить векторизованные/транслируемые операции в библиотеке NumPy, а также операции группировки в библиотеке Pandas. Хотя эти абстракции весьма производительны и эффективно работают для многих распространенных сценариев использования, они зачастую требуют создания временных вспомогательных объектов, что приводит к чрезмерным накладным расходам как процессорного времени, так и оперативной памяти.

Библиотека Pandas включает некоторые методы, позволяющие обращаться к операциям, работающим со скоростью, присущей коду, реализованному на языке C, без выделения существенных объемов памяти на промежуточные массивы. Эти утилиты — функции `eval` и `query`, основанные на пакете Numexpr (https://oreil.ly/acvj5). В этой главе мы рассмотрим их использование и приведем некоторые эмпирические правила, позволяющие решить, имеет ли смысл их применять.

Основания для использования функций query() и eval(): составные выражения

Библиотеки NumPy и Pandas поддерживают быстрые векторизованные операции; например, при сложении элементов двух массивов:

```
In [1]: import numpy as np
        rng = np.random.default_rng(42)
        x = rng.random(1000000)
        y = rng.random(1000000)
        %timeit x + y
Out[1]: 2.21 ms ± 142 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Как обсуждалось в главе 6, такая операция выполняется гораздо быстрее, чем сложение с помощью простого цикла или генератора списков:

```
In [2]: %timeit np.fromiter((xi + yi for xi, yi in zip(x, y)),
                            dtype=x.dtype, count=len(x))
Out[2]: 263 ms ± 43.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Однако данная абстракция оказывается менее эффективной при вычислении составных выражений. Например, рассмотрим выражение:

```
In [3]: mask = (x > 0.5) & (y < 0.5)
```

Поскольку библиотека NumPy вычисляет каждое подвыражение отдельно, оно эквивалентно следующему:

```
In [4]: tmp1 = (x > 0.5)
        tmp2 = (y < 0.5)
        mask = tmp1 & tmp2
```

Другими словами, *для каждого промежуточного шага явным образом выделяется оперативная память*. Если массивы `x` и `y` очень велики, это может привести к значительным накладным расходам оперативной памяти и процессорного времени. Библиотека Numexpr позволяет вычислять подобные составные выражения поэлементно, не требуя выделения памяти под промежуточные массивы целиком. В документации библиотеки Numexpr (https://oreil.ly/acvj5) приведено больше подробностей, но пока достаточно сказать, что функции этой библиотеки принимают строку, содержащую выражение в стиле библиотеки NumPy, которое требуется вычислить:

```
In [5]: import numexpr
        mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
        np.all(mask == mask_numexpr)
Out[5]: True
```

Преимущество заключается в том, что библиотека Numexpr вычисляет выражение, не используя полноразмерных временных массивов, а потому оказывается намного более эффективной, чем NumPy, особенно при работе с большими массивами. Инструменты `query` и `eval` из библиотеки Pandas, которые мы будем обсуждать, идеологически схожи и используют пакет Numexpr.

Использование функции pandas.eval() для эффективных операций

Функция `eval` библиотеки Pandas применяет строковые выражения для эффективных вычислительных операций с объектами `DataFrame`. Например, рассмотрим следующие объекты `DataFrame`:

```
In [6]: import pandas as pd
        nrows, ncols = 100000, 100
        df1, df2, df3, df4 = (pd.DataFrame(rng.random((nrows, ncols)))
                              for i in range(4))
```

Вычисление суммы всех четырех объектов `DataFrame` при стандартном подходе можно реализовать так:

```
In [7]: %timeit df1 + df2 + df3 + df4
Out[7]: 73.2 ms ± 6.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Тот же результат можно получить с помощью функции `pd.eval`, задав выражение в виде строки:

```
In [8]: %timeit pd.eval('df1 + df2 + df3 + df4')
Out[8]: 34 ms ± 4.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Версия этого выражения с функцией `eval` работает на 50 % быстрее (и использует намного меньше памяти), возвращая тот же результат:

```
In [9]: np.allclose(df1 + df2 + df3 + df4,
                    pd.eval('df1 + df2 + df3 + df4'))
Out[9]: True
```

Функция `pd.eval` поддерживает широкий спектр операций. Для их демонстрации мы используем следующие целочисленные данные:

```
In [10]: df1, df2, df3, df4, df5 = (pd.DataFrame(rng.integers(0, 1000, (100, 3)))
                                    for i in range(5))
```

Ниже приводится краткая сводка операций, поддерживаемых `pd.eval`.

Арифметические операторы

Функция `pd.eval` поддерживает все арифметические операторы. Например:

```
In [11]: result1 = -df1 * df2 / (df3 + df4) - df5
         result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
         np.allclose(result1, result2)
Out[11]: True
```

Операторы сравнения

Функция `pd.eval` поддерживает все операторы сравнения, включая выражения, организованные цепочкой:

```
In [12]: result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
         result2 = pd.eval('df1 < df2 <= df3 != df4')
         np.allclose(result1, result2)
Out[12]: True
```

Побитовые операторы

Функция `pd.eval` поддерживает побитовые операторы `&` и `|`:

```
In [13]: result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
         result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
         np.allclose(result1, result2)
Out[13]: True
```

Кроме того, она допускает использование литералов `and` и `or` в булевых выражениях:

```
In [14]: result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
         np.allclose(result1, result3)
Out[14]: True
```

Атрибуты объектов и индексы

Функция `pd.eval` поддерживает доступ к атрибутам объектов с помощью синтаксиса `obj.attr` и к индексам посредством синтаксиса `obj[index]`:

```
In [15]: result1 = df2.T[0] + df3.iloc[1]
         result2 = pd.eval('df2.T[0] + df3.iloc[1]')
         np.allclose(result1, result2)
Out[15]: True
```

Другие операции

Другие операции, например вызовы функций, условные выражения, циклы и другие, более сложные конструкции, пока не реализованы в функции `pd.eval`. При необходимости выполнения подобных сложных видов выражений можно воспользоваться самой библиотекой Numexpr.

Использование метода DataFrame.eval() для выполнения операций по столбцам

У объектов `DataFrame` существует метод `eval`, работающий схожим образом с высокоуровневой функцией `pd.eval` из библиотеки Pandas. Преимущество метода `eval` заключается в возможности ссылаться на столбцы по имени. Возьмем для примера следующий маркированный массив:

```
In [16]: df = pd.DataFrame(rng.random((1000, 3)), columns=['A', 'B', 'C'])
         df.head()
Out[16]:          A         B         C
         0  0.850888  0.966709  0.958690
```

```
        1  0.820126  0.385686  0.061402
        2  0.059729  0.831768  0.652259
        3  0.244774  0.140322  0.041711
        4  0.818205  0.753384  0.578851
```

Воспользовавшись функцией `pd.eval` так, как показано выше, можно вычислять выражения с этими тремя столбцами:

```
In [17]: result1 = (df['A'] + df['B']) / (df['C'] - 1)
         result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
         np.allclose(result1, result2)
Out[17]: True
```

Метод `DataFrame.eval` позволяет описывать вычисления со столбцами гораздо лаконичнее:

```
In [18]: result3 = df.eval('(A + B) / (C - 1)')
         np.allclose(result1, result3)
Out[18]: True
```

Обратите внимание, что мы обращаемся с *именами столбцов* в вычисляемом выражении *как с переменными* и получаем желаемый результат.

Присваивание в методе DataFrame.eval()

Метод `DataFrame.eval` позволяет присвоить значение любому из столбцов. Воспользуемся предыдущим объектом `DataFrame` со столбцами `'A'`, `'B'` и `'C'`:

```
In [19]: df.head()
Out[19]:           A         B         C
        0  0.850888  0.966709  0.958690
        1  0.820126  0.385686  0.061402
        2  0.059729  0.831768  0.652259
        3  0.244774  0.140322  0.041711
        4  0.818205  0.753384  0.578851
```

Вызовом `df.eval` можно создать новый столбец `'D'` и присваивать ему значения, вычисленные на основе других столбцов:

```
In [20]: df.eval('D = (A + B) / C', inplace=True)
         df.head()
Out[20]:           A         B         C          D
        0  0.850888  0.966709  0.958690   1.895916
        1  0.820126  0.385686  0.061402  19.638139
        2  0.059729  0.831768  0.652259   1.366782
        3  0.244774  0.140322  0.041711   9.232370
        4  0.818205  0.753384  0.578851   2.715013
```

Аналогично можно модифицировать значения любого существующего столбца:

```
In [21]: df.eval('D = (A - B) / C', inplace=True)
         df.head()
Out[21]:          A         B         C         D
         0  0.850888  0.966709  0.958690 -0.120812
         1  0.820126  0.385686  0.061402  7.075399
         2  0.059729  0.831768  0.652259 -1.183638
         3  0.244774  0.140322  0.041711  2.504142
         4  0.818205  0.753384  0.578851  0.111982
```

Локальные переменные в методе DataFrame.eval()

Метод `DataFrame.eval` поддерживает дополнительный синтаксис для работы с локальными переменными. Взгляните на следующий фрагмент кода:

```
In [22]: column_mean = df.mean(1)
         result1 = df['A'] + column_mean
         result2 = df.eval('A + @column_mean')
         np.allclose(result1, result2)
Out[22]: True
```

Символ `@` отличает *имя переменной* от *имени столбца*, позволяя эффективно вычислять значения выражений с использованием двух «пространств имен»: пространства имен столбцов и пространства имен объектов Python. Обратите внимание, что символ `@` поддерживается лишь *методом* `DataFrame.eval`, но не поддерживается функцией `pandas.eval`, потому что `pandas.eval` имеет доступ только к одному пространству имен (языка Python).

Метод DataFrame.query()

Объекты DataFrame имеют еще один метод, основанный на вычислении выражений в строковом формате, — метод `query`. Рассмотрим следующий код:

```
In [23]: result1 = df[(df.A < 0.5) & (df.B < 0.5)]
         result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
         np.allclose(result1, result2)
Out[23]: True
```

Как и в примере, который мы использовали при обсуждении метода `DataFrame.eval`, это выражение манипулирует столбцами объекта `DataFrame`. Однако его нельзя выразить с помощью синтаксиса метода `DataFrame.eval`! Зато для подобных операций фильтрации можно воспользоваться методом `query`:

```
In [24]: result2 = df.query('A < 0.5 and B < 0.5')
         np.allclose(result1, result2)
Out[24]: True
```

Он не только обеспечивает более высокую эффективность вычислений по сравнению с выражениями маскирования, но и читается намного проще. Обратите внимание, что метод `query` также позволяет использовать флаг `@` для обозначения локальных переменных:

```
In [25]: Cmean = df['C'].mean()
         result1 = df[(df.A < Cmean) & (df.B < Cmean)]
         result2 = df.query('A < @Cmean and B < @Cmean')
         np.allclose(result1, result2)
Out[25]: True
```

Производительность: когда следует использовать эти функции

В процессе принятия решения, применять ли `eval` и `query`, обратите внимание на два момента: *процессорное время* и *объем используемой памяти*. Предсказать объем используемой памяти намного проще. Как уже упоминалось, все составные выражения, вовлекающие массивы NumPy или объекты `DataFrame` библиотеки Pandas, приводят к неявному созданию временных массивов. Например, этот код:

```
In [26]: x = df[(df.A < 0.5) & (df.B < 0.5)]
```

приблизительно соответствует следующему:

```
In [27]: tmp1 = df.A < 0.5
         tmp2 = df.B < 0.5
         tmp3 = tmp1 & tmp2
         x = df[tmp3]
```

Если размер временных объектов `DataFrame` существен по сравнению с объемом оперативной памяти, доступной в системе (обычно несколько гигабайт), то будет разумно воспользоваться выражениями `eval` или `query`. Вот как можно выяснить приблизительный размер массива в байтах:

```
In [28]: df.values.nbytes
Out[28]: 32000
```

`eval` будет работать быстрее, если использовать не всю оперативную память, доступную в системе. Основную роль играет отношение размера временных объектов `DataFrame` к размеру процессорных кешей L1 и L2 (обычно несколько мегабайт); если они имеют достаточную емкость, то `eval` может избежать потенциально медленного перемещения значений между различными кешами. Я обнаружил, что на практике различие в скорости вычислений между традиционными методами и методом `eval`/`query` обычно довольно незначительно. Напротив, традиционный метод работает быстрее для маленьких массивов! Преимущество

метода `eval`/`query` заключается в экономии оперативной памяти и иногда — в более понятном синтаксисе.

Мы рассмотрели большинство нюансов работы с методами `eval` и `query`, дополнительную информацию можно найти в документации библиотеки Pandas. В частности, для обработки этих запросов можно задавать различные синтаксические анализаторы и механизмы. Подробности ищите в разделе Enhancing Performance («Повышение производительности»; https://oreil.ly/DHNy8).

Дополнительные источники информации

В этой части мы рассмотрели множество примеров эффективного использования библиотеки Pandas для анализа данных. За более полной информацией о библиотеке Pandas я рекомендую обратиться к следующим источникам информации.

- *Онлайн-документация библиотеки Pandas* (http://pandas.pydata.org/). Это основной источник информации о данном пакете. Конечно, примеры в документации обычно основаны на небольших, искусственно сгенерированных наборах данных, но параметры описываются во всей полноте, что очень помогает понять, как использовать различные функции.
- *Python for Data Analysis*[1] (https://oreil.ly/0hdsf). В книге, написанной Уэсом Маккини, подробно рассматриваются инструменты для работы с временными рядами, обеспечившие его как финансового аналитика средствами к существованию. В книге также приведено множество интересных примеров применения Python для получения полезной информации из реальных наборов данных.
- *Effective Pandas* (https://oreil.ly/cn1ls). В этой небольшой электронной книге Том Аугспургер, один из разработчиков библиотеки Pandas, рассказывает об эффективных и идиоматических способах использования всей мощи библиотеки Pandas.
- *Библиотека Pandas на сайте PyVideo* (https://oreil.ly/mh4wI). На многих конференциях, таких как PyCon, SciPy и PyData, выпускались руководства по библиотеке Pandas, написанные разработчиками и опытными пользователями. Руководства, в частности представленные на PyCon, написаны наиболее опытными профессионалами.

Надеюсь, что с этими источниками информации в сочетании с демонстрацией возможностей в данной главе вы сможете справиться с любой задачей по анализу данных с применением библиотеки Pandas!

---

[1] *Маккини У.* Python и анализ данных.

ЧАСТЬ IV

Визуализация с помощью библиотеки Matplotlib

Теперь мы подробнее рассмотрим пакет Matplotlib, предназначенный для построения графиков и диаграмм. Matplotlib — мультиплатформенная библиотека для визуализации данных, основанная на массивах библиотеки NumPy и спроектированная в расчете на работу с обширным стеком SciPy. Она была задумана Джоном Хантером в 2002 году и изначально была реализована как дополнение к оболочке IPython для интерактивного построения графиков в стиле MATLAB из командной строки IPython с помощью утилиты `gnuplot`. Фернандо Перес, создатель оболочки IPython, в этот момент был занят работой над своей диссертацией, поэтому он сообщил Джону, что в ближайшие несколько месяцев у него не будет времени на анализ этого дополнения. Хантер воспринял это как благословение на самостоятельную разработку — так родился пакет Matplotlib, версия 0.1 которого была выпущена в 2003 году. Институт исследований космоса с помощью космического телескопа (Space Telescope Science Institute, занимающийся управлением телескопом «Хаббл») финансово поддержал разработку пакета Matplotlib и обеспечил расширение его возможностей, избрав в качестве пакета для формирования графических изображений.

Одна из важнейших особенностей пакета Matplotlib — хорошая совместимость со множеством операционных систем и графических движков. Matplotlib поддерживает десятки движков и выходных форматов, а значит, вы можете положиться на него независимо от используемой операционной системы или требуемого формата вывода. Самая сильная сторона пакета Matplotlib — кросс-платформенный подход типа «все для всех», способствовавший росту числа пользователей, что, в свою очередь, стало причиной появления большого числа активных разработчиков, расширения возможностей пакета Matplotlib и его быстрого распространения в мире научных вычислений на языке Python.

В последние годы интерфейс и стиль библиотеки Matplotlib начали несколько устаревать. На фоне новых утилит, таких как ggplot и ggvis в языке R, а также веб-инструментов визуализации, основанных на холстах D3js и HTML5, она выглядит неуклюжей и старомодной. Тем не менее я полагаю, что нельзя игнорировать возможности библиотеки Matplotlib — надежного кросс-платформенного механизма визуализации. Свежие версии Matplotlib упрощают настройку новых глобальных стилей вывода графики (см. главу 34). Разрабатываются новые пакеты, предназначенные для работы с ней через более современные API, такие как Seaborn (см. главу 36), ggpy (http://yhat.github.io/ggpy), HoloViews (http://holoviews.org/), и даже саму библиотеку Pandas можно использовать как обертку вокруг Matplotlib API. Однако даже при наличии подобных оберток полезно знать и понимать синтаксис Matplotlib для настройки вывода итогового графика. Поэтому я считаю, что сама библиотека Matplotlib остается жизненно важной частью стека визуализации данных, даже при том, что появление новых инструментов постепенно уводит сообщество пользователей от непосредственного использования Matplotlib API.

ГЛАВА 25

Общие советы по библиотеке Matplotlib

Прежде чем погрузиться в подробности создания графиков и диаграмм с помощью Matplotlib, я хочу рассказать несколько полезных сведений про этот пакет.

Импортирование matplotlib

По аналогии с библиотеками NumPy и Pandas, которые мы импортировали под сокращенными псевдонимами `np` и `pd` соответственно, импортируем библиотеку Matplotlib и ее модули, использовав общепринятые псевдонимы:

```
In [1]: import matplotlib as mpl
        import matplotlib.pyplot as plt
```

Чаще всего мы будем использовать интерфейс `plt`.

Настройка стилей

Выбирать подходящие стили для наших графиков мы будем с помощью директивы `plt.style`. Вот как можно выбрать стиль `classic`, обеспечивающий создание графиков с классическим оформлением:

```
In [2]: plt.style.use('classic')
```

Этот стиль будет использоваться на протяжении всей главы везде, где это уместно. Дополнительную информацию о стилях вы найдете в главе 34.

Использовать или не использовать show()? Как отображать графики

Визуализация, которую не видно, не несет никакой пользы, но вид графиков, выводимых библиотекой Matplotlib, зависит от контекста, в котором она используется:

- в сценарии на Python;
- в сеансе оболочки IPython в терминале;
- в блокноте IPython.

Построение графиков в сценариях

При использовании библиотеки Matplotlib в сценарии вам очень пригодится функция `plt.show`. Она запускает цикл обработки событий, ищет все активные в настоящий момент объекты `Figure` и открывает одно или несколько интерактивных окон для отображения вашего графика (графиков).

Допустим, у вас имеется файл `myplot.py` со следующим кодом:

```
# файл: myplot.py
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

Этот сценарий можно запустить из командной строки, что приведет к открытию окна с графиком:

```
$ python myplot.py
```

Команда `plt.show` выполняет множество операций в процессе взаимодействия с интерактивным графическим движком системы. Детали этих действий различаются для разных операционных систем и конкретных версий, но библиотека Matplotlib делает все, чтобы скрыть эти детали от вас.

Одно важное замечание: команду `plt.show` следует использовать *только один раз* в течение сеанса работы с Python, и чаще всего ее можно встретить в самом конце сценария. Выполнение нескольких команд `show` может привести к непредсказуемому поведению в зависимости от особенностей графического движка, поэтому старайтесь избегать этого.

Построение графиков из командной оболочки IPython

Matplotlib также с успехом можно использовать в интерактивной оболочке IPython (см. часть I). Оболочка IPython прекрасно работает с библиотекой Matplotlib, если перевести ее в режим Matplotlib, для чего достаточно после запуска IPython выполнить «магическую» команду `%matplotlib`:

```
In [1]: %matplotlib
Using matplotlib backend: TkAgg

In [2]: import matplotlib.pyplot as plt
```

После этого любая команда `plot` будет открывать окно графика с возможностью его изменения вызовом дальнейших команд. Некоторые изменения (например, модификация свойств уже нарисованных линий) не будут применяться автоматически, поэтому потребуется выполнить команду `plt.draw`. Выполнять команду `plt.show` в оболочке IPython в режиме Matplotlib не обязательно.

Построение графиков из блокнота Jupyter

Блокнот Jupyter — браузерный интерактивный инструмент для анализа данных, позволяющий совмещать текстовое описание, код, графики, элементы HTML и многое другое в едином выполняемом документе (см. часть I).

Интерактивное построение графиков в блокноте Jupyter возможно с помощью команды `%matplotlib` и осуществляется так же, как в командной оболочке IPython. В Jupyter также есть две альтернативные возможности включения графиков непосредственно в блокнот:

- команда `%matplotlib inline` включит в блокнот *статические* изображения графиков;
- команда `%matplotlib notebook` включит в блокнот *интерактивные* графики.

В книге мы будем придерживаться режима по умолчанию — отображения статических графиков (на рис. 25.1 показан результат выполнения примера построения простого графика):

```
In [3]: %matplotlib inline
In [4]: import numpy as np
        x = np.linspace(0, 10, 100)

        fig = plt.figure()
        plt.plot(x, np.sin(x), '-')
        plt.plot(x, np.cos(x), '--');
```

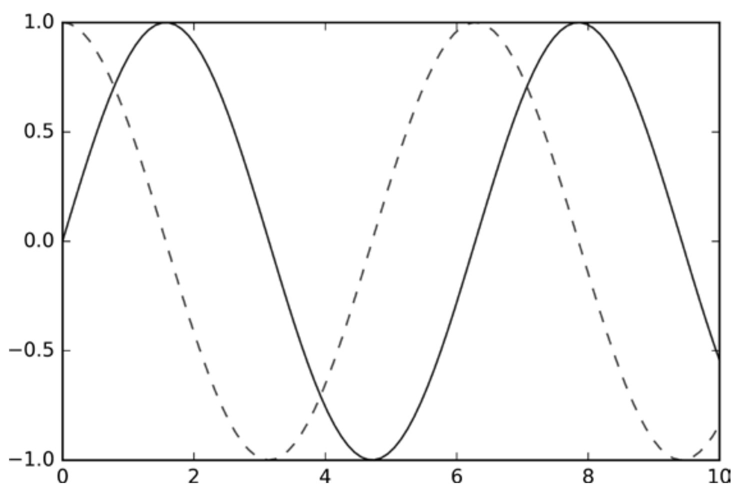


Рис. 25.1. Пример построения простого графика

Сохранение изображений в файлы

Умение сохранять рисунки в файлы различных форматов — одна из замечательных возможностей библиотеки Matplotlib. Сохранение выполняется с помощью команды `savefig`. Например, вот как можно сохранить предыдущее изображение в файл PNG:

```
In [5]: fig.savefig('my_figure.png')
```

Эта команда создаст в текущем рабочем каталоге файл с именем `my_figure.png`:

```
In [6]: !ls -lh my_figure.png
Out[6]: -rw-r--r--  1 jakevdp  staff    26K Feb  1 06:15 my_figure.png
```

Чтобы убедиться, что содержимое файла соответствует нашим ожиданиям, воспользуемся объектом `Image` оболочки IPython для отображения его содержимого (рис. 25.2):

```
In [7]: from IPython.display import Image
        Image('my_figure.png')
```

Команда `savefig` определяет формат файла, исходя из расширения имени файла. В зависимости от установленного графического движка в вашей системе может поддерживаться множество различных форматов файлов. Вывести список поддерживаемых форматов файлов можно с помощью следующего метода объекта `canvas` рисунка:

```
In [8]: fig.canvas.get_supported_filetypes()
Out[8]: {'eps': 'Encapsulated Postscript',
         'jpg': 'Joint Photographic Experts Group',
         'jpeg': 'Joint Photographic Experts Group',
         'pdf': 'Portable Document Format',
         'pgf': 'PGF code for LaTeX',
         'png': 'Portable Network Graphics',
         'ps': 'Postscript',
         'raw': 'Raw RGBA bitmap',
         'rgba': 'Raw RGBA bitmap',
         'svg': 'Scalable Vector Graphics',
         'svgz': 'Scalable Vector Graphics',
         'tif': 'Tagged Image File Format',
         'tiff': 'Tagged Image File Format'}
```

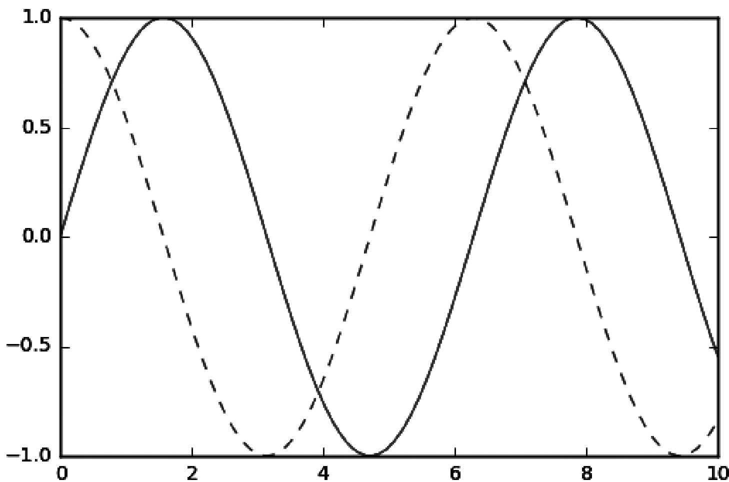


Рис. 25.2. Файл PNG с простым графиком

Обратите внимание, что при сохранении рисунка не обязательно использовать `plt.show` или другие команды, обсуждавшиеся выше.

Два интерфейса по цене одного

Наличие в библиотеке Matplotlib двух интерфейсов (MATLAB-подобного, основанного на сохранении состояния, и объектно-ориентированного, обладающего большими возможностями) потенциально может привести к путанице. Рассмотрим вкратце различия между ними.

Интерфейс в стиле MATLAB

Библиотека Matplotlib изначально была написана как альтернативный вариант (на языке Python) для пользователей пакета MATLAB, и значительная часть ее синтаксиса отражает этот факт. MATLAB-подобные инструменты содержатся в интерфейсе `pyplot` (`plt`). Например, следующий код наверняка покажется очень знакомым пользователям MATLAB (рис. 25.3):

```
In [9]: plt.figure()  # Создание графика

        # Создаем первую из двух областей графика и задаем текущую ось
        plt.subplot(2, 1, 1) # (rows, columns, panel number)
        plt.plot(x, np.sin(x))

        # Создаем вторую область и задаем текущую ось
        plt.subplot(2, 1, 2)
        plt.plot(x, np.cos(x));
```

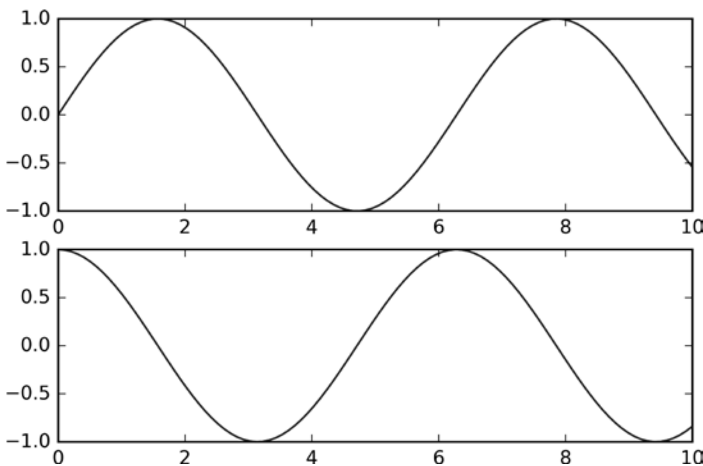


Рис. 25.3. Два графика, созданных с помощью MATLAB-подобного интерфейса

Важно отметить, что этот интерфейс *сохраняет состояние*: он запоминает «текущий» рисунок и его оси координат и применяет к ним все команды `plt`. Получить ссылки на состояние можно с помощью команд `plt.gcf` (от англ. get current figure — «получить текущий рисунок») и `plt.gca` (от англ. get current axes — «получить текущие оси координат»).

Этот интерфейс с состоянием достаточно удобен при работе с простыми графиками, но легко может привести к проблемам. Например, как после создания второго графика вернуться к первому и добавить что-либо в него? В MATLAB-подобном интерфейсе есть способ для этого, но он довольно громоздкий. Существует лучший вариант.

Объектно-ориентированный интерфейс

Объектно-ориентированный интерфейс подходит для более сложных ситуаций, когда вам требуется больше возможностей управления графиком. В объектно-ориентированном интерфейсе функции рисования не полагаются на понятие текущего рисунка или осей, а являются *методами* явно определяемых объектов `Figure` и `Axes`. Вот как можно перерисовать предыдущий рисунок с помощью этого интерфейса (рис. 25.4):

```
In [10]: # Сначала создаем сетку графиков
         # ax — это массив из двух объектов Axes
         fig, ax = plt.subplots(2)

         # Вызываем методы plot() соответствующих объектов
         ax[0].plot(x, np.sin(x))
         ax[1].plot(x, np.cos(x));
```

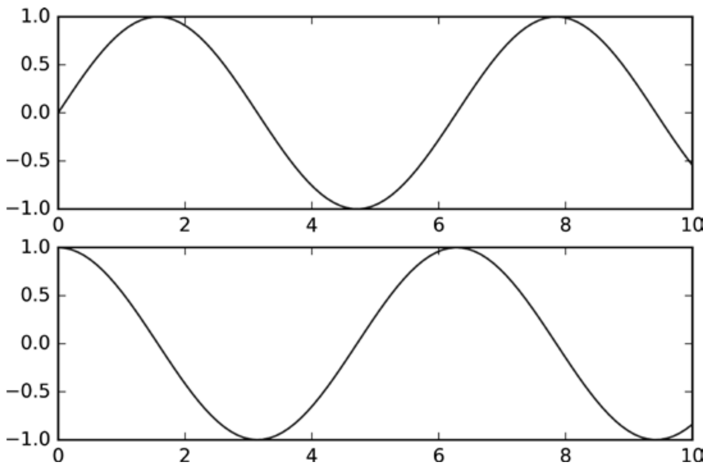


Рис. 25.4. Два графика, созданных с помощью объектно-ориентированного интерфейса

Выбор интерфейса при создании простых графиков в основном вопрос личных предпочтений, но по мере усложнения графиков объектно-ориентированный подход становится необходимостью. На протяжении следующих глав мы будем переключаться между MATLAB-подобным и объектно-ориентированным интерфейсами в зависимости от того, какой из них удобнее для конкретной задачи. В большинстве случаев в коде достаточно лишь заменить `plt.plot` на `ax.plot` и не более того, но есть несколько нюансов, на которые мы будем обращать внимание в следующих главах.

ГЛАВА 26

Простые линейные графики

Вероятно, простейшим из всех графиков является график отдельной функции $y = f(x)$. В этой главе мы рассмотрим создание простого графика такого типа. Как и во всех последующих главах, начнем с настройки блокнота для построения графиков и импорта необходимых функций:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Во всех графиках Matplotlib создание графика начинается с объявления рисунка и системы координат. В простейшем случае рисунок и систему координат можно создать так (рис. 26.1):

```
In [2]: fig = plt.figure()
        ax = plt.axes()
```

Рисунок в библиотеке Matplotlib (экземпляр класса `plt.Figure`) можно рассматривать как контейнер, содержащий все объекты, представляющие систему координат, графические изображения, текст и метки. *Система координат* (или оси координат; экземпляры класса `plt.Axes`) — это ограничивающий прямоугольник с делениями и метками, в который затем будут помещаться элементы графика. В этой части книги мы будем использовать имя переменной `fig` для представления экземпляра рисунка и `ax` для экземпляров системы координат или группы экземпляров систем координат.

После создания осей можно вызвать метод `ax.plot` и построить график на основе некоторых данных. Начнем с простой синусоиды (рис. 26.2):

```
In [3]: fig = plt.figure()
        ax = plt.axes()

        x = np.linspace(0, 10, 1000)
        ax.plot(x, np.sin(x));
```

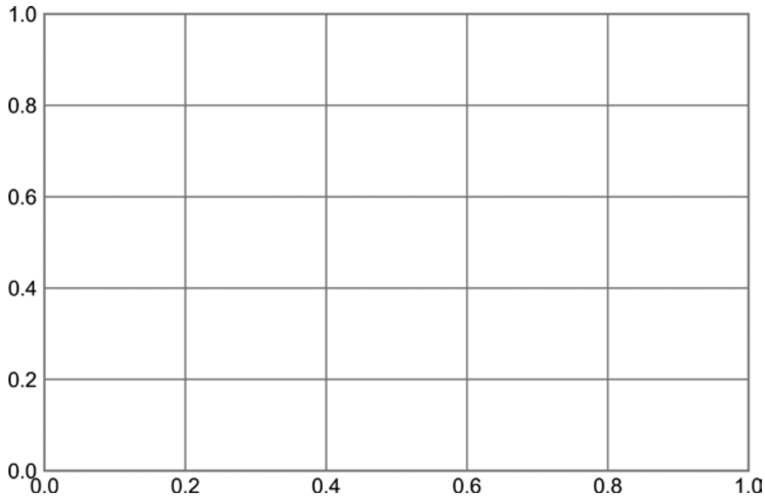


Рис. 26.1. Пустая система координат с координатными осями

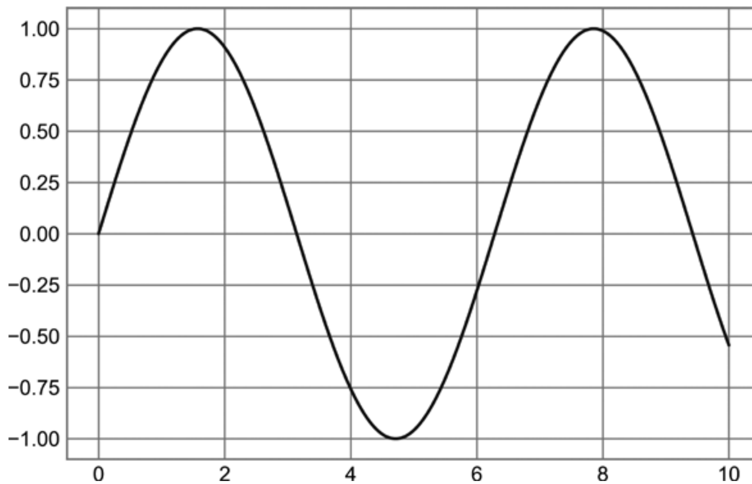


Рис. 26.2. Простая синусоида

Обратите внимание на точку с запятой в конце последней строки. Мы добавили ее не просто так. Она подавляет вывод графика в текстовом виде.

Мы могли бы воспользоваться и интерфейсом PyLab, при этом рисунок и система координат были бы созданы в фоновом режиме (см. обсуждение этих двух интерфейсов в части IV), как показано на рис. 26.3:

```
In [4]: plt.plot(x, np.sin(x));
```

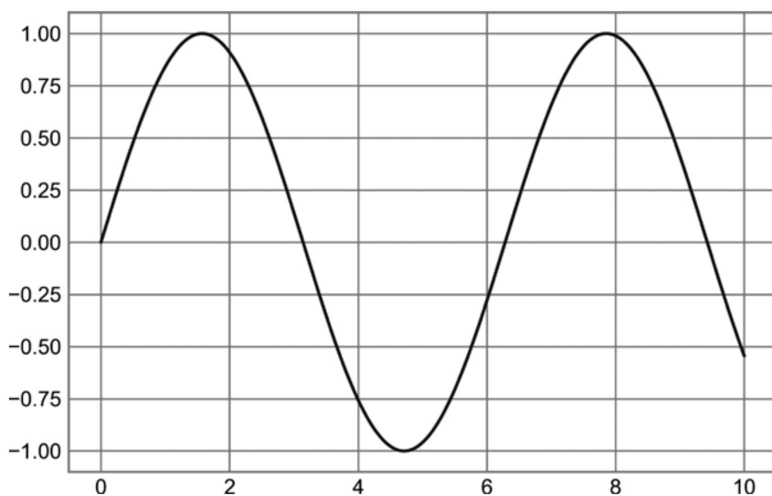


Рис. 26.3. Простая синусоида, построенная с помощью объектно-ориентированного интерфейса

Чтобы создать простой рисунок с несколькими линиями, можно вызвать функцию `plot` несколько раз (рис. 26.4):

```
In [5]: plt.plot(x, np.sin(x))
        plt.plot(x, np.cos(x));
```

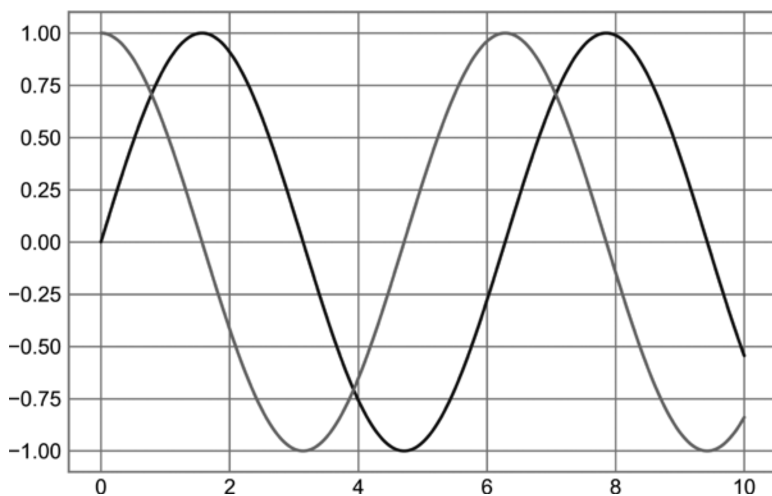


Рис. 26.4. Вывод нескольких линий на графике

Вот и все, что касается построения графиков простых функций с помощью библиотеки Matplotlib! Теперь углубимся в некоторые подробности управления внешним видом осей и линий.

Настройка графика: цвета и стили линий

Первое, что может понадобиться сделать с графиком, — научиться управлять цветами и стилями линий. Функция `plt.plot` принимает дополнительные аргументы, предназначенные для этой цели. Для настройки цвета используйте именованный аргумент `color`, в котором можно передать строковое значение, описывающее требуемый цвет. Задать цвет можно разными способами (рис. 26.5):

```
In[6]: plt.plot(x, np.sin(x - 0), color='blue')    # Название цвета
       plt.plot(x, np.sin(x - 1), color='g')       # Код цвета (rgbcmyk)
       plt.plot(x, np.sin(x - 2), color='0.75')    # Шкала оттенков серого,
                                                   # значение от 0 до 1
       plt.plot(x, np.sin(x - 3), color='#FFDD44') # 16-ричный код
                                                   # (RRGGBB от 00 до FF)
       plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # Кортеж значений RGB от 0 до 1
       plt.plot(x, np.sin(x - 5), color='chartreuse'); # Названия цветов
                                                       # в спецификации HTML
```

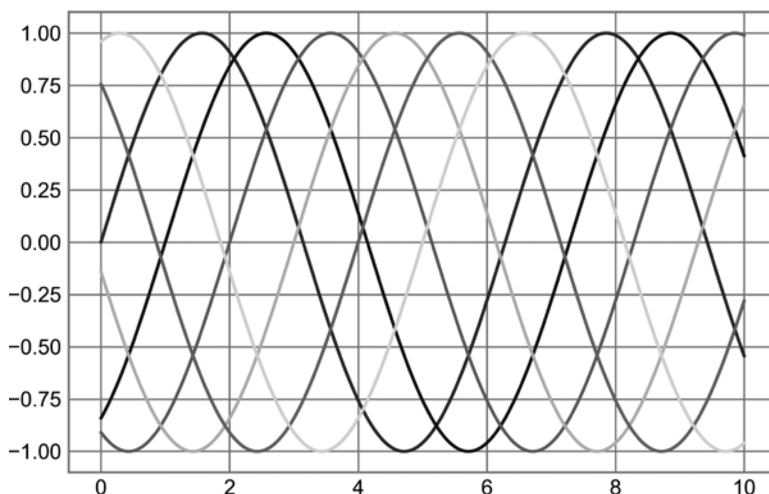


Рис. 26.5. Управление цветом элементов графика

Если цвет не задан явно, то библиотека Matplotlib будет циклически перебирать набор цветов по умолчанию при наличии на графике нескольких линий.

Стиль линий можно настраивать с помощью аргумента `linestyle` (рис. 26.6):

```
In[7]: plt.plot(x, x + 0, linestyle='solid')
       plt.plot(x, x + 1, linestyle='dashed')
       plt.plot(x, x + 2, linestyle='dashdot')
       plt.plot(x, x + 3, linestyle='dotted');

       # Также можно использовать следующие короткие коды:
       plt.plot(x, x + 4, linestyle='-')  # сплошная линия
       plt.plot(x, x + 5, linestyle='--') # штриховая линия
       plt.plot(x, x + 6, linestyle='-.') # штрихпунктирная линия
       plt.plot(x, x + 7, linestyle=':'); # пунктирная линия
```

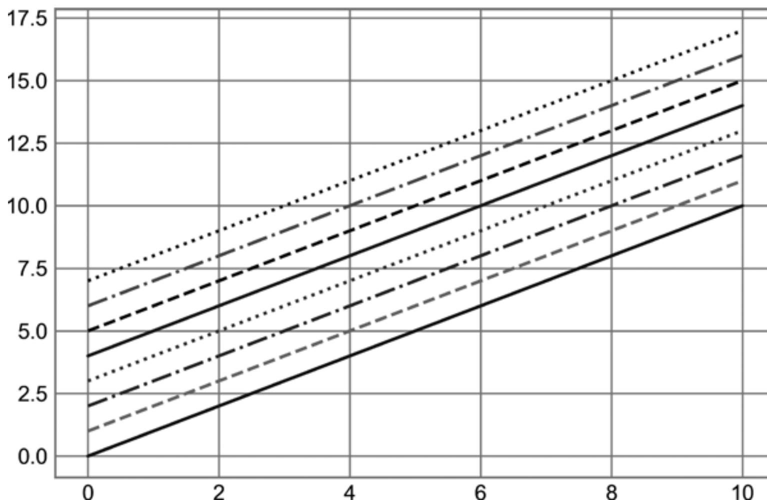


Рис. 26.6. Примеры различных стилей линий

Если вы предпочитаете максимально сжатый синтаксис, пусть и в ущерб удобочитаемости, то для вас предусмотрена возможность объединения кодов `linestyle` и `color` в одном неименованном аргументе функции `plt.plot` (рис. 26.7):

```
In[8]: plt.plot(x, x + 0, '-g')  # сплошная линия зеленого цвета
       plt.plot(x, x + 1, '--c') # штриховая линия голубого цвета
       plt.plot(x, x + 2, '-.k') # штрихпунктирная линия черного цвета
       plt.plot(x, x + 3, ':r'); # пунктирная линия красного цвета
```

Эти односимвольные коды цветов отражают стандартные сокращения, принятые в широко используемых в цифровой графике цветовых моделях RGB (Red/Green/Blue — «красный/зеленый/синий») и CMYK (Cyan/Magenta/Yellow/blacK — «голубой/пурпурный/желтый/черный»).

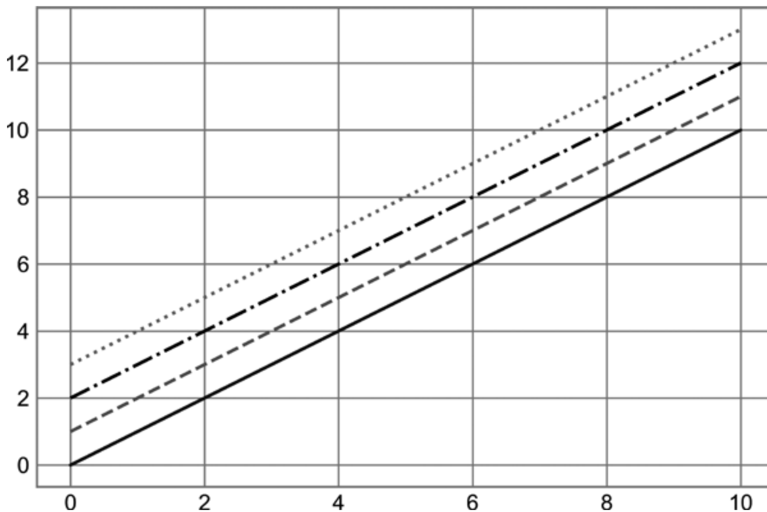


Рис. 26.7. Сокращенный синтаксис управления цветом и стилем

Существует множество других именованных аргументов, позволяющих выполнять тонкую настройку внешнего вида графика. Чтобы узнать больше, рекомендую заглянуть в описание функции `plt.plot` с помощью инструментов получения справки в оболочке IPython (см. главу 1).

Настройка графика: пределы осей координат

Библиотека Matplotlib достаточно хорошо подбирает пределы осей координат по умолчанию, но иногда требуется более точная настройка. Простейший способ настройки пределов осей координат — методы `plt.xlim` и `plt.ylim` (рис. 26.8):

```
In [9]: plt.plot(x, np.sin(x))

        plt.xlim(-1, 11)
        plt.ylim(-1.5, 1.5);
```

Если нужно, чтобы оси отображались в зеркальном отражении, передайте аргументы в обратном порядке (рис. 26.9):

```
In [10]: plt.plot(x, np.sin(x))

         plt.xlim(10, 0)
         plt.ylim(1.2, -1.2);
```

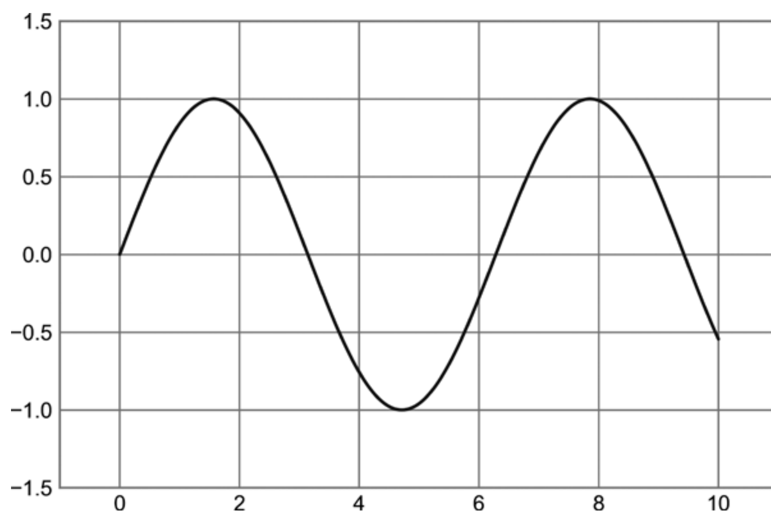


Рис. 26.8. Пример задания пределов осей координат

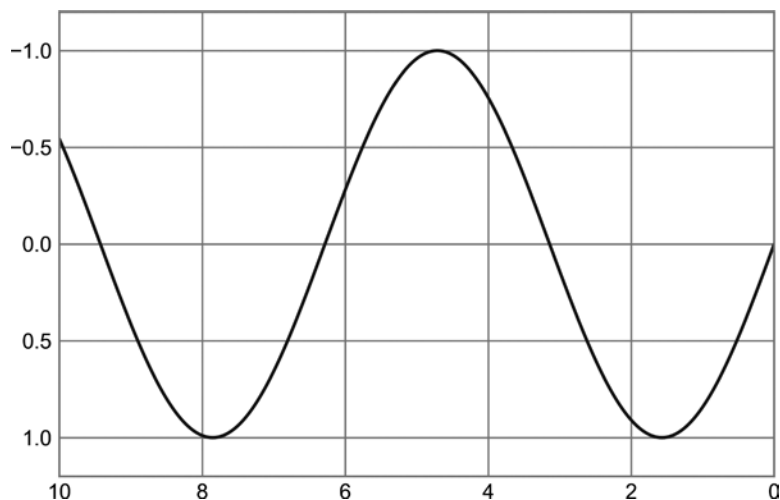


Рис. 26.9. Пример зеркального отображения оси Y

Для этого удобно использовать метод `plt.axis` (не перепутайте метод `plt.axis` с методом `plt.axes`!). Метод `plt.axis` позволяет более точно определить пределы осей. Например, можно автоматически прижать границы системы координат к текущему содержимому, как показано на рис. 26.10:

```
In [11]: plt.plot(x, np.sin(x))
         plt.axis('tight');
```

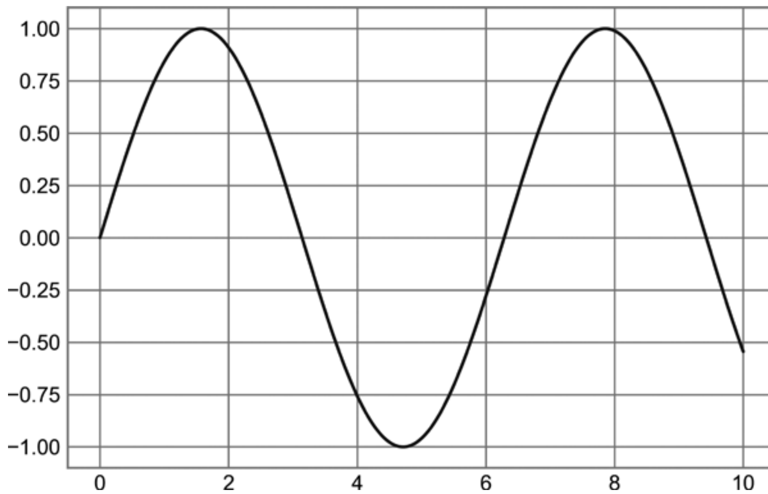


Рис. 26.10. Пример «компактного» графика

Также есть возможность задать отношение сторон графика так, чтобы на вашем экране длина равных приращений по осям X и Y выглядела одинаковой (рис. 26.11):

```
In [12]: plt.plot(x, np.sin(x))
         plt.axis('equal');
```

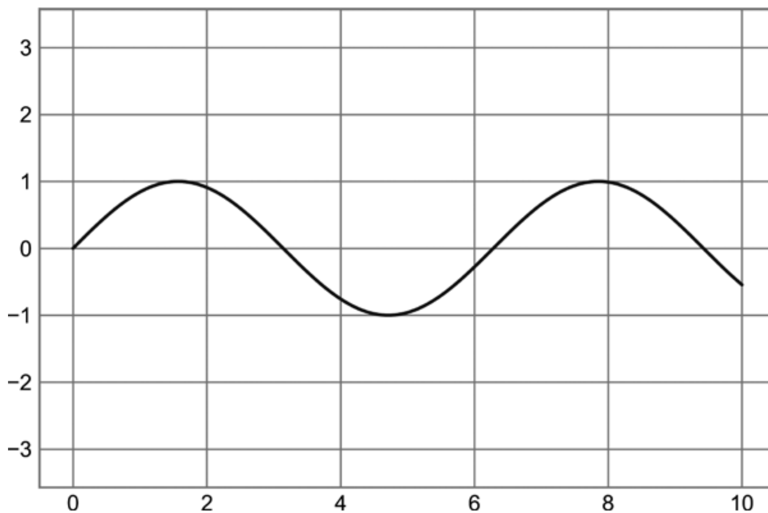


Рис. 26.11. Пример «равностороннего» графика,
в котором приращения по осям соответствуют выходному разрешению

Из других значений параметра метода `plt.axis` доступны: `'on'`, `'off'`, `'square'`, `'image'` и многие другие. Дополнительную информацию по пределам осей координат и другим возможностям метода `plt.axis` можно найти в документации с его описанием.

Метки на графиках

В завершение этой главы рассмотрим способы вывода надписей на графиках: названий, меток осей координат и простых легенд. Названия и метки осей — простейшие из подобных меток. Существуют методы, позволяющие быстро задать их (рис. 26.12):

```
In [13]: plt.plot(x, np.sin(x))
         plt.title("A Sine Curve") # Синусоидальная кривая
         plt.xlabel("x")
         plt.ylabel("sin(x)");
```

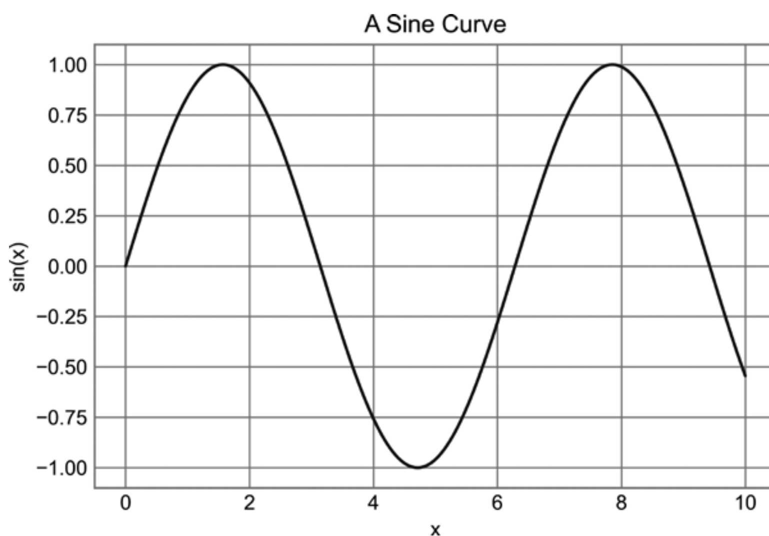


Рис. 26.12. Пример меток осей координат и названия графика

С помощью необязательных аргументов можно настраивать местоположение, размер и стиль меток, как описывается в документации.

В случае отображения нескольких линий в одной координатной сетке удобно создать легенду, описывающую каждый тип линии. В библиотеке Matplotlib для быстрого создания такой легенды имеется встроенный метод `plt.legend`. Хотя существует несколько возможных способов, проще всего, как мне кажется, за-

дать метку каждой линии с помощью именованного аргумента `label` функции `plot` (рис. 26.13):

```
In [14]: plt.plot(x, np.sin(x), '-g', label='sin(x)')
         plt.plot(x, np.cos(x), ':b', label='cos(x)')
         plt.axis('equal')

         plt.legend();
```

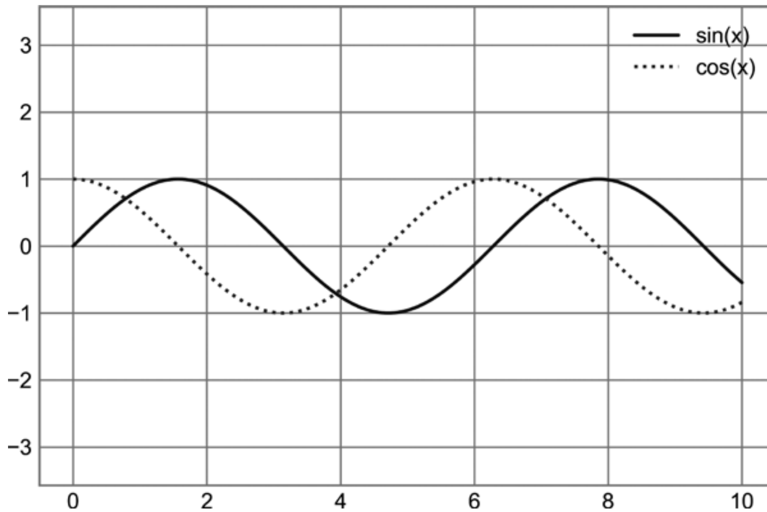


Рис. 26.13. Пример легенды графика

Функция `plt.legend` отслеживает стиль и цвет линий и устанавливает их соответствие с нужной меткой. Больше информации по созданию и оформлению легенд графиков можно найти в docstring метода `plt.legend`. Дополнительные параметры легенд мы рассмотрим в главе 29.

Нюансы использования Matplotlib

Хотя для большинства функций интерфейса `plt` соответствующие методы интерфейса `ax` имеют такие же имена (например, `plt.plot` → `ax.plot`, `plt.legend` → `ax.legend` и т. д.), это касается не всех команд. В частности, функции для задания пределов, меток и названий графиков имеют несколько иные имена. Вот список соответствий между функциями и методами MATLAB-подобного и объектно-ориентированного интерфейсов:

- `plt.xlabel` → `ax.set_xlabel`;
- `plt.ylabel` → `ax.set_ylabel`;

- plt.xlim → ax.set_xlim;
- plt.ylim → ax.set_ylim;
- plt.title → ax.set_title.

В объектно-ориентированном интерфейсе построения графиков вместо вызова этих функций по отдельности удобнее использовать метод ax.set, чтобы задать значения всех этих параметров за один раз (рис. 26.14):

```
In [15]: ax = plt.axes()
         ax.plot(x, np.sin(x))
         ax.set(xlim=(0, 10), ylim=(-2, 2),
                xlabel='x', ylabel='sin(x)',
                title='A Simple Plot'); # Простая диаграмма
```

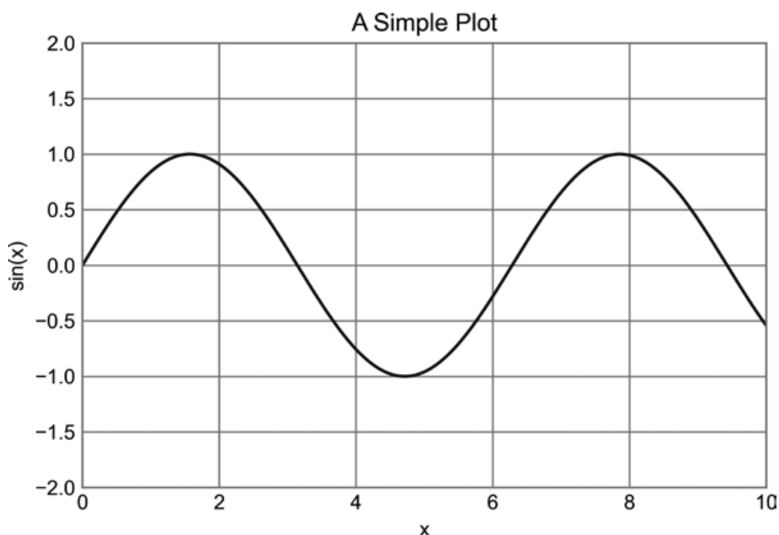


Рис. 26.14. Пример использования метода ax.set() для задания значений нескольких параметров за один вызов

ГЛАВА 27

Простые диаграммы рассеяния

Еще один часто используемый тип графиков — диаграммы рассеяния, родственные линейным графикам. В них точки не соединяются отрезками линий, а представлены по отдельности точками, кругами или другими фигурами. Начнем с настройки блокнота для построения графиков и импорта необходимых функций:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Построение диаграмм рассеяния с помощью plt.plot

В предыдущей главе мы рассмотрели построение линейных графиков посредством функции `plt.plot`/`ax.plot`. С ее помощью можно строить и диаграммы рассеяния (рис. 27.1):

```
In [2]: x = np.linspace(0, 10, 30)
        y = np.sin(x)

        plt.plot(x, y, 'o', color='black');
```

Третий аргумент в вызове этой функции описывает тип символа, применяемого для представления точек. Для управления стилем линии можно использовать такие значения, как `'-'` и `'--'`. Аналогично для управления стилем маркеров существует свой набор коротких кодов. Полный список можно найти в документации по функции `plt.plot` или в онлайн-документации по библиотеке Matplotlib (https://oreil.ly/tmYIL). Большинство вариантов интуитивно понятны, и ниже мы продемонстрируем используемые наиболее часто (рис. 27.2):

```
In [3]: rng = np.random.default_rng(0)
        for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
            plt.plot(rng.random(2), rng.random(2), marker, color='black',
                     label="marker='{0}'".format(marker))
        plt.legend(numpoints=1, fontsize=13)
        plt.xlim(0, 1.8);
```

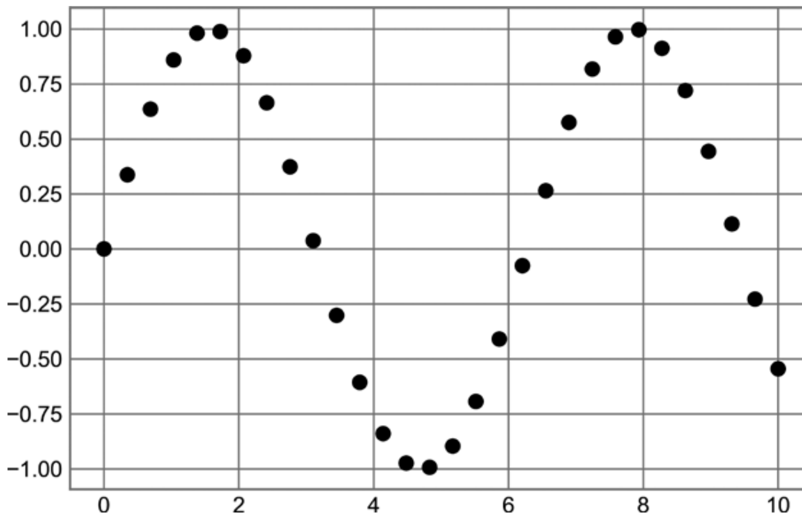


Рис. 27.1. Пример диаграммы рассеяния

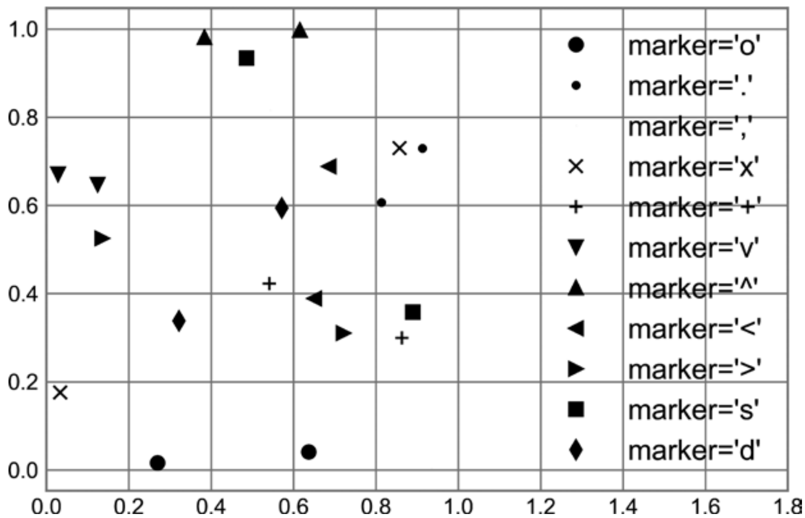


Рис. 27.2. Демонстрация различных маркеров

Коды, описывающие стиль маркеров, можно комбинировать с кодами цветов и стилей линий, соединяющих маркеры, что открывает еще более широкие возможности (рис. 27.3):

```
In [4]: plt.plot(x, y, '-ok');
```

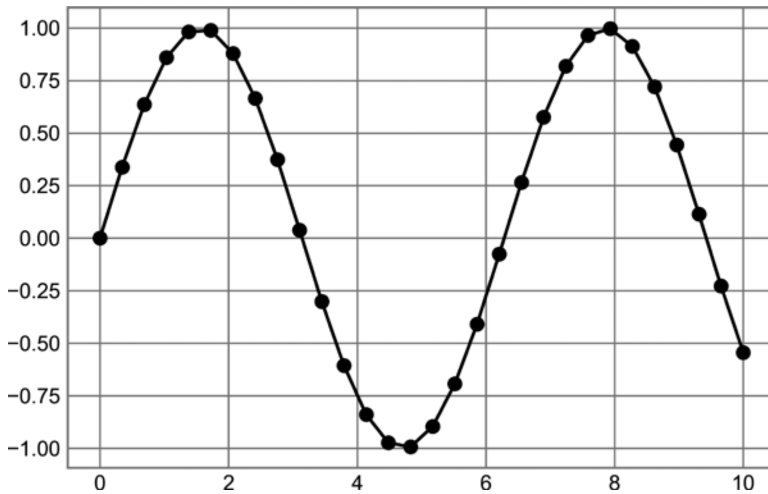


Рис. 27.3. Комбинирование стилей линий и маркеров

С помощью дополнительных именованных аргументов функции `plt.plot` можно задавать множество свойств линий и маркеров (рис. 27.4):

```
In [5]: plt.plot(x, y, '-p', color='gray',
                 markersize=15, linewidth=4,
                 markerfacecolor='white',
                 markeredgecolor='gray',
                 markeredgewidth=2)
        plt.ylim(-1.2, 1.2);
```

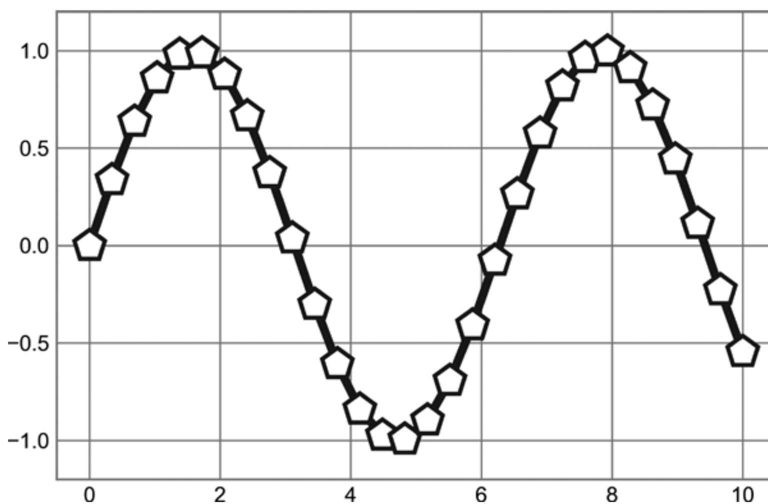


Рис. 27.4. Индивидуальная настройка вида линий и маркеров точек

Подобная гибкость функции `plt.plot` позволяет использовать широкий диапазон настроек визуализации. Полное описание имеющихся настроек можно найти в документации по функции `plt.plot`.

Построение диаграмм рассеяния с помощью plt.scatter

Еще большими возможностями обладает метод построения диаграмм рассеяния `plt.scatter`, во многом напоминающий функцию `plt.plot` (рис. 27.5):

```
In [6]: plt.scatter(x, y, marker='o');
```

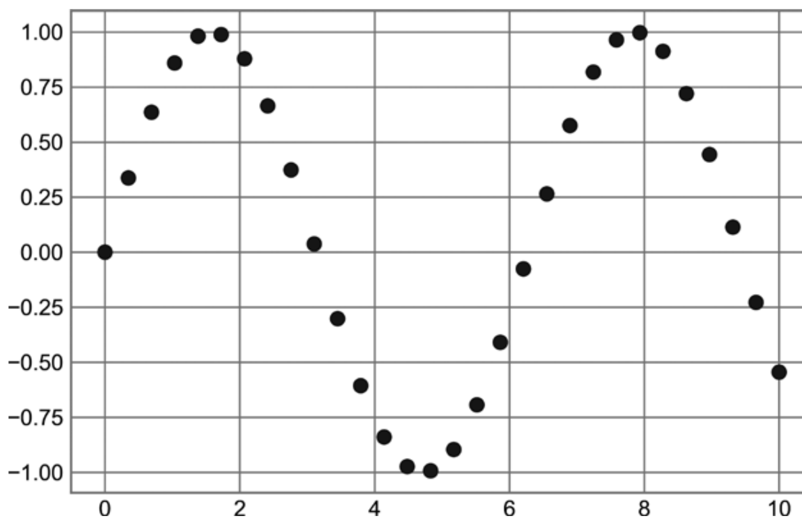


Рис. 27.5. Простая диаграмма рассеяния

Основное различие между `plt.scatter` и `plt.plot` состоит в том, что с помощью первой можно создавать диаграммы рассеяния с индивидуально задаваемыми (или выбираемыми в соответствии с данными) свойствами каждой точки (размер, цвет заливки, цвет рамки и т. д.).

Продемонстрируем это, создав случайную диаграмму рассеяния с точками различных цветов и размеров. Чтобы лучше видеть перекрывающиеся результаты, воспользуемся именованным аргументом `alpha` для настройки уровня прозрачности (рис. 27.6):

```
In [7]: rng = np.random.default_rng(0)
        x = rng.normal(size=100)
        y = rng.normal(size=100)
        colors = rng.random(100)
        sizes = 1000 * rng.random(100)

        plt.scatter(x, y, c=colors, s=sizes, alpha=0.3)
        plt.colorbar();  # Отображаем цветовую шкалу
```

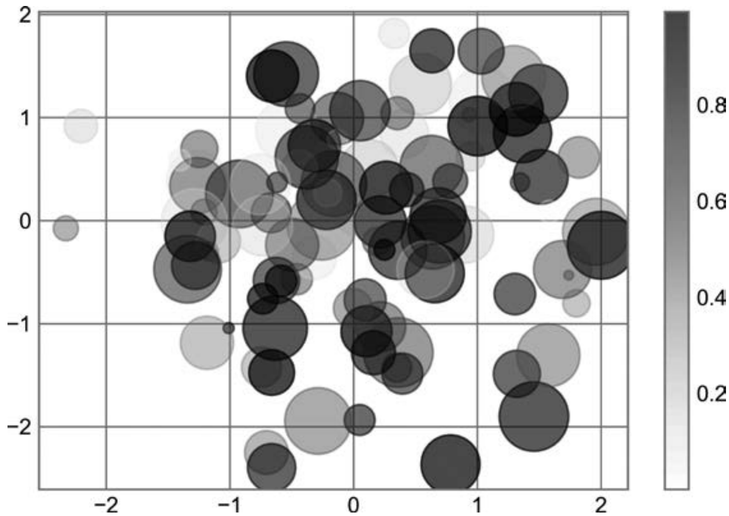


Рис. 27.6. Изменение размера, цвета и прозрачности точек на диаграмме рассеяния

Обратите внимание, что цвета автоматически привязываются к цветовой шкале (которую мы отобразили с помощью команды `colorbar`), а размеры указываются в пикселах. Используя такое решение, можно задавать цвет и размер точек для передачи информации на графике с целью иллюстрации многомерных данных.

Для примера возьмем набор данных Iris из библиотеки Scikit-Learn. Каждая выборка — это один из трех типов цветков с тщательно измеренными лепестками и чашелистиками (рис. 27.7):

```
In [8]: from sklearn.datasets import load_iris
        iris = load_iris()
        features = iris.data.T

        plt.scatter(features[0], features[1], alpha=0.4,
                    s=100*features[3], c=iris.target, cmap='viridis')
        plt.xlabel(iris.feature_names[0])
        plt.ylabel(iris.feature_names[1]);
```

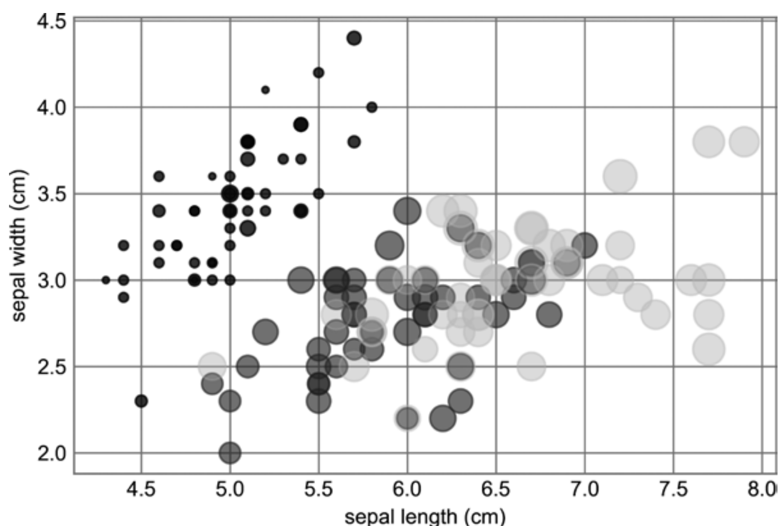


Рис. 27.7. Использование свойств точек для передачи признаков в наборе данных Iris

Как видите, эта диаграмма рассеяния позволяет исследовать сразу четыре различных измерения данных: (x, y)-координаты каждой точки соответствуют длине и ширине чашелистика, размер точки — ширине лепестков, цвет — конкретной разновидности цветка. Подобные многоцветные диаграммы рассеяния с несколькими признаками для каждой точки очень полезны как для исследования, так и для представления данных.

plot и scatter: примечание относительно производительности

Помимо различных возможностей, предоставляемых `plt.plot` и `plt.scatter`, какие еще характеристики могут влиять на выбор той или иной из них? При небольших объемах данных это не играет роли, но при работе с наборами, объем которых превышает несколько тысяч точек, функция `plt.plot` может оказаться намного эффективнее `plt.scatter`. Поскольку `plt.scatter` позволяет отображать точки с разными размерами и цветами, механизму визуализации приходится выполнять дополнительную работу по формированию этих точек. Функция `plt.plot`, напротив, всегда отображает точки как точные копии друг друга, поэтому работа по определению внешнего вида точек выполняется только один раз для всего набора данных. При визуализации больших наборов данных это различие может приводить к существенным различиям в производительности, поэтому в таких случаях следует использовать функцию `plt.plot`, а не `plt.scatter`.

Визуализация погрешностей

Точный учет погрешностей, как и точная информация о самих значениях, важен для любых научных измерений. Например, представьте, что я использую некоторые астрофизические наблюдения для оценки постоянной Хаббла, то есть измеряю скорость расширения Вселенной в данной точке. Мне известно, что в современных источниках по этому вопросу указывается значение около 70 (км/с)/Мпк, а я с помощью моего метода получил значение 74 (км/с)/Мпк. Не противоречат ли значения друг другу? По вышеприведенной информации ответить на этот вопрос невозможно.

Теперь допустим, что я дополнил эту информацию погрешностями: современные источники указывают значение около 70 ± 2,5 (км/с)/Мпк, а мой метод дал значение 74 ± 5 (км/с)/Мпк. Не противоречат ли теперь значения друг другу? Это вопрос, на который вполне можно дать количественный ответ.

При визуализации данных и результатов эффективное отображение погрешностей позволяет передавать с помощью графика намного более полную информацию.

Простые планки погрешностей

Простые планки погрешностей можно создать с помощью вызова всего одной функции библиотеки Matplotlib (рис. 27.8):

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
In [2]: x = np.linspace(0, 10, 50)
        dy = 0.8
        y = np.sin(x) + dy * np.random.randn(50)

        plt.errorbar(x, y, yerr=dy, fmt='.k');
```

Здесь `fmt` — код форматирования, управляющий внешним видом линий и точек, его синтаксис совпадает с кратким синтаксисом, используемым в функции `plt.plot`, описываемой в предыдущей главе.

Помимо этих простейших, функция `errorbar` имеет множество других параметров для тонкой настройки выводимых данных. С помощью этих дополнительных параметров можно легко настроить внешний вид графика планок погрешностей в соответствии со своими требованиями. Планки погрешностей удобно делать более светлыми, чем точки, особенно на насыщенных графиках (рис. 27.9):

```
In [3]: plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
                     ecolor='lightgray', elinewidth=3, capsize=0);
```

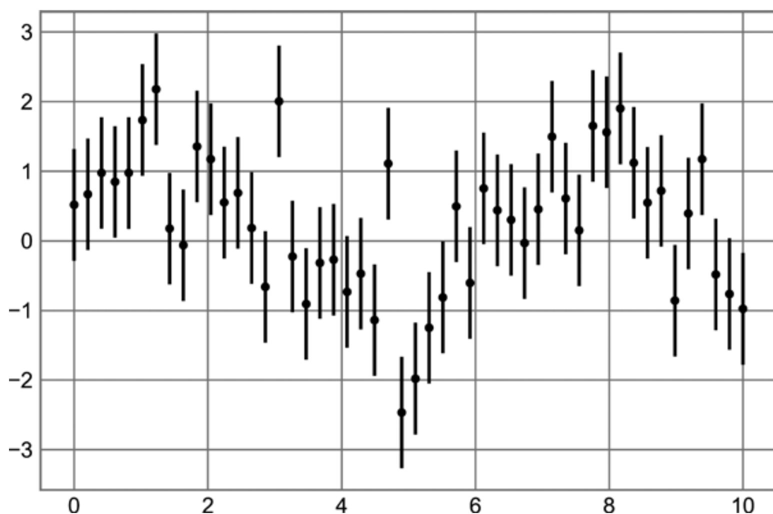


Рис. 27.8. Пример планок погрешностей

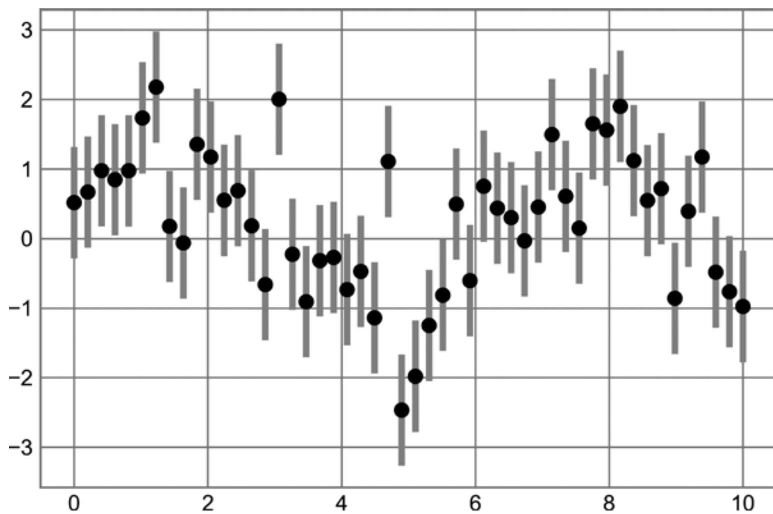


Рис. 27.9. Вид планок погрешностей с измененными настройками

Кроме того, есть также возможность создавать горизонтальные планки погрешностей, односторонние планки погрешностей и много других их вариантов. Чтобы узнать больше об имеющихся возможностях, обращайтесь к docstring функции `plt.errorbar`.

Непрерывные погрешности

В некоторых случаях желательно отображать планки погрешностей для непрерывных величин. Хотя в библиотеке Matplotlib отсутствуют встроенные утилиты для решения данной задачи, не составит особого труда скомбинировать такие примитивы, как `plt.plot` и `plt.fill_between`, для получения искомого результата.

Выполним с помощью пакета Scikit-Learn (см. подробности в главе 38) простую *регрессию на основе гауссова процесса* (Gaussian process regression, GPR). Это метод подбора, по имеющимся данным, очень гибкой непараметрической функции с непрерывной мерой неопределенности измерения. Мы не будем углубляться в детали регрессии на основе гауссова процесса, а сконцентрируемся на визуализации подобной непрерывной погрешности измерения:

```
In [4]: from sklearn.gaussian_process import GaussianProcessRegressor

        # Описываем модель и отрисовываем некоторые данные
        model = lambda x: x * np.sin(x)
        xdata = np.array([1, 3, 5, 6, 8])
        ydata = model(xdata)

        # Выполняем подгонку гауссова процесса
        gp = GaussianProcessRegressor()
        gp.fit(xdata[:, np.newaxis], ydata)

        xfit = np.linspace(0, 10, 1000)
        yfit, dyfit = gp.predict(xfit[:, np.newaxis], return_std=True)
```

Теперь у нас имеются значения `xfit`, `yfit` и `dyfit`, представляющие выборку непрерывных приближений к нашим данным. Мы можем передать их в вызов функции `plt.errorbar`, но не хотелось бы рисовать 1000 точек с 1000 планок погрешностей. Вместо этого можно воспользоваться функцией `plt.fill_between` и визуализировать эту непрерывную погрешность в виде области со светлой заливкой (рис. 27.10):

```
In [5]: # Визуализируем результат
        plt.plot(xdata, ydata, 'or')
        plt.plot(xfit, yfit, '-', color='gray')
        plt.fill_between(xfit, yfit - dyfit, yfit + dyfit,
                         color='gray', alpha=0.2)
        plt.xlim(0, 10);
```

Обратите внимание на параметры, передаваемые функции `plt.fill_between`: мы передали ей значение `x`, затем нижнюю границу по `y`, затем верхнюю границу по `y`, и в результате получили область с заливкой между ними.

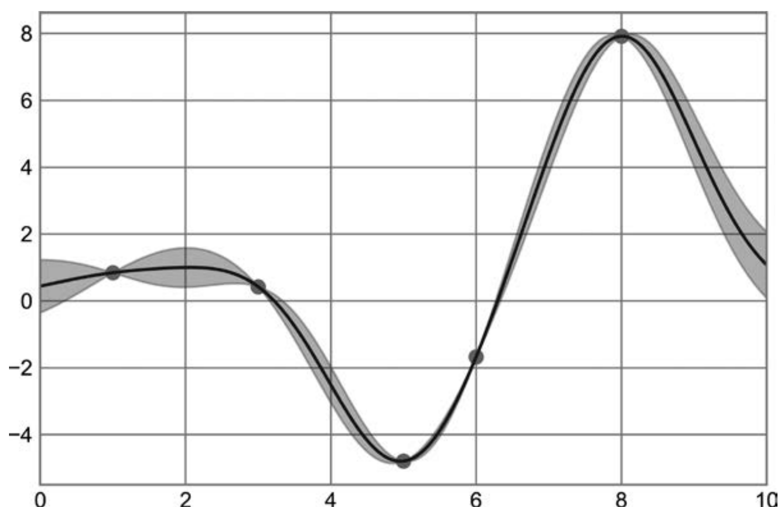


Рис. 27.10. Представляем непрерывную погрешность в виде областей с заливкой

Получившийся рисунок отлично поясняет, что делает алгоритм регрессии на основе гауссова процесса. В областях возле измеренной точки данных модель жестко ограничена, что и отражается в малых ошибках модели. В областях, удаленных от измеренной точки, модель не имеет жестких ограничений и ошибки модели растут.

Чтобы узнать больше о возможностях функции `plt.fill_between` (и родственной ей функции `plt.fill`), см. ее docstring или документацию библиотеки Matplotlib.

Наконец, если описанное вам не по вкусу, загляните в главу 36, где обсуждается пакет Seaborn, предлагающий более прямолинейный API для визуализации подобных непрерывных погрешностей.

ГЛАВА 28

Графики плотности и контурные графики

Отображать трехмерные данные в двумерной плоскости иногда удобнее с помощью контуров или цветных областей. В библиотеке Matplotlib существуют три функции, предназначенные для этой цели:

- `plt.contour` — для контурных графиков;
- `plt.contourf` — для контурных графиков с заливкой цветом;
- `plt.imshow` — для отображения изображений.

В этой главе мы рассмотрим несколько примеров использования перечисленных функций. Начнем с настройки блокнота для построения графиков и импорта необходимых функций:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-white')
        import numpy as np
```

Визуализация трехмерной функции

Начнем с демонстрации контурного графика для функции вида $z = f(x, y)$, воспользовавшись для этой цели функцией f (мы уже встречались с ней в главе 8, в примере транслирования массивов):

```
In [2]: def f(x, y):
            return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

Создать контурный график можно с помощью функции `plt.contour`. Она принимает три аргумента: координатную сетку значений x, координатную сетку значений y и координатную сетку значений z. Значения x и y представлены точками

на графике, а значения z — контурами уровней. Вероятно, наиболее простой способ подготовить такие данные — воспользоваться функцией `np.meshgrid`, формирующей двумерные координатные сетки из одномерных массивов:

```
In [3]: x = np.linspace(0, 5, 50)
        y = np.linspace(0, 5, 40)
        X, Y = np.meshgrid(x, y)
        Z = f(X, Y)
```

Теперь посмотрим на эти данные, визуализировав их в виде обычного линейного контурного графика (рис. 28.1):

```
In [4]: plt.contour(X, Y, Z, colors='black');
```

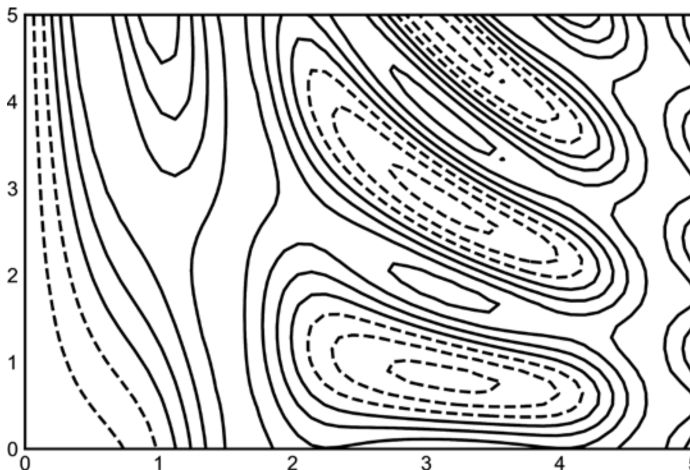


Рис. 28.1. Визуализация трехмерных данных с помощью контуров

Обратите внимание, что по умолчанию при использовании одного цвета отрицательные значения обозначаются штриховыми линиями, а положительные — сплошными. Можно также кодировать линии различными цветами, задав карты цветов с помощью аргумента `cmap`. В следующем примере мы решили нарисовать больше линий, разделив данные на 20 интервалов с равными промежутками (рис. 28.2):

```
In [5]: plt.contour(X, Y, Z, 20, cmap='RdGy');
```

Мы выбрали карту цветов `RdGy` (сокращенно от Red-Gray — «красно-серая») — отличный выбор для случая центрированных данных (то есть данных со средним значением, равным нулю). В библиотеке Matplotlib доступен широкий диапазон карт цветов, список которых можно получить в IPython с помощью автодополнения по клавише TAB, применительно к модулю `plt.cm`:

```
plt.cm.<TAB>
```

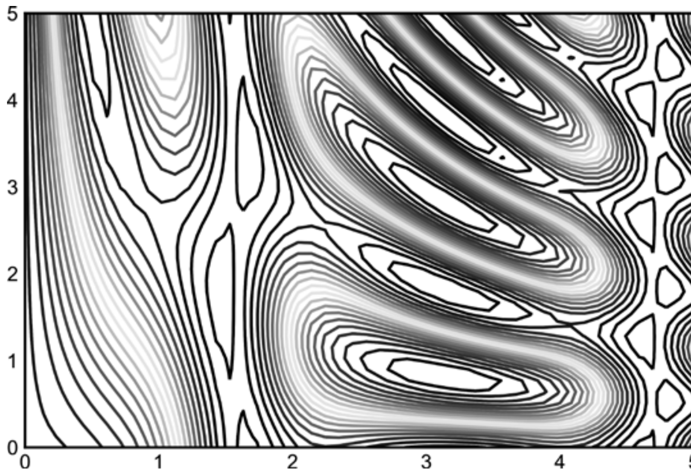


Рис. 28.2. Визуализация трехмерных данных с помощью разноцветных контуров

Наш график приобрел более приятный глазу вид, но промежутки между линиями несколько отвлекают внимание. Решить эту проблему можно, воспользовавшись функцией `plt.contourf`, которая создает контурный график с заливкой. Ее синтаксис практически не отличается от синтаксиса функции `plt.contour`.

Дополнительно воспользуемся командой `plt.colorbar`, которая автоматически создает дополнительную ось с информацией о цвете (рис. 28.3):

```
In [6]: plt.contourf(X, Y, Z, 20, cmap='RdGy')
        plt.colorbar();
```

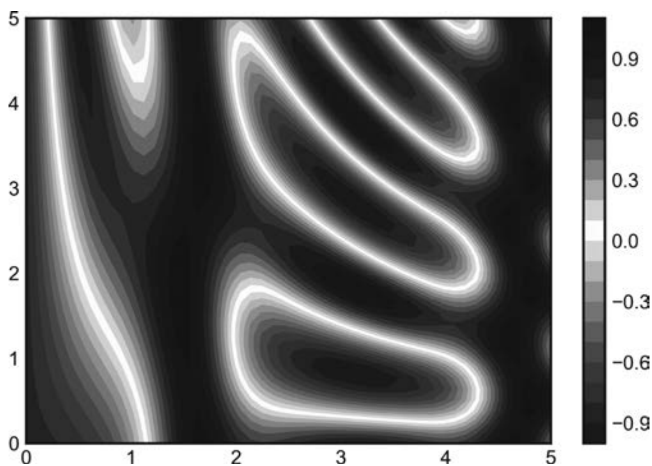


Рис. 28.3. Визуализация трехмерных данных с помощью контуров с заливкой

Цветовая шкала справа на рис. 28.3 наглядно демонстрирует, что черные области соответствуют максимумам, а светлые — минимумам.

Потенциальная проблема такого графика — его «пятнистость». Градации цветов на графике дискретны, а не непрерывны, что не всегда удобно. Исправить это можно, задав большее количество контуров, что ухудшит производительность: библиотеке Matplotlib придется визуализировать новый полигон для каждого шага уровня. Лучшее решение — воспользоваться функцией `plt.imshow`, которая принимает аргумент `interpolation`, определяющий алгоритм для получения сглаженного двумерного представления данных (рис. 28.4):

```
In [7]: plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower', cmap='RdGy',
                   interpolation='gaussian', aspect='equal')
        plt.colorbar();
```

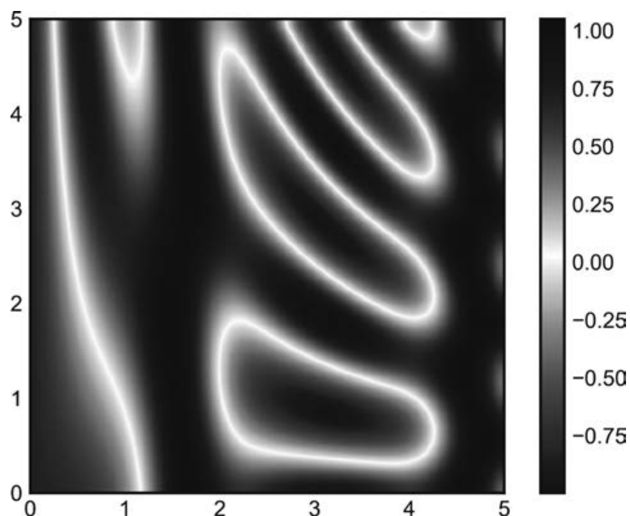


Рис. 28.4. Представление трехмерных данных в виде сглаженного двумерного изображения

Однако `plt.imshow` имеет несколько недостатков:

- она не принимает сетку *x* и *y*, поэтому ей нужно явно указать границы [*xmin*, *xmax*, *ymin*, *ymax*] изображения на графике;
- по умолчанию она использует стандартное соглашение для изображений, в соответствии с которым начало координат находится в левом верхнем углу, а не в левом нижнем. Это необходимо учитывать при отображении данных с координатной сеткой;
- она автоматически регулирует отношение сторон в соответствии с входными данными; это поведение можно изменить с помощью аргумента `aspect`.

Наконец, иногда бывает полезно объединить контурный график и график в виде сглаженного изображения. Следующий пример генерирует полупрозрачное фоновое изображение (передачей параметра alpha) и накладывает на него контуры с метками, используя функцию plt.clabel (рис. 28.5).

```
In [8]: contours = plt.contour(X, Y, Z, 3, colors='black')
        plt.clabel(contours, inline=True, fontsize=8)

        plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
                   cmap='RdGy', alpha=0.5)
        plt.colorbar();
```

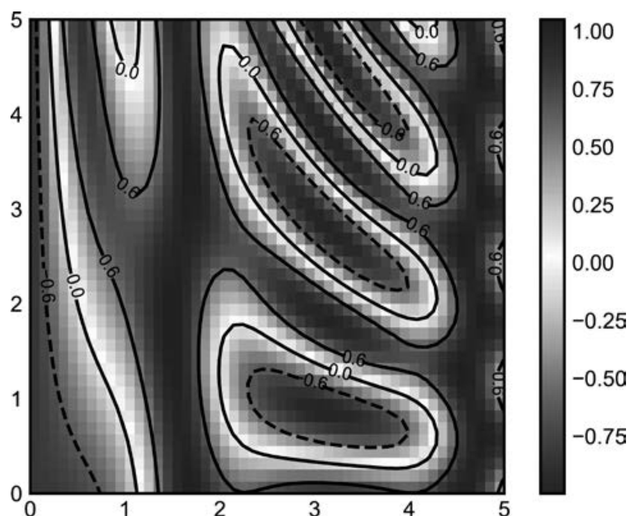


Рис. 28.5. Контуры с метками поверх сглаженного изображения

Возможность комбинирования этих трех функций — plt.contour, plt.contourf и plt.imshow — открывает практически неограниченные варианты отображения трехмерных данных на двумерных графиках. Дополнительную информацию о параметрах этих функций вы найдете в их docstring. Если вас интересует трехмерная визуализация таких данных, загляните в главу 35.

Гистограммы, разбиения по интервалам и плотность

Простая гистограмма может принести огромную пользу при первичном анализе набора данных. Ранее мы видели пример использования функции из библиотеки Matplotlib (обсуждалась в главе 9) для создания простой гистограммы в одну

строку, которая становится доступна после выполнения всех обычных инструкций импортирования (рис. 28.6):

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-white')

        rng = np.random.default_rng(1701)
        data = rng.normal(size=1000)

In [2]: plt.hist(data);
```

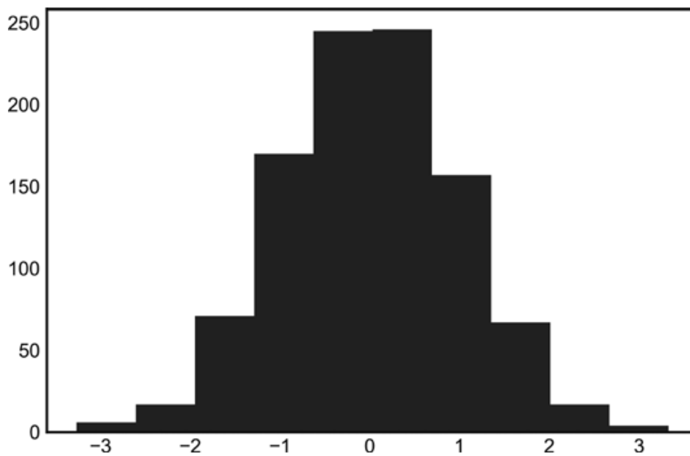


Рис. 28.6. Простая гистограмма

Функция `hist` имеет множество параметров для настройки как вычисления, так и отображения. Вот пример гистограммы с некоторыми дополнительными настройками (рис. 28.7):

```
In [3]: plt.hist(data, bins=30, density=True, alpha=0.5,
                 histtype='stepfilled', color='steelblue',
                 edgecolor='none');
```

Описательный комментарий (docstring) функции `plt.hist` содержит более подробную информацию о других доступных возможностях настройки. Сочетание параметра `histtype='stepfilled'` с заданной прозрачностью `alpha` очень удобно использовать для сравнения гистограмм нескольких распределений (рис. 28.8):

```
In [4]: x1 = rng.normal(0, 0.8, 1000)
        x2 = rng.normal(-2, 1, 1000)
        x3 = rng.normal(3, 2, 1000)

        kwargs = dict(histtype='stepfilled', alpha=0.3, density=True, bins=40)
```

```
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```

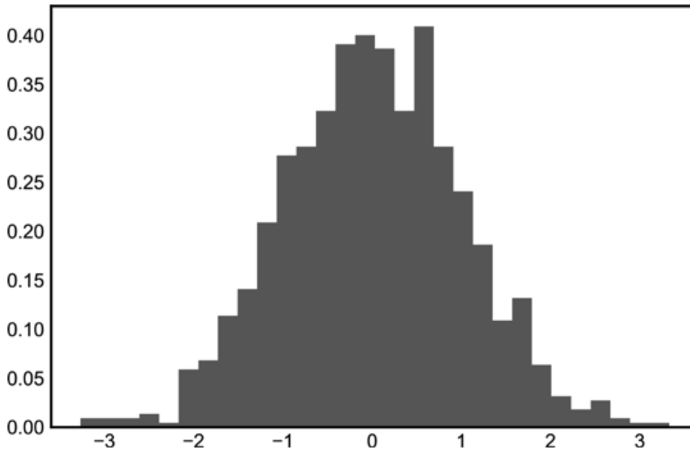


Рис. 28.7. Гистограмма с дополнительными настройками

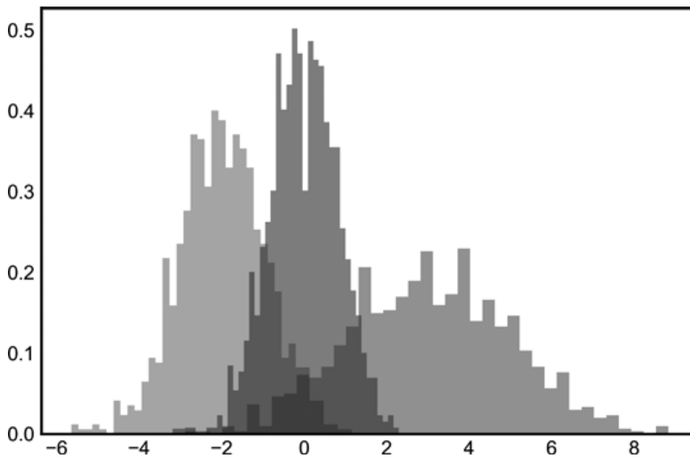


Рис. 28.8. Несколько гистограмм, наложенных друг на друга

Если нужно лишь вычислить гистограмму (то есть подсчитать количество точек в заданном интервале), но не отображать ее, то к вашим услугам функция `np.histogram`:

```
In [5]: counts, bin_edges = np.histogram(data, bins=5)
        print(counts)
Out[5]: [ 23 241 491 224  21]
```

Двумерные гистограммы и разбиение по интервалам

Аналогично тому, как создаются одномерные гистограммы, разбиением последовательности чисел по интервалам можно создавать и двумерные гистограммы, распределяя точки по двумерным интервалам. Рассмотрим несколько примеров, начав с описания данных массивов `x` и `y`, полученных из многомерного гауссова распределения:

```
In [6]: mean = [0, 0]
        cov = [[1, 1], [1, 2]]
        x, y = rng.multivariate_normal(mean, cov, 10000).T
```

Функция plt.hist2d: двумерная гистограмма

Один из простых способов нарисовать двумерную гистограмму — воспользоваться функцией `plt.hist2d` из библиотеки Matplotlib (рис. 28.9).

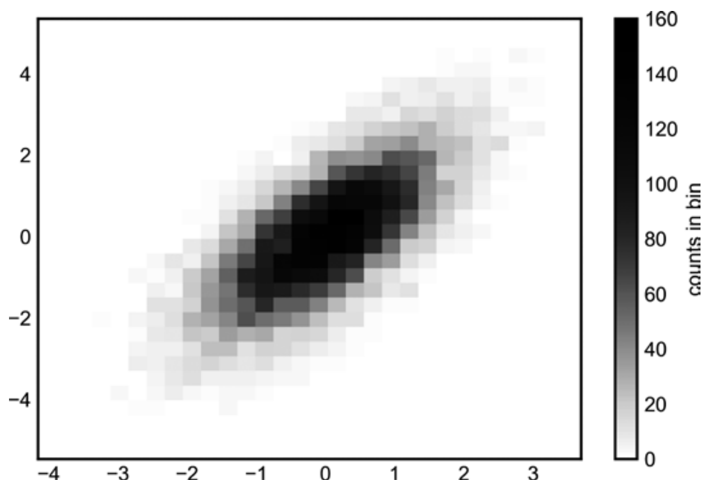


Рис. 28.9. Двумерная гистограмма, построенная с помощью функции plt.hist2d

```
In [7]: plt.hist2d(x, y, bins=30)
          cb = plt.colorbar()
          cb.set_label('counts in bin')
```

Функция `plt.hist2d`, как и `plt.hist`, имеет множество дополнительных параметров для тонкой настройки отображения графика и разбиения по интервалам, подроб-

но описанных в ее docstring. Подобно функции `plt.hist`, имеющей эквивалент `np.histogram`, функция `plt.hist2d` тоже имеет эквивалент `np.histogram2d`:

```
In [8]: counts, xedges, yedges = np.histogram2d(x, y, bins=30)
        print(counts.shape)
Out[8]: (30, 30)
```

Обобщенное разбиение по интервалам с числом измерений больше 2 можно выполнить с помощью функции `np.histogramdd`.

Функция plt.hexbin: гексагональное разбиение по интервалам

Двумерная гистограмма создает мозаичное представление квадратами вдоль координатных осей. Однако для подобного мозаичного представления прекрасно подойдет еще одна геометрическая фигура — правильный шестиугольник. Библиотека Matplotlib предоставляет функцию `plt.hexbin`, формирующую двумерный набор данных, разбитых по сетке из шестиугольников (рис. 28.10):

```
In [9]: plt.hexbin(x, y, gridsize=30)
        cb = plt.colorbar(label='count in bin')
```

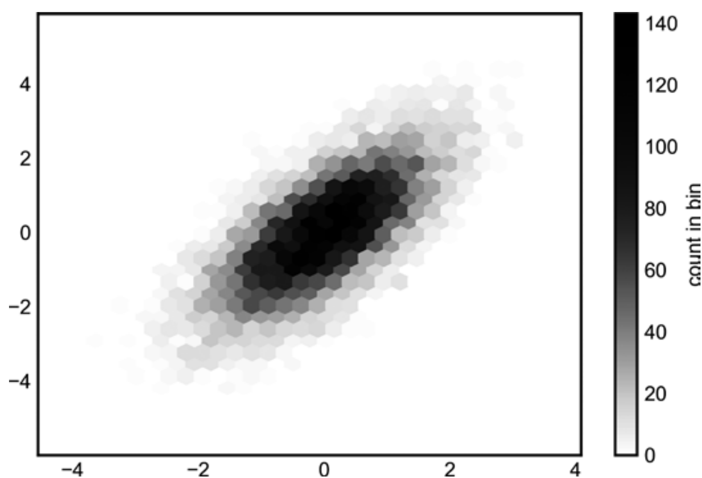


Рис. 28.10. Двумерная гистограмма, построенная с помощью функции plt.hexbin

Функция `plt.hexbin` имеет множество дополнительных параметров, позволяющих, кроме всего прочего, задавать вес для каждой точки и выводить для каждого интер-

вала любой сводный показатель, поддерживаемый библиотекой NumPy (среднее значение весов, стандартное отклонение весов и т. д.).

Ядерная оценка плотности распределения

Еще один часто используемый метод оценки плотностей в многомерном пространстве — *ядерная оценка плотности распределения* (kernel density estimation, KDE). Более подробно мы разберем ее в главе 49, а пока отметим, что KDE можно рассматривать как способ «размазать» точки в пространстве и сложить результаты для получения гладкой функции. В пакете `scipy.stats` имеется исключительно быстрая и простая реализация KDE. Вот короткий пример ее использования (рис. 28.11):

```
In [10]: from scipy.stats import gaussian_kde

         # Выполняем подбор на массиве размера [Ndim, Nsamples]
         data = np.vstack([x, y])
         kde = gaussian_kde(data)

         # Вычисляем на регулярной координатной сетке
         xgrid = np.linspace(-3.5, 3.5, 40)
         ygrid = np.linspace(-6, 6, 40)
         Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
         Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))

         # Выводим результат в виде сглаженного изображения
         plt.imshow(Z.reshape(Xgrid.shape),
                    origin='lower', aspect='auto',
                    extent=[-3.5, 3.5, -6, 6])
         cb = plt.colorbar()
         cb.set_label("density")
```

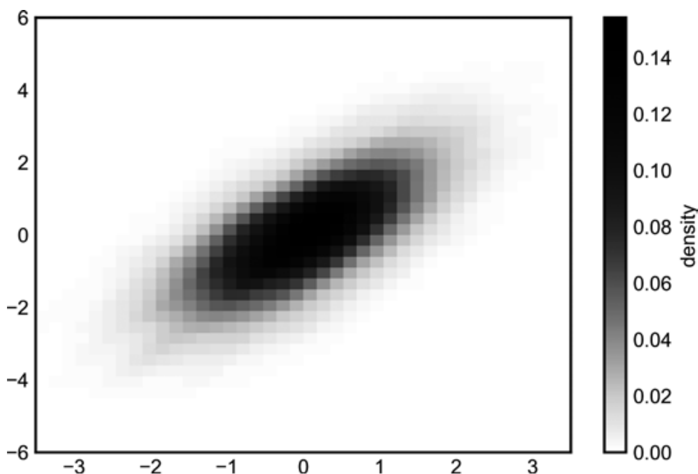


Рис. 28.11. Ядерная оценка плотности распределения

Длина сглаживания метода KDE позволяет эффективно выбирать компромисс между гладкостью и детальностью (один из примеров вездесущих компромиссов между смещением и дисперсией). Существует обширная литература, посвященная выбору подходящей длины сглаживания: в функции `gaussian_kde` используется эмпирическое правило для поиска квазиоптимальной длины сглаживания для входных данных.

В экосистеме SciPy имеются и другие реализации метода KDE, каждая со своими сильными и слабыми сторонами, например методы `sklearn.neighbors.KernelDensity` и `statsmodels.nonparametric.kernel_density.KDEMultivariate`.

Использование библиотеки Matplotlib для создания визуализаций на основе метода KDE требует писать избыточный код. Библиотека Seaborn, которую мы обсудим в главе 36, предлагает для создания таких визуализаций функции с намного более компактным синтаксисом.

ГЛАВА 29

Настройка легенд на графиках

Легенды придают осмысленность графикам, описывая различные элементы графика. Мы ранее уже рассматривали создание простой легенды, а здесь покажем дополнительные возможности настройки местоположения и внешнего вида легенд в Matplotlib.

С помощью команды `plt.legend` можно автоматически создать простейшую легенду для любых маркированных элементов графика (рис. 29.1):

```
In [1]: import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
In [2]: %matplotlib inline
        import numpy as np
In [3]: x = np.linspace(0, 10, 1000)
        fig, ax = plt.subplots()
        ax.plot(x, np.sin(x), '-b', label='Sine')    # Синус
        ax.plot(x, np.cos(x), '--r', label='Cosine') # Косинус
        ax.axis('equal')
        leg = ax.legend()
```

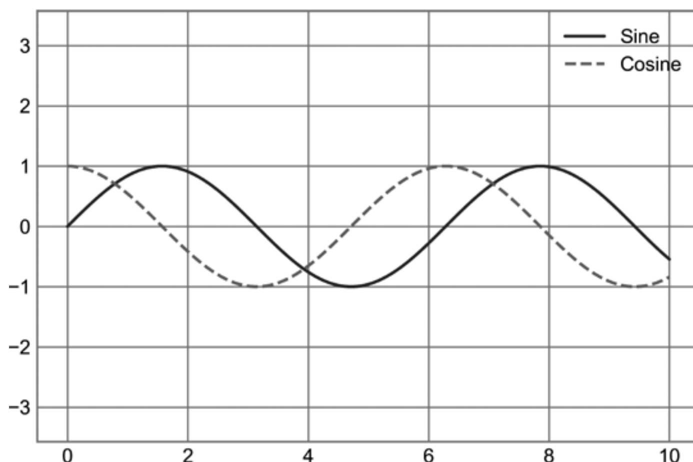


Рис. 29.1. Легенда графика по умолчанию

Существует множество вариантов настройки легенды. Например, можно задать местоположение легенды и включить рамку (рис. 29.2):

```
In[4]: ax.legend(loc='upper left', frameon=True)
       fig
```

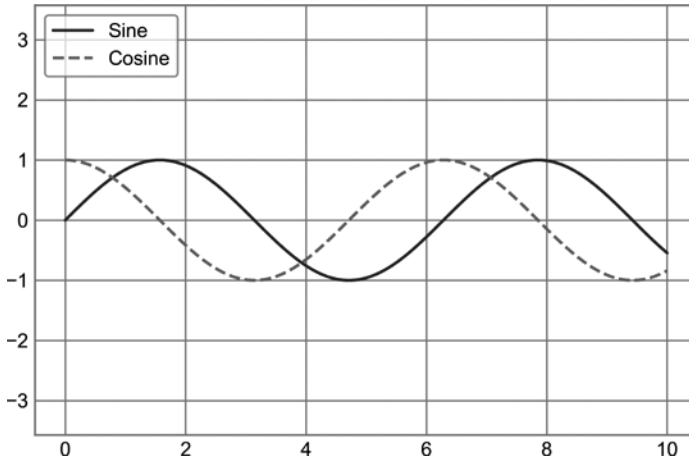


Рис. 29.2. Вывод легенды с дополнительными настройками

Можно воспользоваться командой `ncol`, чтобы задать количество столбцов в легенде (рис. 29.3):

```
In [5]: ax.legend(loc='lower center', ncol=2)
        Fig
```

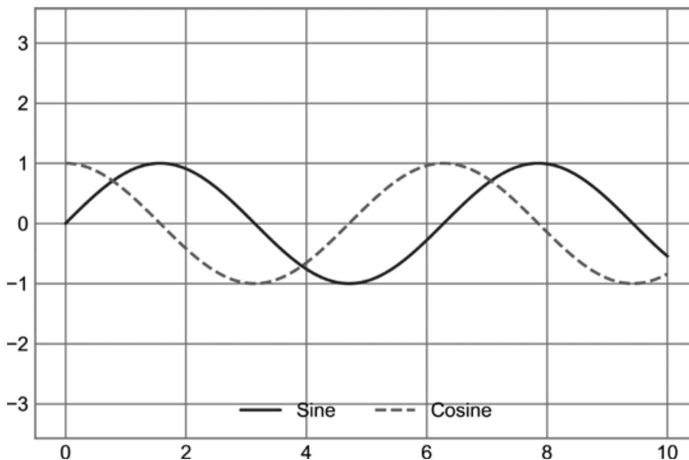


Рис. 29.3. Вывод легенды в два столбца

Вокруг легенды можно нарисовать скругленную прямоугольную рамку (`fancybox`) или добавить тень, поменять прозрачность (альфа-фактор) рамки или поля, окружающего текст (рис. 29.4):

```
In [6]: ax.legend(frameon=True, fancybox=True, framealpha=1,
                  shadow=True, borderpad=1)
        fig
```

Дополнительную информацию о доступных настройках легенд можно найти в docstring функции `plt.legend`.

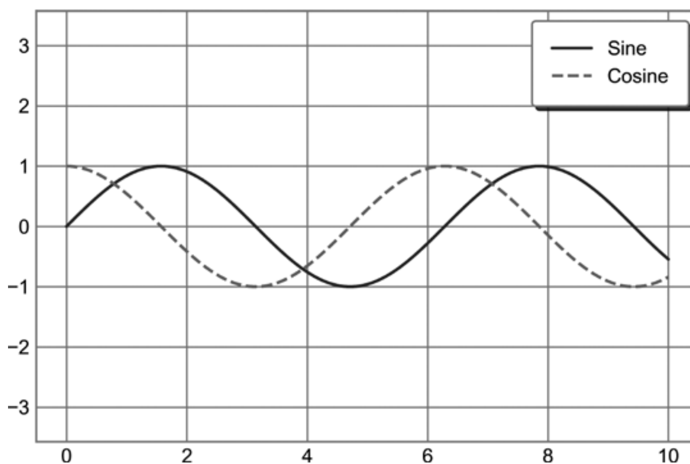


Рис. 29.4. Легенда в скругленной прямоугольной рамке

Выбор элементов для легенды

По умолчанию легенда включает все маркированные элементы. Если в этом нет необходимости, то можно указать, какие элементы и метки должны присутствовать в легенде, воспользовавшись объектами, возвращаемыми командами `plot`. Команда `plt.plot` может нарисовать несколько линий за один вызов и вернуть список созданных экземпляров линий. Передав любой из них в `plt.legend` вместе с соответствующей меткой, можно сообщить о необходимости отражения этого элемента в легенде (рис. 29.5):

```
In [7]: y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
        lines = plt.plot(x, y)

        # lines — это список экземпляров класса plt.Line2D
        plt.legend(lines[:2], ['first', 'second'], frameon=True); # первый, второй
```

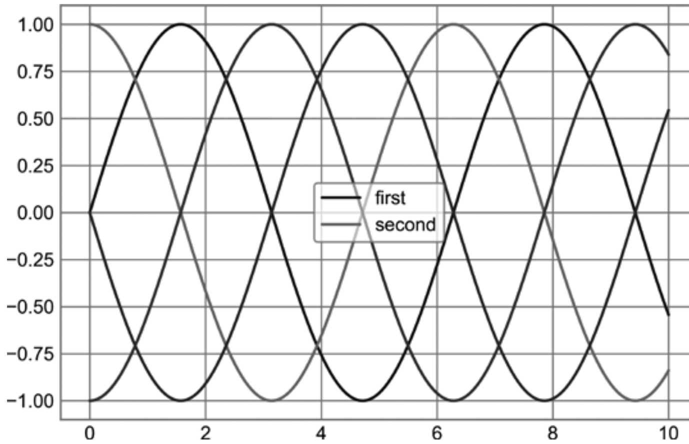


Рис. 29.5. Настройка содержимого легенды

Обычно на практике мне удобнее использовать первый способ, указывая метки непосредственно для элементов, которые нужно отобразить в легенде (рис. 29.6):

```
In [8]: plt.plot(x, y[:, 0], label='first')  # первый
        plt.plot(x, y[:, 1], label='second') # второй
        plt.plot(x, y[:, 2:])
        plt.legend(frameon=True);
```

Обратите внимание, что по умолчанию в легенде игнорируются все элементы, в которых не установлен атрибут `label`.

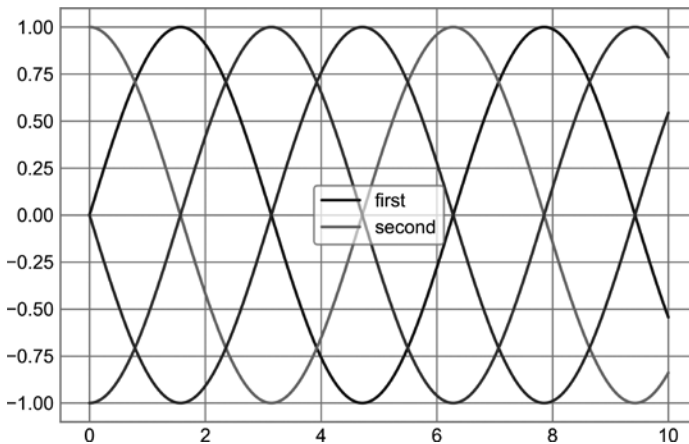


Рис. 29.6. Альтернативный способ настройки содержимого легенды

Задание легенды для точек разного размера

Иногда возможностей легенды по умолчанию недостаточно для описания графика. Допустим, на графике используются точки разного размера для представления определенных признаков данных, и было бы желательно отразить это в легенде. Вот пример, отражающий численность населения городов Калифорнии с помощью размера точек. Нам нужна легенда со шкалой размеров точек, и мы создадим ее путем вывода маркированных данных без самих меток (рис. 29.7):

```
In [9]: # Раскомментируйте следующие строки, чтобы скачать данные
        # url = ('https://raw.githubusercontent.com/jakevdp/
        #        PythonDataScienceHandbook/''master/notebooks/data/
        #        california_cities.csv')
        # !cd data && curl -O {url}

In [10]: import pandas as pd
         cities = pd.read_csv('data/california_cities.csv')

         # Извлекаем интересующие нас данные
         lat, lon = cities['latd'], cities['longd']
         population, area = cities['population_total'], cities['area_total_km2']

         # Распределяем точки по карте,
         # задавая размеры и цвета, но без меток
         plt.scatter(lon, lat, label=None,
                     c=np.log10(population), cmap='viridis',
                     s=area, linewidth=0, alpha=0.5)
         plt.axis('equal')
         plt.xlabel('longitude')
         plt.ylabel('latitude')
         plt.colorbar(label='log$_{10}$(population)')
         plt.clim(3, 7)

         # Создаем легенду:
         # выводим на график пустые списки с нужным размером и меткой
         for area in [100, 300, 500]:
             plt.scatter([], [], c='k', alpha=0.3, s=area,
                         label=str(area) + ' km$^2$')
         plt.legend(scatterpoints=1, frameon=False, labelspacing=1,
                    title='City Area')

         # Города Калифорнии: местоположение и численность населения
         plt.title('California Cities: Area and Population');
```

Легенда всегда относится к какому-либо находящемуся на графике объекту, поэтому, если нам нужно отобразить объект конкретного вида, необходимо сначала его нарисовать на графике. В данном случае нужных нам объектов (кругов серого цвета) на графике нет, поэтому идем на хитрость и выводим на график пустые списки. Обратите внимание, что в легенде перечислены только те элементы графика, для которых задан атрибут `label`.

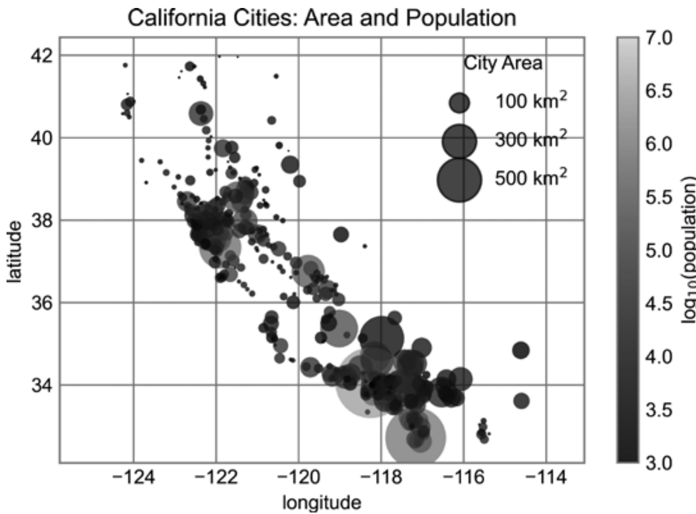


Рис. 29.7. Местоположение, размер и численность населения городов штата Калифорния

Посредством вывода на график пустых списков мы создали маркированные объекты, которые затем включаются в легенду. Теперь легенда дает нам полезную информацию. Эту стратегию можно использовать для создания более сложных визуализаций.

Отображение нескольких легенд

Иногда требуется добавить в график несколько легенд. К сожалению, библиотека Matplotlib не предлагает ничего, что упростило бы эту задачу: используя стандартный интерфейс `legend`, можно создать только одну легенду для всего графика. Если попытаться создать вторую легенду вызовом `plt.legend` и `ax.legend`, то вторая легенда просто перекроет первую. Решить эту проблему можно, создав для легенды новый слой рисунка (экземпляр класса `Artist` — базового класса в Matplotlib, используемого для представления визуальных атрибутов), и затем с помощью низкоуровневого метода `ax.add_artist` вручную добавить этот слой в график (рис. 29.8):

```
In [11]: fig, ax = plt.subplots()

         lines = []
         styles = ['-', '--', '-.', ':']
         x = np.linspace(0, 10, 1000)

         for i in range(4):
             lines += ax.plot(x, np.sin(x - i * np.pi / 2),
                              styles[i], color='black')
         ax.axis('equal')
```

```
# Задаем линии и метки первой легенды
ax.legend(lines[:2], ['line A', 'line B'], # Линия A, Линия B
          loc='upper right')

# Создаем вторую легенду и добавляем новый слой вручную
from matplotlib.legend import Legend
leg = Legend(ax, lines[2:], ['line C', 'line D'], # Линия C, Линия D
             loc='lower right')
ax.add_artist(leg);
```

Мы мельком рассмотрели низкоуровневые объекты слоев рисунка, из которых состоит любой график, создаваемый библиотекой Matplotlib. Если заглянуть в исходный код метода `ax.legend` (напомню, что сделать это можно в блокноте Jupyter с помощью команды `ax.legend??`), то можно увидеть, что эта функция просто создает слой `Legend`, сохраняет его в атрибуте `legend_` и затем добавляет в рисунок при отрисовке графика.

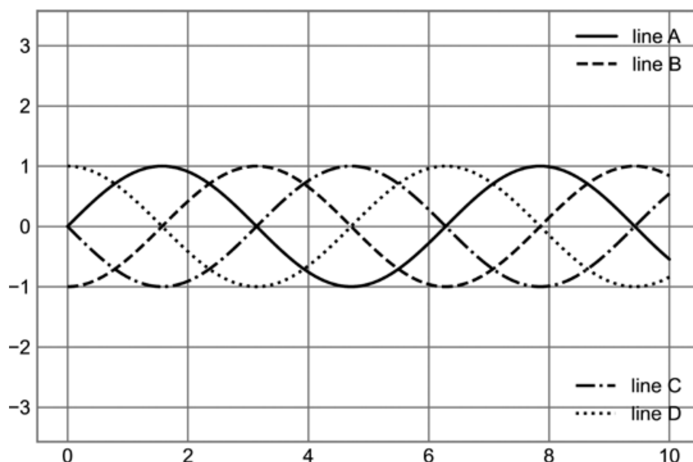


Рис. 29.8. Разделенная на части легенда

ГЛАВА 30

Настройка цветовых шкал

Легенды графиков отображают соответствие дискретных меток дискретным точкам. В случае непрерывных меток, базирующихся на цвете точек, линий или областей, отлично подойдет такой инструмент, как цветовая шкала. Библиотека Matplotlib отображает цветовую шкалу как отдельную систему координат, предоставляющую ключ к значениям цветов на графике. Начнем с настройки блокнота для построения графиков и импорта необходимых функций:

```
In [1]: import matplotlib.pyplot as plt
        plt.style.use('seaborn-white')

In [2]: %matplotlib inline
        import numpy as np
```

Простейшую цветовую шкалу можно создать с помощью функции `plt.colorbar` (рис. 30.1):

```
In [3]: x = np.linspace(0, 10, 1000)
        I = np.sin(x) * np.cos(x[:, np.newaxis])

        plt.imshow(I)
        plt.colorbar();
```

Далее мы рассмотрим несколько идей по настройке цветовой шкалы и эффективному ее использованию в разных ситуациях.

Настройка цветовой шкалы

Задать карту цветов можно с помощью аргумента `cmap` функции создания визуализации (рис. 30.2):

```
In [4]: plt.imshow(I, cmap='Blues');
```

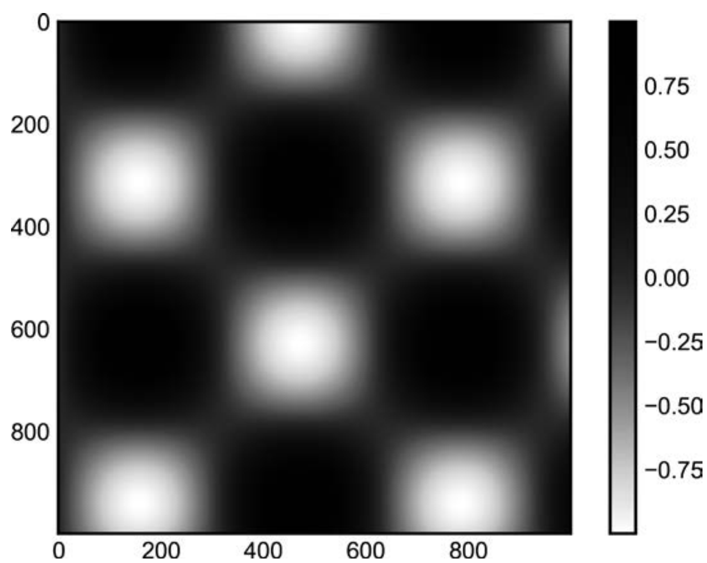


Рис. 30.1. Простая легенда с цветовой шкалой

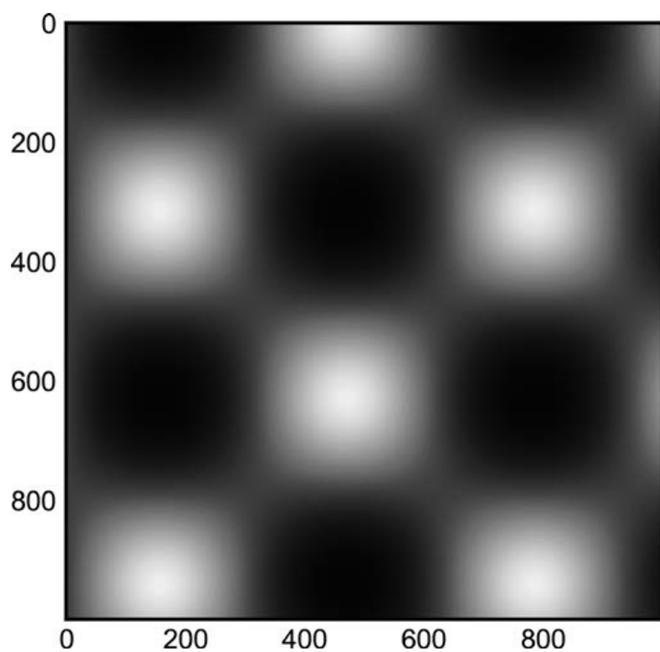


Рис. 30.2. Карта цветов на основе оттенков синего

Все доступные для использования карты цветов определены в пространстве имен `plt.cm`. Получить полный список доступных вариантов можно с помощью автодополнения клавишей Tab в оболочке IPython:

```
plt.cm.<TAB>
```

Но *возможность* выбора карты цветов — лишь первый шаг, гораздо важнее сделать *правильный* выбор из имеющихся вариантов! Как оказывается, сделать такой выбор часто намного сложнее, чем можно было бы ожидать.

Выбор карты цветов

Всестороннее рассмотрение вопроса выбора цветов для визуализации выходит за рамки этой книги, но желающие углубиться в этот вопрос могут прочитать статью *Ten Simple Rules for Better Figures* («Десять простых правил для улучшения изображений»; https://oreil.ly/g4GLV) Николаса Ружье, Майкла Дроттбума и Филипа Борна. Онлайн-документация библиотеки Matplotlib тоже содержит интересную информацию по вопросу выбора карты цветов (https://oreil.ly/Ll1ir).

Вам следует знать, что существуют три категории карт цветов:

- *последовательные карты цветов*, состоящие из одной непрерывной последовательности цветов (например, `binary` или `viridis`);
- *дивергентные карты цветов*, обычно содержащие два хорошо различимых цвета, отражающих положительные и отрицательные отклонения от среднего значения (например, `RdBu` или `PuOr`);
- *качественные карты цветов*, в которых цвета смешиваются без какого-либо четкого порядка (например, `rainbow` или `jet`).

Карта цветов `jet`, использовавшаяся по умолчанию в библиотеке Matplotlib до версии 2.0, — это пример качественной карты цветов. Ее выбор как карты цветов по умолчанию был весьма неудачен, потому что качественные карты цветов плохо подходят для представления количественных данных: обычно они не отражают равномерного роста яркости при продвижении по шкале.

Продемонстрировать это можно, преобразовав цветовую шкалу `jet` в черно-белое представление (рис. 30.3):

```
In [5]: from matplotlib.colors import LinearSegmentedColormap

        def grayscale_cmap(cmap):
            """Возвращает черно-белую версию заданной карты цветов"""
```

```
        cmap = plt.cm.get_cmap(cmap)
        colors = cmap(np.arange(cmap.N))

        # Преобразуем RGBA в воспринимаемую глазом черно-белую светимость
        # ср. http://alienryderflex.com/hsp.html
        RGB_weight = [0.299, 0.587, 0.114]
        luminance = np.sqrt(np.dot(colors[:, :3] ** 2, RGB_weight))
        colors[:, :3] = luminance[:, np.newaxis]

        return LinearSegmentedColormap.from_list(
            cmap.name + "_gray", colors, cmap.N)

    def view_colormap(cmap):
        """Рисует карту цветов и ее черно-белый эквивалент"""
        cmap = plt.cm.get_cmap(cmap)
        colors = cmap(np.arange(cmap.N))

        cmap = grayscale_cmap(cmap)
        grayscale = cmap(np.arange(cmap.N))

        fig, ax = plt.subplots(2, figsize=(6, 2),
                               subplot_kw=dict(xticks=[], yticks=[]))
        ax[0].imshow([colors], extent=[0, 10, 0, 1])
        ax[1].imshow([grayscale], extent=[0, 10, 0, 1])

In [6]: view_colormap('jet')
```

Рис. 30.3. Карта цветов jet и ее неравномерная шкала светимости

```
In [7]: view_colormap('viridis')
```

Обратите внимание на яркие полосы в ахроматическом изображении. Даже в полном цвете эта неравномерная яркость означает, что определенные части цветового диапазона будут притягивать больше внимания, что может привести к акцентированию несущественных частей набора данных. Лучше применять такие карты цветов, как `viridis` (используется по умолчанию, начиная с версии Matplotlib 2.0), специально сконструированные для равномерного изменения яркости по диапазону. Они не только хорошо согласуются с нашим цветовым восприятием, но и адекватно преобразуются для целей черно-белой печати (рис. 30.4).

Рис. 30.4. Карта цветов viridis и ее равномерная шкала светимости

В других случаях, например для отображения положительных и отрицательных отклонений от среднего значения, удобнее такие двуцветные карты, как `RdBu` (сокращенно от Red-Blue — «красно-синяя»). Однако, как показано на рис. 30.5, такая информация будет потеряна при переходе к черно-белому эквиваленту!

```
In [8]: view_colormap('RdBu')
```

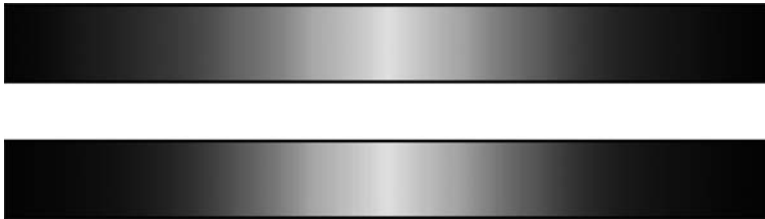


Рис. 30.5. Карта цветов RdBu (красно-синяя) и ее светимость

Далее мы рассмотрим примеры использования некоторых из этих карт цветов.

В библиотеке Matplotlib определено множество разных карт цветов. Получить полный список этих карт можно, исследовав содержимое модуля `plt.cm` с помощью оболочки IPython. Более принципиальный подход к использованию цветов в языке Python можно найти в инструментах и документации библиотеки Seaborn (см. главу 36).

Ограничение и расширение карты цветов

Библиотека Matplotlib позволяет настраивать цветовые шкалы в весьма широких пределах. Сами по себе цветовые шкалы — это просто экземпляры класса `plt.Axes`, поэтому к ним применимы все уже изученные нами трюки, связанные с форматированием осей координат и делений на них. Цветовые шкалы обладают достаточной гибкостью: например, позволяют сузить границы диапазона цветов, обозначив

выходящие за пределы этого диапазона значения с помощью треугольных стрелок вверху и внизу настройкой значения свойства `extend`. Это может оказаться удобно, например, при выводе зашумленного изображения (рис. 30.6):

```
In [9]: # создаем шум в 1 % пикселов изображения
        speckles = (np.random.random(I.shape) < 0.01)
        I[speckles] = np.random.normal(0, 3, np.count_nonzero(speckles))
        plt.figure(figsize=(10, 3.5))

        plt.subplot(1, 2, 1)
        plt.imshow(I, cmap='RdBu')
        plt.colorbar()

        plt.subplot(1, 2, 2)
        plt.imshow(I, cmap='RdBu')
        plt.colorbar(extend='both')
        plt.clim(-1, 1)
```

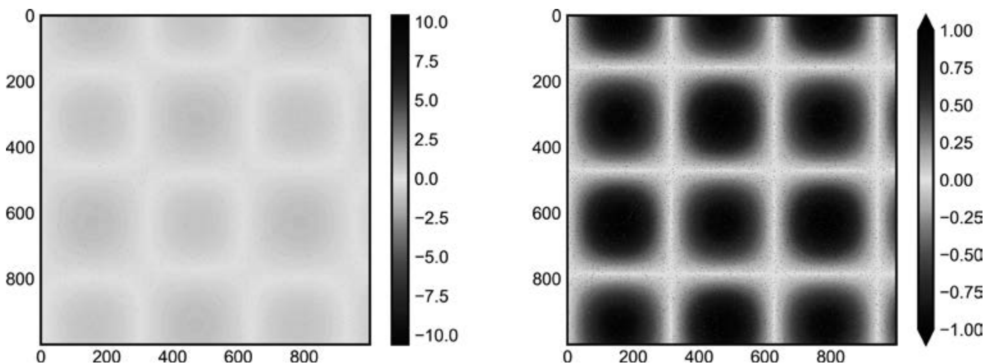


Рис. 30.6. Расширение карты цветов

Обратите внимание, что на рисунке слева зашумленные пикселы влияют на пределы диапазона цветов, по этой причине диапазон шума делает совершенно неразличимыми интересующие нас закономерности. На правом рисунке мы задаем пределы диапазона цветов вручную и добавляем стрелки, указывающие на значения, выходящие за эти пределы. В результате получаем намного более качественную визуализацию данных.

Дискретные цветовые шкалы

Карты цветов по умолчанию непрерывны, но иногда нужно обеспечить отображение дискретных значений. Простейший способ этого добиться — воспользоваться функцией `plt.cm.get_cmap()`, передав ей имя карты цветов с требуемым количеством диапазонов (рис. 30.7):

```
In [10]: plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))
         plt.colorbar(extend='both')
         plt.clim(-1, 1);
```

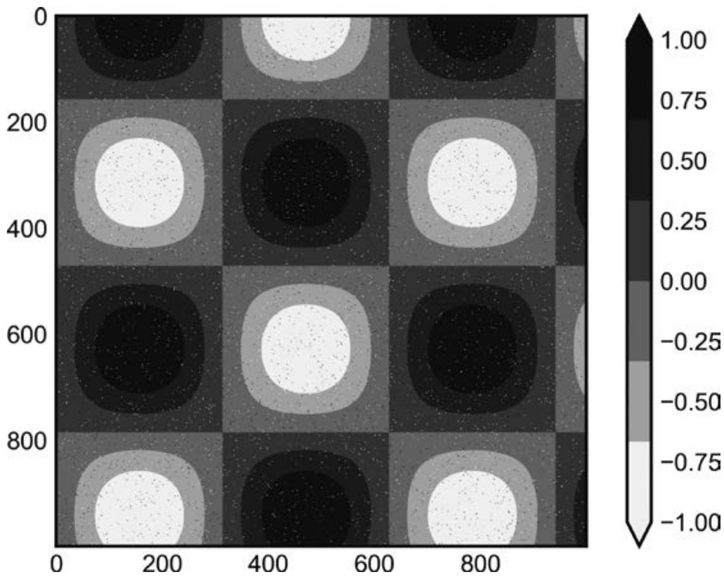


Рис. 30.7. Дискретизированная карта цветов

Дискретный вариант можно использовать точно так же, как и любую другую карту цветов.

Пример: рукописные цифры

В качестве примера рассмотрим интересную визуализацию данных с рукописными цифрами. Набор данных включен в библиотеку Scikit-Learn и состоит почти из 2000 миниатюр размером 8 × 8 с рукописными цифрами.

Начнем со скачивания данных и вывода нескольких примеров изображений с помощью функции `plt.imshow()` (рис. 30.8):

```
In [11]: # Загружаем изображения цифр от 0 до 5 и выводим некоторые из них
         from sklearn.datasets import load_digits
         digits = load_digits(n_class=6)

         fig, ax = plt.subplots(8, 8, figsize=(6, 6))
         for i, axi in enumerate(ax.flat):
             axi.imshow(digits.images[i], cmap='binary')
             axi.set(xticks=[], yticks=[])
```

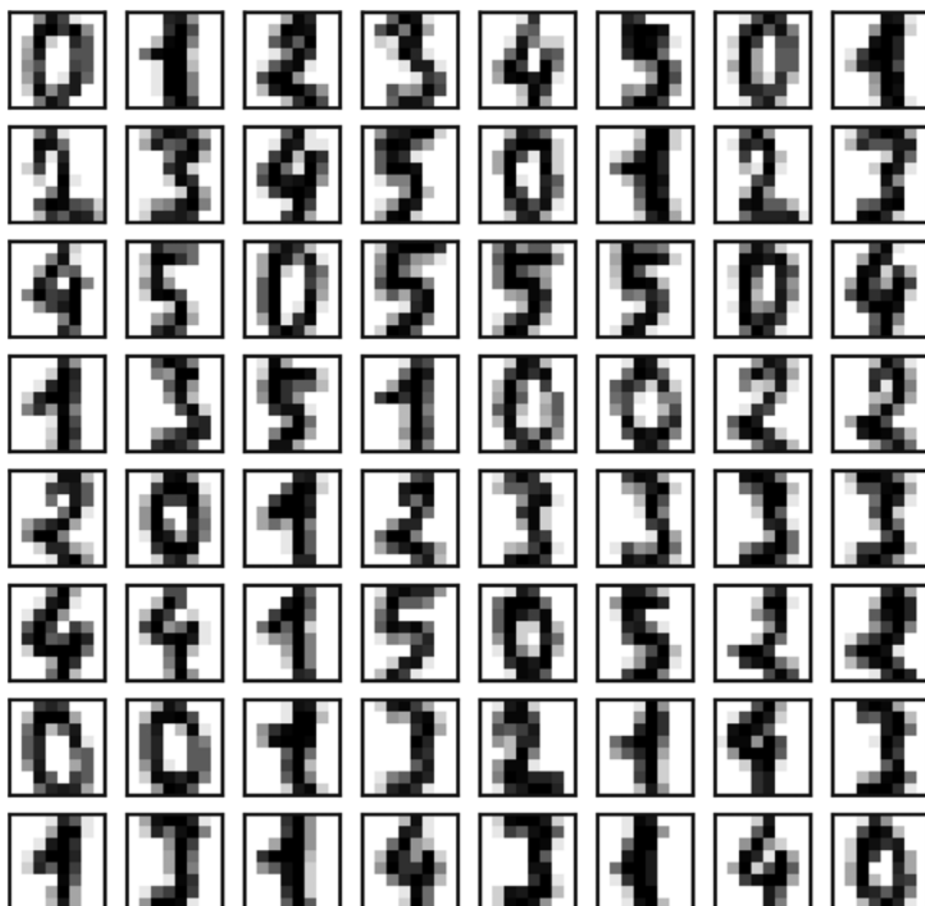


Рис. 30.8. Некоторые примеры изображений рукописных цифр

Поскольку каждая цифра определяется оттенками ее 64 пикселов, можно считать ее точкой в 64-мерном пространстве: каждое измерение отражает яркость одного пиксела. Однако визуализация зависимостей в пространстве с таким большим числом измерений — исключительно непростая задача. Одно из возможных решений — воспользоваться каким-либо из методов *понижения размерности* (dimensionality reduction), например обучением на базе многообразий (manifold learning) с целью снижения размерности данных при сохранении интересующих нас зависимостей. Понижение размерности — пример машинного обучения без учителя (unsupervised machine learning). Мы обсудим его подробнее в главе 37.

Рассмотрим отображение с помощью обучения на базе многообразий наших данных в двумерное пространство (см. подробности в главе 46):

```
In [12]: # Отображаем цифры в двумерное пространство с помощью функции Isomap
         from sklearn.manifold import Isomap
         iso = Isomap(n_components=2, n_neighbors=15)
         projection = iso.fit_transform(digits.data)
```

Воспользуемся нашей дискретной цветовой картой для просмотра результатов, задав параметры `ticks` и `clim` для улучшения внешнего вида итогового изображения (рис. 30.9):

```
In [13]: # Визуализируем результаты
            plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,
                        c=digits.target, cmap=plt.cm.get_cmap('plasma', 6))
            plt.colorbar(ticks=range(6), label='digit value')
            plt.clim(-0.5, 5.5)
```

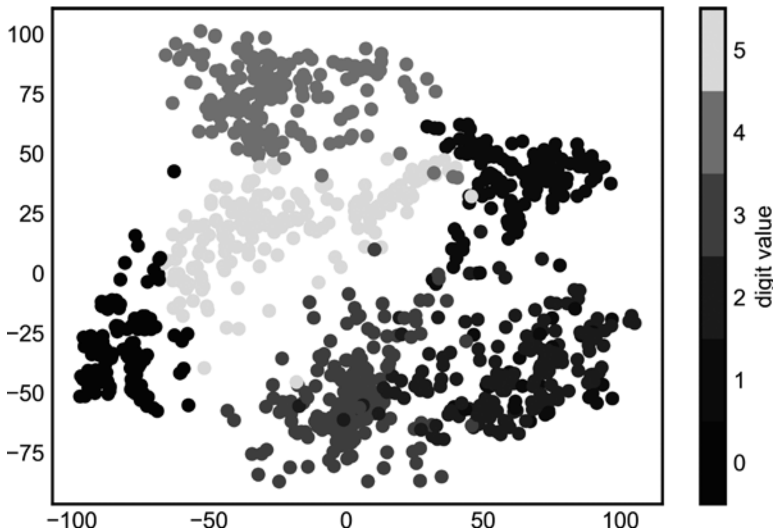


Рис. 30.9. Многообразие рукописных цифр

Это отображение также дает полезную информацию о зависимостях внутри набора данных. Например, диапазоны цифр 2 и 3 в проекции практически перекрываются, то есть некоторые рукописные двойки и тройки отличить друг от друга довольно непросто и, следовательно, выше вероятность, что автоматический алгоритм классификации будет их путать. Другие значения, например 0 и 1, разделены более четко, значит, вероятность путаницы намного меньше.

Мы вернемся к обучению на базе многообразий и классификации цифр в части V.

ГЛАВА 31

Множественные субграфики

Иногда удобно сравнить различные представления данных, разместив их бок о бок. В библиотеке Matplotlib на такой случай предусмотрено понятие *субграфиков* (subplots), когда несколько систем координат располагаются на одном рисунке. Эти субграфики можно разместить как вставки, сетки графиков или более сложным способом. В данном разделе мы рассмотрим четыре функции для создания субграфиков. Начнем с настройки блокнота для построения графиков и импорта необходимых функций:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-white')
        import numpy as np
```

plt.axes: создание субграфиков вручную

Функция `plt.axes` предлагает простейший метод создания систем координат. По умолчанию она создает стандартный объект системы координат, заполняющий весь рисунок. `plt.axes` также принимает необязательный аргумент — список из четырех чисел в системе координат рисунка (`[низ, левый угол, ширина, высота]`), отсчет которых начинается с 0 в нижнем левом и заканчивается 1 в верхнем правом углу рисунка.

Например, мы можем создать «вставную» систему координат в верхнем правом углу другой системы координат, задав координаты *x* и *y* ее местоположения рав-

ными 0,65 (то есть начинающимися на 65 % ширины и 65 % высоты рисунка), а ее размеры по осям *X* и *Y* равными 0,2 (то есть размер этой системы координат составляет 20 % ширины и 20 % высоты рисунка). Результат показан на рис. 31.1:

```
In [2]: ax1 = plt.axes()  # оси координат созданные по умолчанию
        ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```

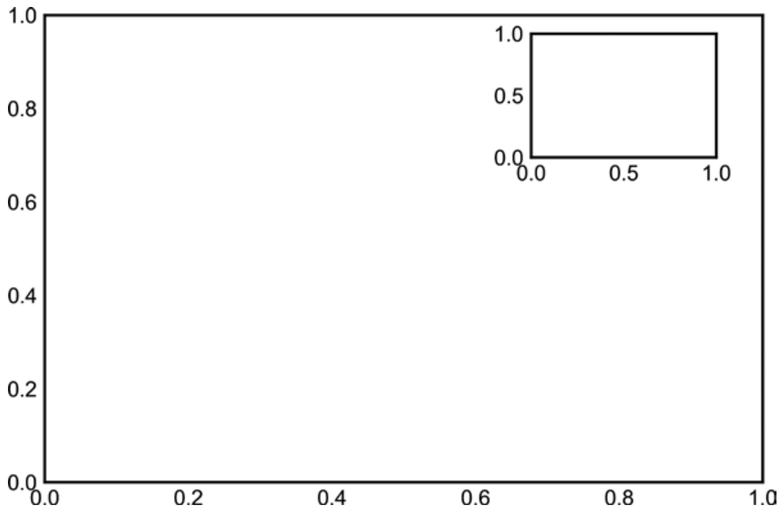


Рис. 31.1. Пример вставной системы координат

Аналог этой команды в объектно-ориентированном интерфейсе — функция `fig.add_axes`. Воспользуемся ею для создания двух расположенных друг над другом систем координат (рис. 31.2):

```
In [3]: fig = plt.figure()
        ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],
                           xticklabels=[], ylim=(-1.2, 1.2))
        ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4],
                           ylim=(-1.2, 1.2))

        x = np.linspace(0, 10)
        ax1.plot(np.sin(x))
        ax2.plot(np.cos(x));
```

Мы получили две соприкасающиеся системы координат (верхняя — без делений): низ верхней области (находящейся в позиции 0,5) соответствует верху нижней области (находящейся в позиции 0,1 + 0,4).

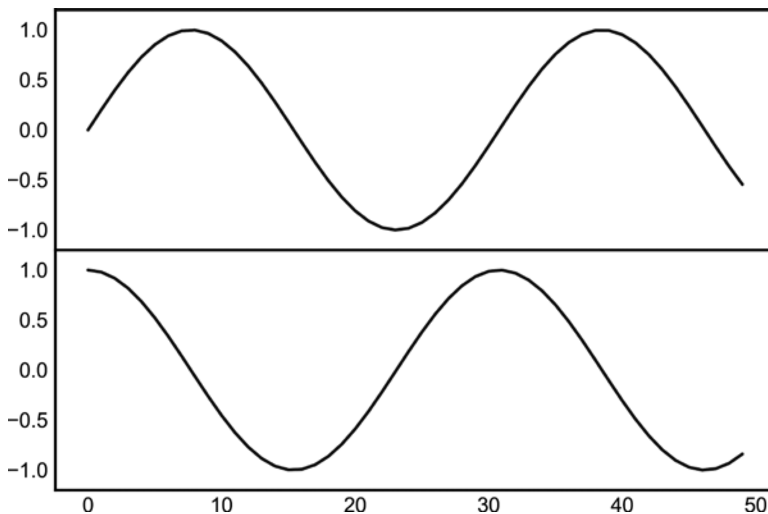


Рис. 31.2. Пример размещения систем координат друг над другом

plt.subplot: простые сетки субграфиков

Субграфики, выровненные по столбцам или строкам, бывают нужны достаточно часто, поэтому в библиотеку Matplotlib было включено несколько удобных утилит, облегчающих их создание. Самая низкоуровневая из них — функция `plt.subplot`. Она создает отдельный субграфик внутри сетки. Эта команда принимает три целочисленных аргумента — количество строк, количество столбцов и индекс создаваемого по такой схеме графика, отсчет которого начинается в верхнем левом углу и заканчивается в правом нижнем (рис. 31.3):

```
In [4]: for i in range(1, 7):
            plt.subplot(2, 3, i)
            plt.text(0.5, 0.5, str((2, 3, i)),
                     fontsize=18, ha='center')
```

Настроить размеры полей между субграфиками можно с помощью `plt.subplots_adjust`. Следующий код (результат которого показан на рис. 31.4) использует эквивалентную объектно-ориентированную команду `fig.add_subplot`:

```
In [5]: fig = plt.figure()
        fig.subplots_adjust(hspace=0.4, wspace=0.4)
        for i in range(1, 7):
            ax = fig.add_subplot(2, 3, i)
            ax.text(0.5, 0.5, str((2, 3, i)),
                   fontsize=18, ha='center')
```

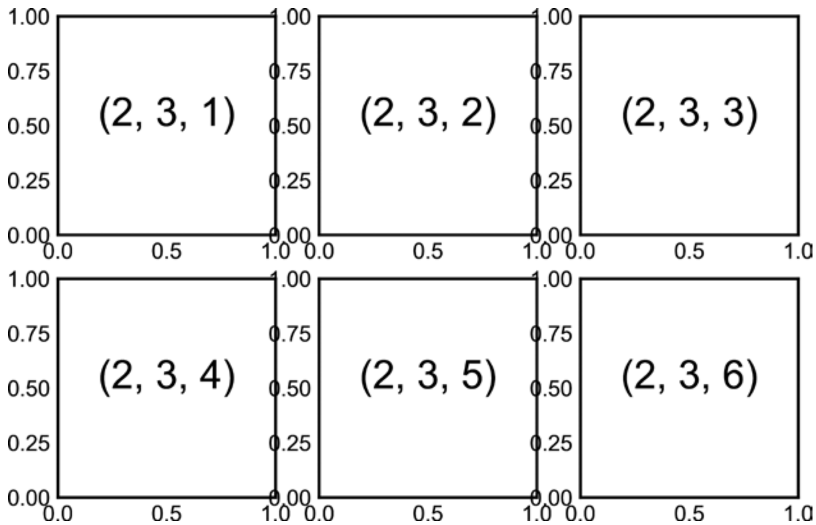


Рис. 31.3. Пример использования функции plt.subplot

Здесь мы использовали аргументы `hspace` и `wspace` функции `plt.subplots_adjust`, позволяющие задать поля по высоте и ширине рисунка в единицах размеров субграфика (в данном случае поля составляют 40 % от ширины и высоты субграфика).

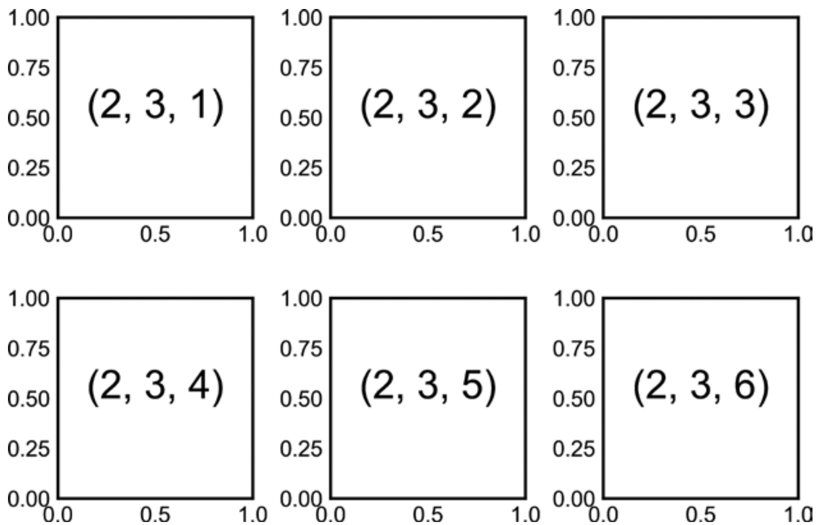


Рис. 31.4. plt.subplot с выровненными полями

plt.subplots: создание всей сетки за один раз

Только что описанный подход может оказаться довольно трудоемким при создании большой сетки субграфиков, особенно если нужно скрыть метки осей X и Y на внутренних графиках. В этом случае удобнее использовать функцию `plt.subplots` (обратите внимание на букву `s` в конце `subplots`). Вместо отдельного субграфика эта функция создает целую сетку субграфиков одной строкой кода и возвращает их в массиве NumPy. Она принимает в аргументах количество строк и столбцов, а также необязательные именованные аргументы `sharex` и `sharey`, позволяющие задавать связи между различными системами координат.

В следующем примере создается сетка 2 × 3 субграфиков, в которой у всех систем координат в одной строке одинаковая шкала по оси Y, а у всех систем координат в одном столбце — одинаковая шкала по оси X (рис. 31.5):

```
In [6]: fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```

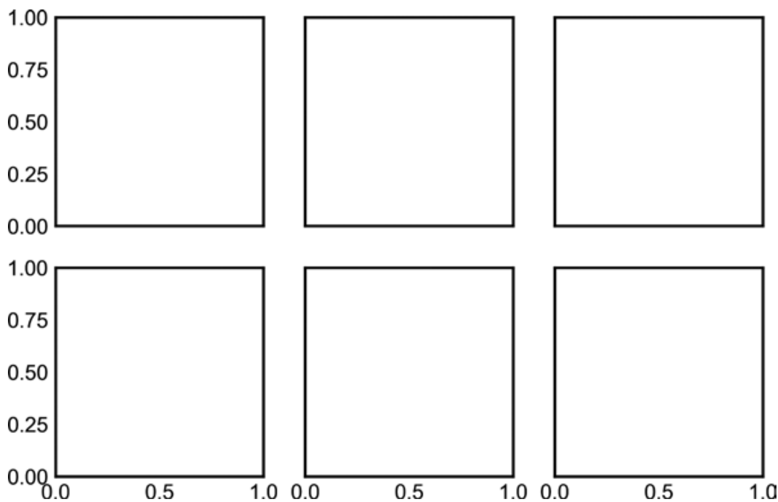


Рис. 31.5. Общие оси координат X и Y при использовании plt.subplots

Обратите внимание, что передача аргументов `sharex` и `sharey` приводит к автоматическому удалению внутренних меток в сетке для освобождения пространства графика. Итоговая сетка систем координат возвращается в массиве NumPy, что дает возможность легко ссылаться на требуемую систему координат с помощью обычной для массивов индексации (рис. 31.6):

```
In [7]: # Системы координат располагаются в двумерном массиве [строка, столбец]
        for i in range(2):
            for j in range(3):
```

```
        ax[i, j].text(0.5, 0.5, str((i, j)),
                      fontsize=18, ha='center')
fig
```

В отличие от `plt.subplot`, функция `plt.subplots` намного лучше согласуется с принятой в языке Python индексацией, начинающейся с 0, тогда как `plt.subplot` использует индексацию в стиле MATLAB, начинающуюся с 1.

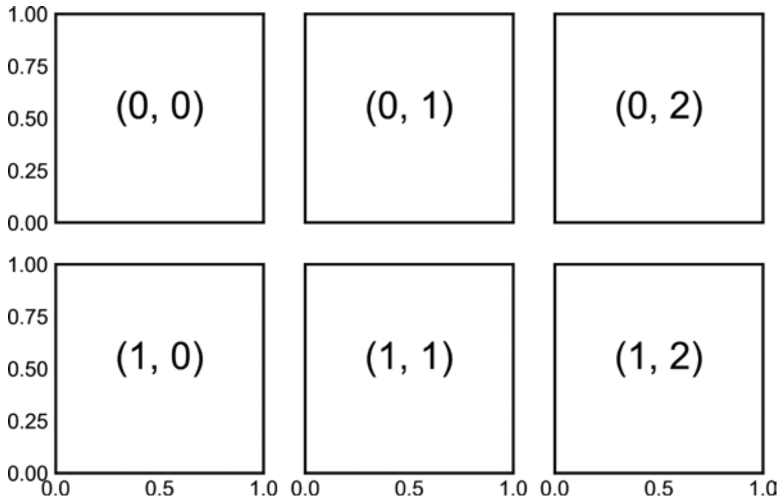


Рис. 31.6. Нумерация графиков в сетке субграфиков

plt.GridSpec: более сложные конфигурации

При выходе за пределы обычной сетки графиков к субграфикам, занимающим много строк и столбцов, наилучшим инструментом считается `plt.GridSpec`. Сам по себе объект `plt.GridSpec` не создает графиков, это просто удобный интерфейс, понятный команде `plt.subplot`. Например, создание объекта `GridSpec`, представляющего сетку из двух строк и трех столбцов с заданными значениями ширины и высоты, будет выглядеть так:

```
In [8]: grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
```

Создав объект, можно задать местоположение и размеры субграфиков с помощью обычного синтаксиса срезов языка Python (рис. 31.7):

```
In [9]: plt.subplot(grid[0, 0])
        plt.subplot(grid[0, 1:])
        plt.subplot(grid[1, :2])
        plt.subplot(grid[1, 2]);
```

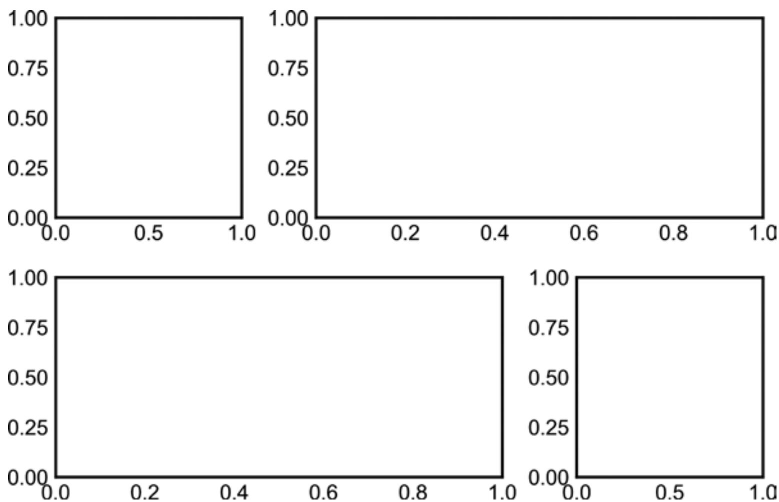


Рис. 31.7. Создание субграфиков неодинаковой формы с помощью plt.GridSpec

Подобное гибкое формирование сетки находит множество различных применений. Я часто использую этот прием для создания графиков гистограмм с несколькими системами координат (рис. 31.8):

```
In [10]: # Генерируем нормально распределенные данные
         mean = [0, 0]
         cov = [[1, 1], [1, 2]]
         rng = np.random.default_rng(1701)
         x, y = rng.multivariate_normal(mean, cov, 3000).T

         # Задаем системы координат с помощью GridSpec
         fig = plt.figure(figsize=(6, 6))
         grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)
         main_ax = fig.add_subplot(grid[:-1, 1:])
         y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=main_ax)
         x_hist = fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=main_ax)

         # Распределяем точки по основной системе координат
         main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)

         # Рисуем гистограммы на дополнительных системах координат
         x_hist.hist(x, 40, histtype='stepfilled',
                     orientation='vertical', color='gray')
         x_hist.invert_yaxis()

         y_hist.hist(y, 40, histtype='stepfilled',
                     orientation='horizontal', color='gray')
         y_hist.invert_xaxis()
```

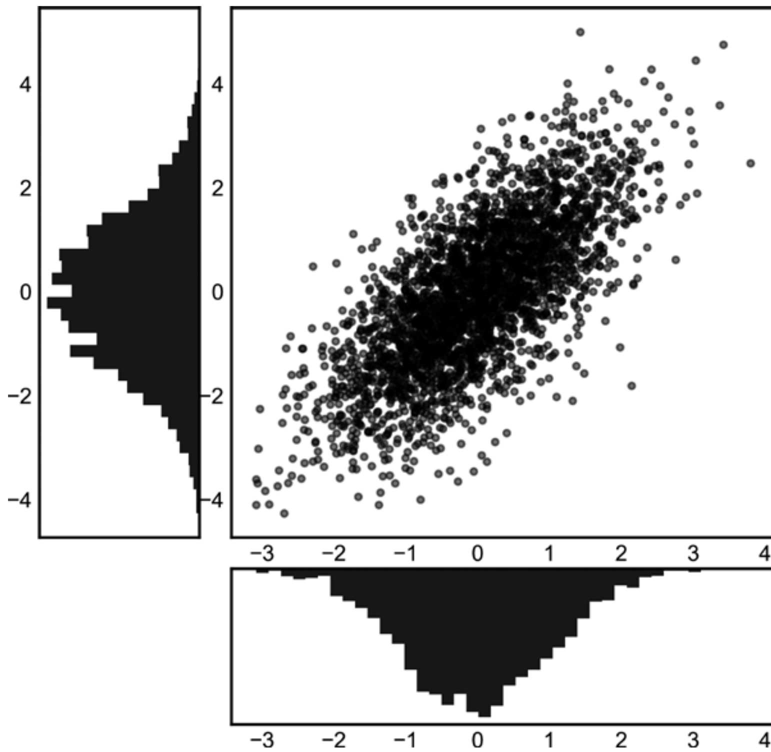


Рис. 31.8. Визуализация многомерных распределений с помощью plt.GridSpec

Такой способ вывода распределения на отдельных графиках по бокам настолько распространен, что в пакете Seaborn для этой цели предусмотрен отдельный API. Подробную информацию см. в главе 36.

ГЛАВА 32

Текст и поясняющие надписи

Хорошая визуализация должна рассказывать читателю историю. В некоторых случаях это можно сделать только визуальными средствами, без дополнительного текста, но иногда небольшие текстовые подсказки и метки необходимы. Вероятно, простейший вид поясняющих надписей — метки на осях координат и их названия, но имеющиеся возможности гораздо шире. Рассмотрим на примере каких-нибудь данных, как можно их визуализировать и добавить поясняющие надписи, чтобы донести до читателя полезную информацию. Начнем с настройки блокнота для построения графиков и импорта необходимых функций:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import matplotlib as mpl
        plt.style.use('seaborn-whitegrid')
        import numpy as np
        import pandas as pd
```

Вернемся к данным, с которыми мы работали ранее в разделе «Пример: данные о рождаемости» главы 21, где мы сгенерировали график среднего количества рождений детей в зависимости от дня календарного года. Начнем с той же процедуры очистки данных, которую мы использовали ранее, и построим график результатов (рис. 32.1):

```
In [2]: # команды для скачивания набора данных:
        # !cd data && curl -O \
        #   https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/
        #   births.csv

In [3]: from datetime import datetime

        births = pd.read_csv('data/births.csv')

        quartiles = np.percentile(births['births'], [25, 50, 75]) # рождений

        mu, sig = quartiles[1], 0.74 * (quartiles[2] - quartiles[0])
        births = births.query('(births > @mu - 5 * @sig) &
                               (births < @mu + 5 * @sig)')
```

```
        births['day'] = births['day'].astype(int)

        births.index = pd.to_datetime(10000 * births.year +
                                      100 * births.month +
                                      births.day, format='%Y%m%d')
        births_by_date = births.pivot_table('births',
                                            [births.index.month, births.index.day])
        births_by_date.index = [datetime(2012, month, day)
                                for (month, day) in births_by_date.index]

In [4]: fig, ax = plt.subplots(figsize=(12, 4))
        births_by_date.plot(ax=ax);
```

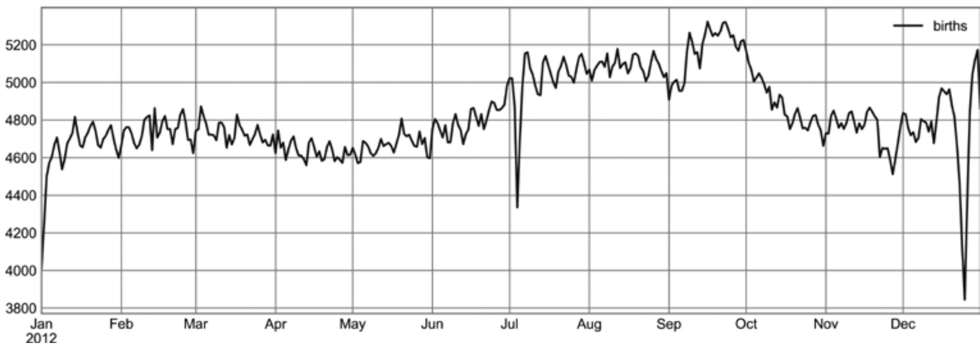


Рис. 32.1. Среднее ежедневное количество новорожденных в зависимости от даты[1]

При визуализации подобных данных часто полезно снабдить элементы графика пояснениями для привлечения внимания читателя. Это можно сделать вручную с помощью команды `plt.text`/`ax.text`, которая поместит текст в месте, соответствующем конкретным значениям координат *x*/*y* (рис. 32.2):

```
In [5]: fig, ax = plt.subplots(figsize=(12, 4))
        births_by_date.plot(ax=ax)

        # Добавляем метки на график
        style = dict(size=10, color='gray')

        ax.text('2012-1-1', 3950, "New Year's Day", **style)
        ax.text('2012-7-4', 4250, "Independence Day", ha='center', **style)
        ax.text('2012-9-4', 4850, "Labor Day", ha='center', **style)
        ax.text('2012-10-31', 4600, "Halloween", ha='right', **style)
        ax.text('2012-11-25', 4450, "Thanksgiving", ha='center', **style)
        ax.text('2012-12-25', 3850, "Christmas ", ha='right', **style)
```

[1] Полноразмерную версию рисунков из этой главы можно найти на GitHub (https://oreil.ly/PDSH_GitHub).

```
# Добавляем метки для осей координат
ax.set(title='USA births by day of year (1969-1988)',
       ylabel='average daily births')
# Ежедневное количество новорожденных в зависимости от даты, США (1969-1988)

# Размечаем ось X центрированными метками для месяцев
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));
```

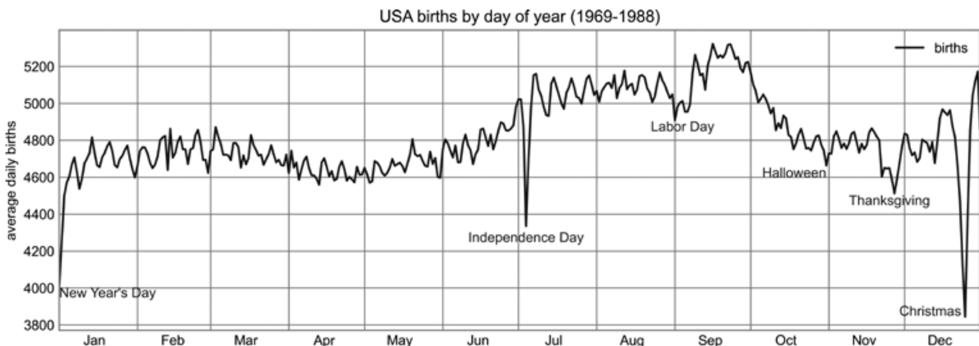


Рис. 32.2. Ежедневное количество рождаемых детей в зависимости от даты, с комментариями

Метод `ax.text` принимает координату *x*, координату *y*, строковое значение и необязательные именованные аргументы, задающие цвет, размер, стиль, выравнивание и другие свойства текста. В данном случае мы передали значения `ha='right'` и `ha='center'`, где `ha` — сокращение от *horizontal alignment* (выравнивание по горизонтали). Дополнительную информацию о доступных параметрах смотрите в docstring для `plt.text` и `mpl.text.Text`.

Преобразования и координаты текста

В предыдущем примере мы привязали текстовые пояснения к конкретным значениям данных. Иногда бывает удобнее привязать текст к координатам на осях рисунка независимо от данных. В библиотеке Matplotlib это осуществляется путем модификации *преобразования* (transform).

Matplotlib может использовать несколько разных систем координат: точка данных с координатами $(x, y) = (1, 1)$ соответствует определенному местоположению на осях или рисунке, которое, в свою очередь, соотносится с пикселом на экране. С математической точки зрения подобные преобразования несложны, и сама библиотека Matplotlib имеет для их выполнения хорошо продуманный набор инструментов (они находятся в подмодуле `matplotlib.transforms`).

Среднестатистический пользователь редко задумывается о деталях этих преобразований, но если речь идет о размещении текста на рисунке, не помешает иметь о них представление. Существуют три предопределенных преобразования, которые могут оказаться полезными в подобной ситуации:

- `ax.transData` — преобразование из системы координат данных;
- `ax.transAxes` — преобразование из системы координат объекта `Axes` (в единицах размеров рисунка);
- `fig.transFigure` — преобразование из системы координат объекта `Figure` (в единицах размеров рисунка).

Рассмотрим пример вывода текста в различных местах рисунка с помощью этих преобразований (рис. 32.3):

```
In [6]: fig, ax = plt.subplots(facecolor='lightgray')
        ax.axis([0, 10, 0, 10])

        # transform=ax.transData — значение по умолчанию,
        # но мы все равно указываем его
        ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
        ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
        ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure);
```

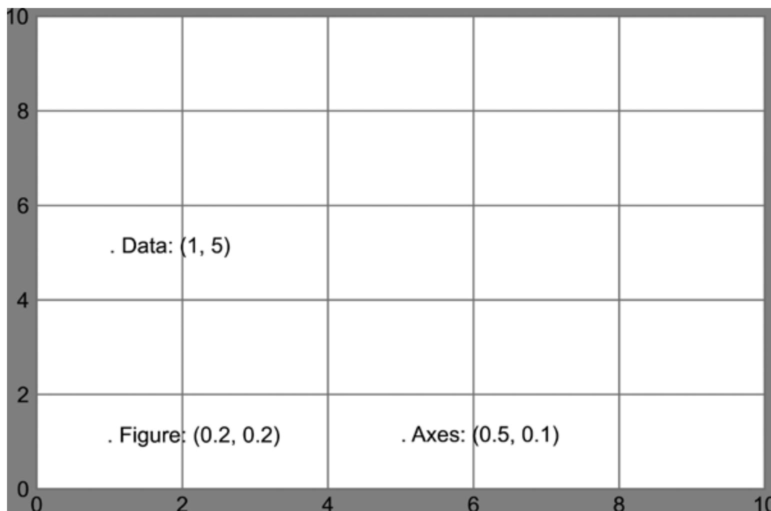


Рис. 32.3. Сравнение различных систем координат, поддерживаемых в Matplotlib

По умолчанию текст выравнивается по базовой линии и левому краю указанных координат, поэтому точка в начале каждой строки приблизительно отмечает заданные координаты.

Координаты `transData` задают обычные координаты данных, соответствующие меткам на осях X и Y. Координаты `transAxes` задают местоположение относительно нижнего левого угла системы координат (здесь — белый прямоугольник) в виде доли от размера системы координат. Координаты `transFigure` схожи с `transAxes`, но задают местоположение относительно нижнего левого угла рисунка (здесь — серый прямоугольник) в виде доли от размера рисунка.

Примечательно, что если поменять пределы осей координат, то это повлияет только на координаты `transData`, а другие останутся неизменными (рис. 32.4):

```
In [7]: ax.set_xlim(0, 2)
        ax.set_ylim(-6, 6)
        fig
```

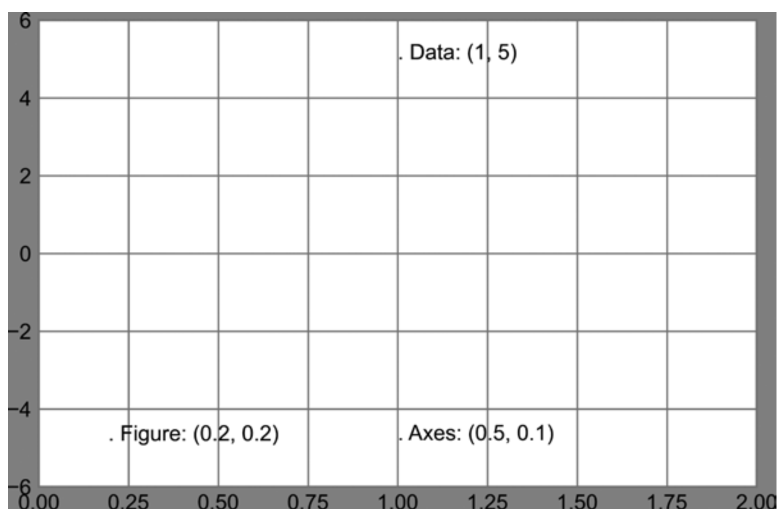


Рис. 32.4. Сравнение различных систем координат, поддерживаемых в Matplotlib

Продемонстрировать это поведение более наглядно можно путем интерактивного изменения пределов осей координат. В блокноте этого можно добиться, заменив `%matplotlib inline` на `%matplotlib notebook` и воспользовавшись меню каждого из графиков для работы с ним.

Стрелки и поясняющие надписи

Наряду с метками делений и текстом удобной поясняющей меткой является простая стрелка.

Рисование стрелок в Matplotlib зачастую оказывается более сложной задачей, чем вы могли бы предполагать. Несмотря на существование функции `plt.arrow`,

использовать ее я бы не советовал: она создает стрелки в виде SVG-объектов, которые могут изменяться с изменением соотношения сторон графиков, поэтому результат редко оказывается соответствующим ожиданиям. Вместо этого я предложил бы воспользоваться функцией `plt.annotate`. Она создает текст и стрелку, причем позволяет очень гибко задавать настройки для стрелки.

В следующем примере мы используем функцию `annotate` с несколькими параметрами (рис. 32.5):

```
In [8]: fig, ax = plt.subplots()

        x = np.linspace(0, 20, 1000)
        ax.plot(x, np.cos(x))
        ax.axis('equal')

        # локальный максимум
        ax.annotate('local maximum', xy=(6.28, 1), xytext=(10, 4),
                    arrowprops=dict(facecolor='black', shrink=0.05))

        # локальный минимум
        ax.annotate('local minimum', xy=(5 * np.pi, -1), xytext=(2, -6),
                    arrowprops=dict(arrowstyle="->",
                                    connectionstyle="angle3,angleA=0,angleB=-90"));
```

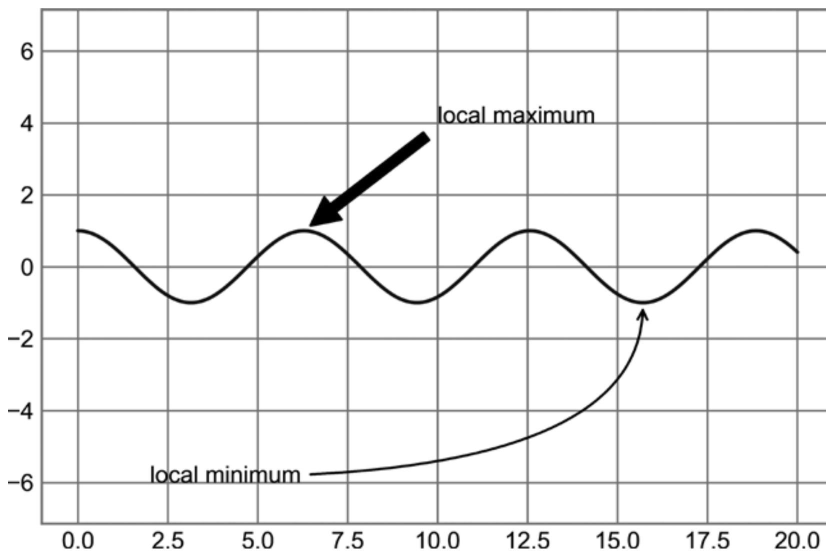


Рис. 32.5. Примеры поясняющих надписей

Стилем стрелки можно управлять с помощью словаря `arrowprops` со множеством параметров. Эти параметры отлично описаны в онлайн-документации библиотеки Matplotlib, поэтому вместо их перечисления я просто покажу несколько

возможностей. Продемонстрируем часть имеющихся параметров на уже знакомом вам графике рождаемости (рис. 32.6):

```
In [9]: fig, ax = plt.subplots(figsize=(12, 4))
        births_by_date.plot(ax=ax)

        # Добавляем на график метки
        ax.annotate("New Year's Day", xy=('2012-1-1', 4100),  xycoords='data',
                    xytext=(50, -30), textcoords='offset points',
                    arrowprops=dict(arrowstyle="->",
                                    connectionstyle="arc3,rad=-0.2"))

        ax.annotate("Independence Day", xy=('2012-7-4', 4250),  xycoords='data',
                    bbox=dict(boxstyle="round", fc="none", ec="gray"),
                    xytext=(10, -40), textcoords='offset points', ha='center',
                    arrowprops=dict(arrowstyle="->"))

        ax.annotate('Labor Day Weekend', xy=('2012-9-4', 4850), xycoords='data',
                    ha='center', xytext=(0, -20), textcoords='offset points')
        ax.annotate('', xy=('2012-9-1', 4850), xytext=('2012-9-7', 4850),
                    xycoords='data', textcoords='data',
                    arrowprops={'arrowstyle': '|-|,widthA=0.2,widthB=0.2', })

        ax.annotate('Halloween', xy=('2012-10-31', 4600),  xycoords='data',
                    xytext=(-80, -40), textcoords='offset points',
                    arrowprops=dict(arrowstyle="fancy",
                                    fc="0.6", ec="none",
                                    connectionstyle="angle3,angleA=0,angleB=-90"))

        ax.annotate('Thanksgiving', xy=('2012-11-25', 4500),  xycoords='data',
                    xytext=(-120, -60), textcoords='offset points',
                    bbox=dict(boxstyle="round4,pad=.5", fc="0.9"),
                    arrowprops=dict(
                        arrowstyle="->",
                        connectionstyle="angle,angleA=0,angleB=80,rad=20"))

        ax.annotate('Christmas', xy=('2012-12-25', 3850),  xycoords='data',
                     xytext=(-30, 0), textcoords='offset points',
                     size=13, ha='right', va="center",
                     bbox=dict(boxstyle="round", alpha=0.1),
                     arrowprops=dict(arrowstyle="wedge,tail_width=0.5", alpha=0.1));

        # Задаем метки для осей координат
        ax.set(title='USA births by day of year (1969-1988)',
               ylabel='average daily births')

        # Размечаем ось X центрированными метками для месяцев
        ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
        ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
        ax.xaxis.set_major_formatter(plt.NullFormatter())
        ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));

        ax.set_ylim(3600, 5400);
```

Рис. 32.6. Примеры поясняющих надписей

Богатство параметров делают `annotate` мощным и гибким инструментом: с его помощью можно создавать стрелки любых форм и размеров. К сожалению, это также означает, что создание подобных элементов требует отладки вручную — процесс, занимающий немало времени, если речь идет о создании графики типографского уровня качества! Наконец, отмечу, что использовать для представления данных продемонстрированную выше смесь стилей я отнюдь не рекомендую, она дана в качестве примера возможностей.

Дальнейшее обсуждение и примеры стилей стрелок и поясняющих надписей можно найти в галерее библиотеки Matplotlib (https://oreil.ly/abuPw).

ГЛАВА 33

Настройка делений на осях координат

Локаторы и форматеры делений на осях, используемые по умолчанию в библиотеке Matplotlib, спроектированы так, что в большинстве обычных ситуаций их вполне достаточно, хотя они отнюдь не оптимальны для всех графиков. В этой главе мы рассмотрим несколько примеров настройки размещения делений и их форматирования для конкретных интересующих нас видов графиков.

Прежде чем перейти к примерам, разберемся в объектной иерархии графиков библиотеки Matplotlib. Matplotlib старается представлять как объекты Python все элементы на графике, например объект `Figure` представляет прямоугольник, ограничивающий снаружи все элементы графика. Каждый объект библиотеки Matplotlib также служит контейнером для вложенных объектов. Например, любой объект `Figure` может содержать один или более объектов `Axes`, каждый из которых, в свою очередь, содержит другие объекты, отражающие содержимое графика.

Метки делений не исключение. У каждого объекта `Axes` имеются атрибуты `xaxis` и `yaxis`, которые, в свою очередь, содержат все свойства линий, делений и меток соответствующих осей координат.

Основные и промежуточные деления осей координат

На каждой оси координат имеются *основные* и *промежуточные* метки делений. Основные деления обычно больше или более заметны, а промежуточные — меньше. По умолчанию библиотека Matplotlib редко использует промежуточные деления, одно из мест, где их можно увидеть, — логарифмические графики (рис. 33.1):

```
In [1]: import matplotlib.pyplot as plt
        plt.style.use('classic')
        import numpy as np

        %matplotlib inline

In [2]: ax = plt.axes(xscale='log', yscale='log')
        ax.set(xlim=(1, 1E3), ylim=(1, 1E3))
        ax.grid(True);
```

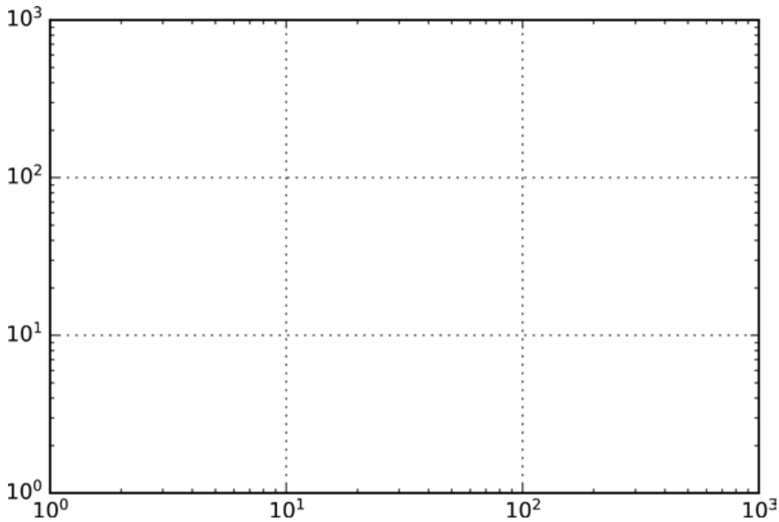


Рис. 33.1. Пример логарифмических шкал и меток

На этом графике видно, что каждое основное деление состоит из большого деления и метки, а промежуточное — из маленького деления без метки.

Этими свойствами делений (местоположением и выводом меток) можно управлять, настроив объекты `formatter` и `locator` каждой из осей. Рассмотрим их значения для оси X на только что созданном графике:

```
In [3]: print(ax.xaxis.get_major_locator())
        print(ax.xaxis.get_minor_locator())
Out[3]: <matplotlib.ticker.LogLocator object at 0x1129b9370>
        <matplotlib.ticker.LogLocator object at 0x1129aaf70>

In [4]: print(ax.xaxis.get_major_formatter())
        print(ax.xaxis.get_minor_formatter())
Out[4]: <matplotlib.ticker.LogFormatterSciNotation object at 0x1129aaa00>
        <matplotlib.ticker.LogFormatterSciNotation object at 0x1129aac10>
```

Как видите, местоположение меток как основных, так и промежуточных делений задает локатор `LogLocator` (что логично для логарифмического графика). Метки промежуточных делений форматируются форматером `NullFormatter` (это означает, что метки отображаться не будут).

Продемонстрируем несколько примеров настройки этих локаторов и форматеров для различных графиков.

Сокрытие делений и/или меток

Наиболее частая операция с делениями/метками — их сокрытие с помощью классов `plt.NullLocator` и `plt.NullFormatter`, как показано на рис. 33.2:

```
In [5]: ax = plt.axes()
        rng = np.random.default_rng(1701)
        ax.plot(rng.random(50))
        ax.grid()

        ax.yaxis.set_major_locator(plt.NullLocator())
        ax.xaxis.set_major_formatter(plt.NullFormatter())
```

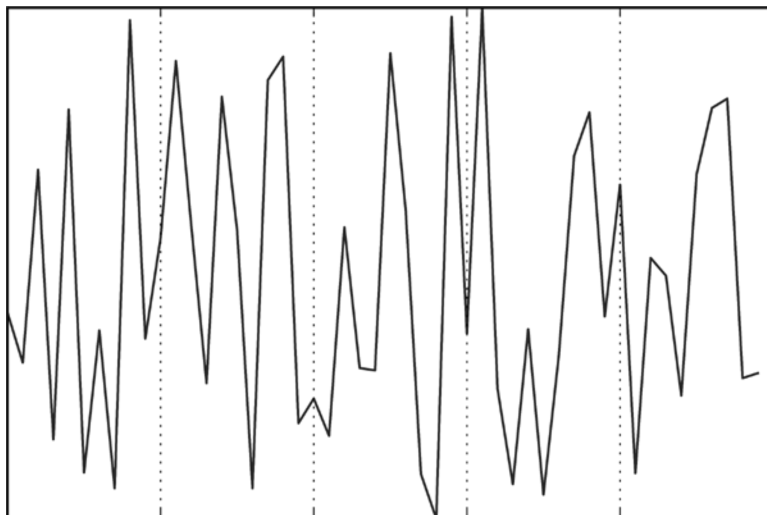


Рис. 33.2. График со скрытыми метками делений (ось X) и скрытыми делениями (ось Y)

Мы убрали метки (но оставили деления/линии координатной сетки) с оси *X* и убрали деления (а следовательно, и метки) с оси *Y*. Отсутствие делений мо-

жет быть полезно во многих случаях, например, если нужно отобразить сетку изображений. Например, рассмотрим рис. 33.3, содержащий изображения лиц людей, — пример, часто используемый в задачах машинного обучения с учителем (см., например, главу 43):

```
In [6]: fig, ax = plt.subplots(5, 5, figsize=(5, 5))
        fig.subplots_adjust(hspace=0, wspace=0)

        # Получаем данные по лицам людей из библиотеки Scikit-Learn
        from sklearn.datasets import fetch_olivetti_faces
        faces = fetch_olivetti_faces().images

        for i in range(5):
            for j in range(5):
                ax[i, j].xaxis.set_major_locator(plt.NullLocator())
                ax[i, j].yaxis.set_major_locator(plt.NullLocator())
                ax[i, j].imshow(faces[10 * i + j], cmap='binary_r')
```

Рис. 33.3. Сокрытие делений на графиках с изображениями

Каждое изображение — это отдельная система координат, и мы сделали локаторы пустыми, поскольку значения делений (в данном случае количество пикселов) не несут никакой относящейся к делу информации.

Уменьшение или увеличение количества делений

Распространенная проблема с настройками по умолчанию — слишком близкое друг к другу размещение меток на маленьких субграфиках. Это заметно на сетке графиков, показанной на рис. 33.4:

```
In [7]: fig, ax = plt.subplots(4, 4, sharex=True, sharey=True)
```

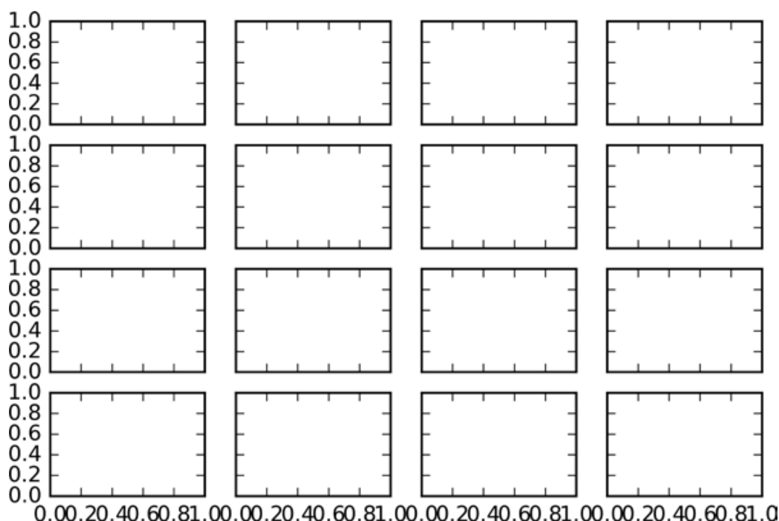


Рис. 33.4. Вид рисунка по умолчанию со слишком плотно расположенными делениями

Числа практически сливаются друг с другом, особенно в метках вдоль оси *X*, из-за чего их очень сложно разобрать. Исправить это можно с помощью класса `plt.MaxNLocator`, который дает возможность задавать максимальное отображаемое количество делений. При задании этого числа конкретные местоположения делений выберет внутренняя логика библиотеки Matplotlib (рис. 33.5):

```
In [8]: # Задаем локаторы основных делений осей X и Y для всех систем координат
        for axi in ax.flat:
            axi.xaxis.set_major_locator(plt.MaxNLocator(3))
            axi.yaxis.set_major_locator(plt.MaxNLocator(3))
        fig
```

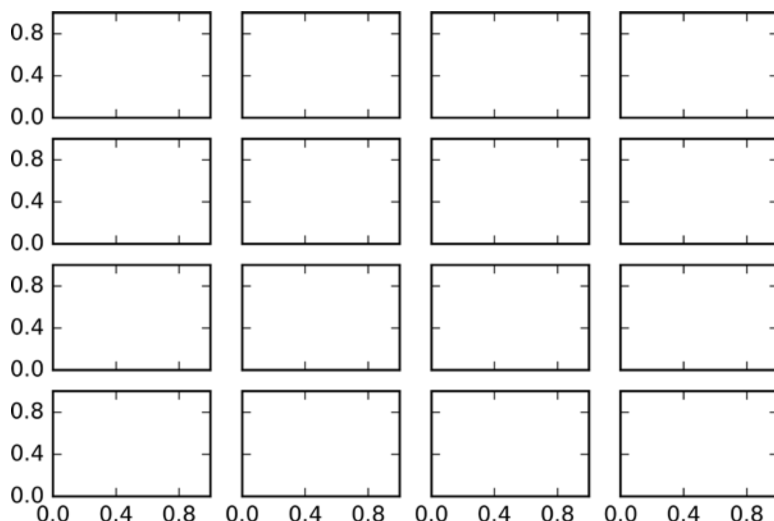


Рис. 33.5. Уменьшенное число делений

Благодаря этому рисунок становится гораздо понятнее. При необходимости более точного управления размещением делений с равными интервалами можно воспользоваться классом `plt.MultipleLocator`, который мы обсудим в следующем разделе.

Экзотические форматы делений

Форматирование делений по умолчанию в библиотеке Matplotlib оставляет желать лучшего. В качестве варианта по умолчанию, который бы подходил для широкого спектра ситуаций, оно работает неплохо, но иногда требуется нечто большее. Рассмотрим показанный на рис. 33.6 график синуса и косинуса:

```
In [9]: # Строим графики синуса и косинуса
        fig, ax = plt.subplots()
        x = np.linspace(0, 3 * np.pi, 1000)
        ax.plot(x, np.sin(x), lw=3, label='Sine')    # Синус
        ax.plot(x, np.cos(x), lw=3, label='Cosine')  # Косинус

        # Настраиваем сетку, легенду и пределы осей координат
        ax.grid(True)
        ax.legend(frameon=False)
        ax.axis('equal')
        ax.set_xlim(0, 3 * np.pi);
```

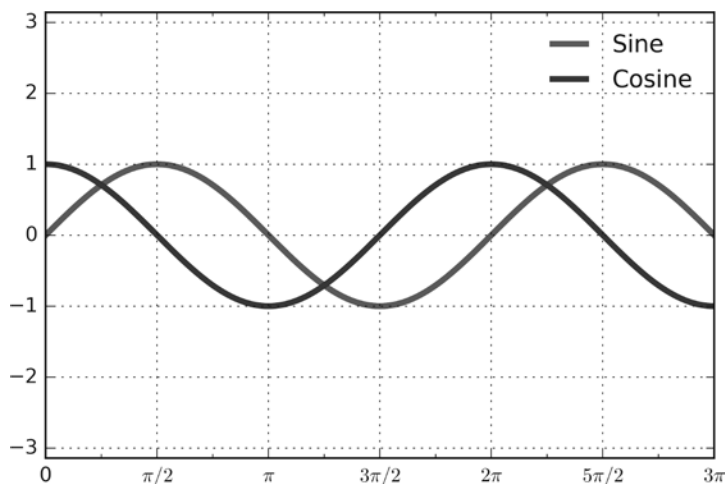


Рис. 33.6. График по умолчанию с целочисленными делениями

Хотелось бы внести несколько изменений. Во-первых, для такого рода данных лучше располагать деления и линии сетки по точкам, кратным числу π. Сделать это можно, настроив локатор `MultipleLocator`, располагающий деления в точках, кратных переданному ему числу. В дополнение добавим промежуточные деления в точках, кратных π/2 и π/4 (рис. 33.7):

```
In [10]: ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
         ax.xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 4))
         fig
```

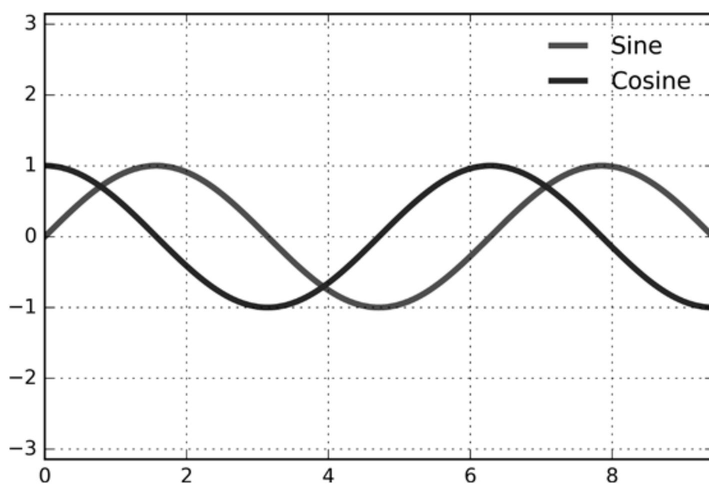


Рис. 33.7. Деления в точках, кратных π/2 и π/4

Но теперь метки делений выглядят несколько по-дурацки: можно догадаться, что они кратны π, но из десятичного представления это неочевидно. Необходимо модифицировать форматер делений. Встроенного форматера для нашей задачи нет, поэтому воспользуемся форматером `plt.FuncFormatter`, принимающим пользовательскую функцию, обеспечивающую более точное управление форматом вывода делений (рис. 33.8):

```
In [11]: def format_func(value, tick_number):
             # Определяем количество кратных пи/2 значений
             N = int(np.round(2 * value / np.pi))
             if N == 0:
                 return "0"
             elif N == 1:
                 return r"$\pi/2$"
             elif N == 2:
                 return r"$\pi$"
             elif N % 2 > 0:
                 return rf"${N}\pi/2$"
             else:
                 return rf"${N // 2}\pi$"

         ax.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
         fig
```

Намного лучше! Обратите внимание: мы воспользовались тем, что библиотека Matplotlib поддерживает систему верстки LaTeX. Для ее использования необходимо заключить нужную строку в знаки доллара с двух сторон. Это облегчает отображение математических символов и формул. В нашем случае `"$\pi$"` отображается в виде греческой буквы π.

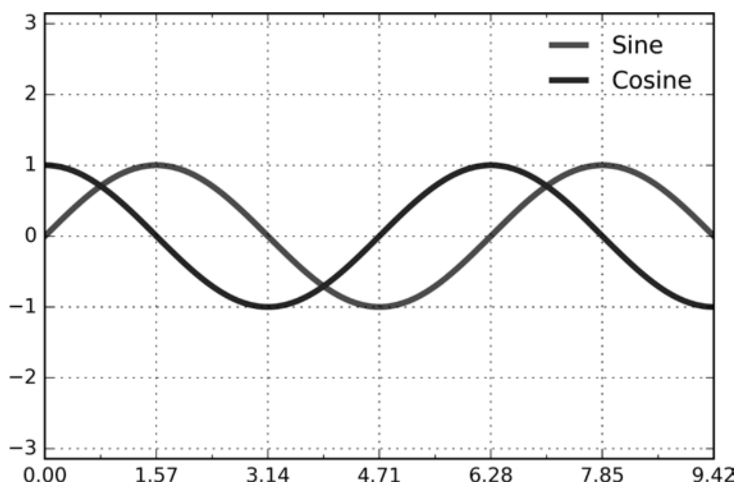


Рис. 33.8. Деления с настроенными метками

Краткая сводка локаторов и форматеров

Мы уже видели несколько форматеров и локаторов. В заключение этой главы мы перечислим все встроенные локаторы (табл. 33.1) и форматеры (табл. 33.2). Дополнительную информацию о них вы можете найти в их docstring или онлайн-документации библиотеки Matplotlib. Все перечисленные ниже форматеры и локаторы доступны в пространстве имен `plt`.

Таблица 33.1. Локаторы

Класс локатора	Описание
`NullLocator`	Без делений
`FixedLocator`	Положения делений фиксированы
`IndexLocator`	Локатор для графика индексированной переменной (например, для `x = range(len(y))`)
`LinearLocator`	Равномерно распределенные деления от min до max
`LogLocator`	Логарифмически распределенные деления от min до max
`MultipleLocator`	Деления и диапазон значений кратны заданному основанию
`MaxNLocator`	Находит удачные местоположения для делений в количестве, не превышающем заданного максимального числа
`AutoLocator`	(по умолчанию) `MaxNLocator` с простейшими значениями по умолчанию
`AutoMinorLocator`	Локатор для промежуточных делений

Таблица 33.2. Форматеры

Класс форматера	Описание
`NullFormatter`	Деления без меток
`IndexFormatter`	Задает строковые значения для меток на основе списка
`FixedFormatter`	Позволяет задавать строковые значения для меток вручную
`FuncFormatter`	Значения меток задаются с помощью пользовательской функции
`FormatStrFormatter`	Для всех значений используется строка формата
`ScalarFormatter`	Форматер по умолчанию для скалярных значений
`LogFormatter`	Форматер по умолчанию для логарифмических систем координат

В последующих главах мы увидим еще примеры использования этих классов.

ГЛАВА 34

Настройка Matplotlib: конфигурации и таблицы стилей

В предыдущих главах мы затронули множество тем, так или иначе связанных с настройкой оформления отдельных элементов графиков, однако Matplotlib имеет также механизмы для настройки общего стиля всей диаграммы сразу. В этой главе мы рассмотрим некоторые конфигурационные параметры среды выполнения (runtime configuration, *rc*) библиотеки Matplotlib и новую особенность — *таблицы стилей* (stylesheets), содержащие неплохие наборы конфигураций по умолчанию.

Настройка графиков вручную

На протяжении всей этой части книги мы видели, как менять отдельные настройки графиков, получая в итоге нечто более приятное глазу, чем настройки по умолчанию. Эти настройки можно выполнять и для каждого графика отдельно. Например, вот гистограмма с довольно скучным оформлением по умолчанию (рис. 34.1):

```
In [1]: import matplotlib.pyplot as plt
        plt.style.use('classic')
        import numpy as np

        %matplotlib inline
In [2]: x = np.random.randn(1000)
        plt.hist(x);
```

Мы можем настроить ее вид вручную, превратив эту гистограмму в намного более приятный глазу график (рис. 34.2):

```
In [3]: # Используем серый фон
        fig = plt.figure(facecolor='white')
        ax = plt.axes(facecolor='#E6E6E6')
        ax.set_axisbelow(True)
```

```
# Рисуем сплошные белые линии сетки
plt.grid(color='w', linestyle='solid')

# Скрываем основные линии осей координат
for spine in ax.spines.values():
    spine.set_visible(False)

# Скрываем деления сверху и справа
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()

# Осветляем цвет делений и меток
ax.tick_params(colors='gray', direction='out')
for tick in ax.get_xticklabels():
    tick.set_color('gray')
for tick in ax.get_yticklabels():
    tick.set_color('gray')

# Задаем цвет заливки и границ гистограммы
ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');
```

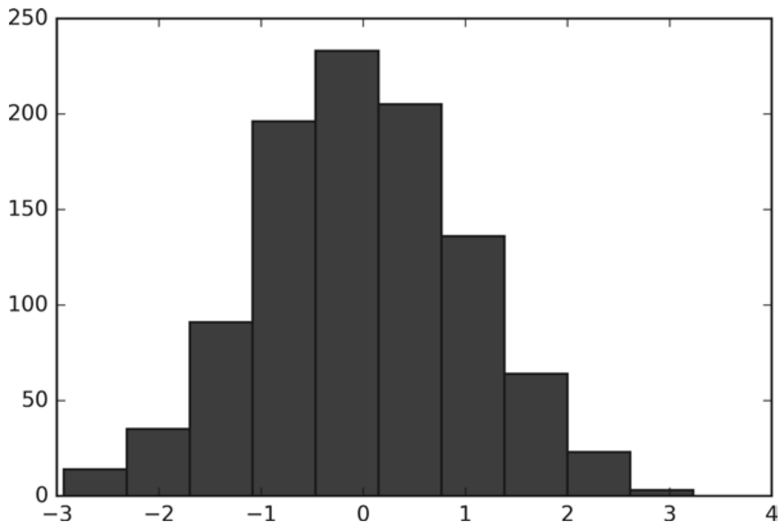


Рис. 34.1. Гистограмма с оформлением по умолчанию

Выглядит намного лучше, и можно заметить, что образцом для этого оформления послужил пакет визуализации `ggplot` языка R. Но чтобы получить его, потребовалось немало труда, и не хотелось бы делать одно и то же для каждого создаваемого графика. К счастью, существует способ задать эти настройки один раз для всех графиков.

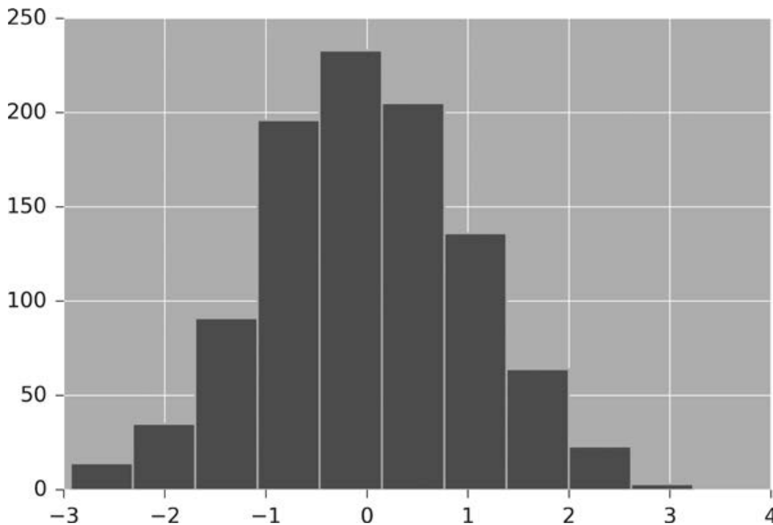


Рис. 34.2. Гистограмма с оформлением, настроенным вручную

Изменение значений по умолчанию: rcParams

Каждый раз при загрузке библиотеки Matplotlib она определяет конфигурацию среды выполнения, содержащую стили по умолчанию для всех элементов графиков. Эту конфигурацию можно настроить в любой момент, воспользовавшись удобной утилитой `plt.rc`. Давайте посмотрим, как модифицировать параметры rc, чтобы по умолчанию графики получали оформление, подобное показанному выше.

Воспользуемся функцией `plt.rc` и изменим некоторые из настроек:

```
In [4]: from matplotlib import cycler
        colors = cycler('color',
                        ['#EE6666', '#3388BB', '#9988DD',
                         '#EECC55', '#88BB44', '#FFBBBB'])
        plt.rc('figure', facecolor='white')
        plt.rc('axes', facecolor='#E6E6E6', edgecolor='none',
               axisbelow=True, grid=True, prop_cycle=colors)
        plt.rc('grid', color='w', linestyle='solid')
        plt.rc('xtick', direction='out', color='gray')
        plt.rc('ytick', direction='out', color='gray')
        plt.rc('patch', edgecolor='#E6E6E6')
        plt.rc('lines', linewidth=2)
```

Теперь создадим график и посмотрим, действительно ли к нему будут применены наши настройки оформления (рис. 34.3):

```
In [5]: plt.hist(x);
```

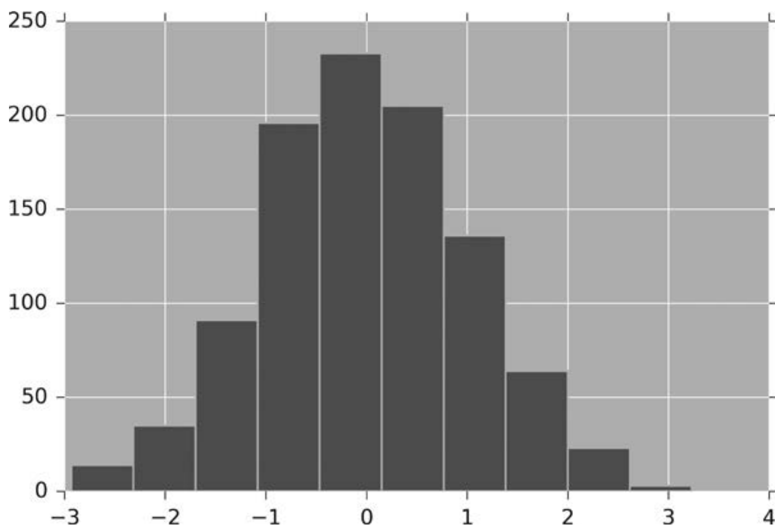


Рис. 34.3. Гистограмма с настройками оформления, измененными через rc

Посмотрим, как выглядят простые графики с этими настройками rc (рис. 34.4):

```
In [6]: for i in range(4):
            plt.plot(np.random.rand(10))
```

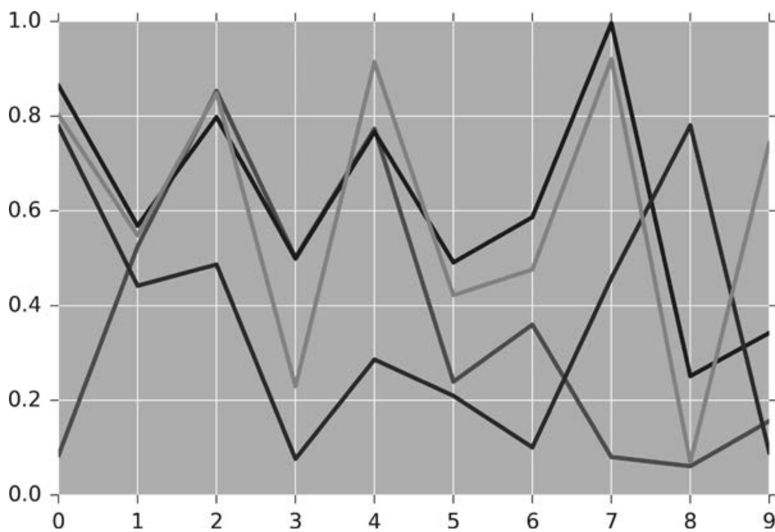


Рис. 34.4. Линейный график с измененными настройками оформления

Для диаграмм, отображаемых на экране, а не на бумаге, такие стили представляются мне гораздо более эстетически приятными, чем стили по умолчанию. Если мое чувство прекрасного расходится с вашим, то у меня есть для вас хорошая новость: вы можете настроить параметры rc под свой вкус! Эти настройки можно затем сохранить в файле `.matplotlibrc`, подробно описанном в документации библиотеки Matplotlib (https://oreil.ly/UwM2u).

Таблицы стилей

Новый механизм настройки стилей диаграмм реализован в Matplotlib в виде модуля `style`. Он включает ряд таблиц стилей по умолчанию, а также позволяет создавать и упаковывать собственные стили. Формат этих таблиц стилей аналогичен упомянутому выше файлу `.matplotlibrc`, но хранятся они в файлах с расширением `.mplstyle`.

Даже если вы не хотите создавать свои стили, вы с успехом можете использовать уже готовые стили. Имеющиеся стили перечислены в `plt.style.available` — для краткости приведу только первые пять:

```
In [7]: plt.style.available[:5]
Out[7]: ['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic',
        >'dark_background']
```

Стандартный способ переключить таблицу стилей — вызвать функцию `style.use`:

```
plt.style.use('stylename')
```

Но не забывайте, что это приведет к изменению стиля на весь остаток сеанса! В качестве альтернативы можно воспользоваться диспетчером контекста стилей, устанавливающим стиль временно:

```
with plt.style.context('stylename'):
    make_a_plot()
```

Для демонстрации создадим функцию, рисующую два простых графика:

```
In [8]: def hist_and_lines():
            np.random.seed(0)
            fig, ax = plt.subplots(1, 2, figsize=(11, 4))
            ax[0].hist(np.random.randn(1000))
            for i in range(3):
                ax[1].plot(np.random.rand(10))
            ax[1].legend(['a', 'b', 'c'], loc='lower left')
```

Воспользуемся ею, чтобы увидеть, как выглядят эти графики при использовании различных встроенных стилей.

Стиль по умолчанию default

Стиль `default` был обновлен в версии Matplotlib 2.0; рассмотрим его первым (рис. 34.5):

```
In [9]: with plt.style.context('default'):
            hist_and_lines()
```

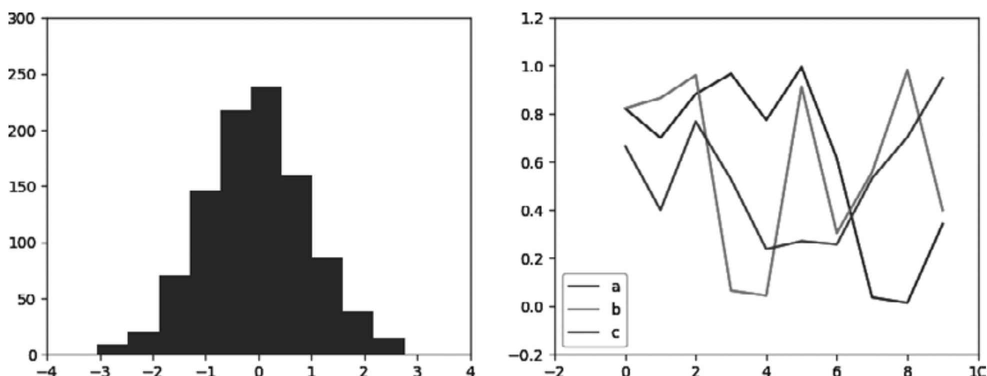


Рис. 34.5. Стиль default в библиотеке Matplotlib

Стиль FiveThirtyEight

Стиль `fivethirtyeight` подражает оформлению популярного сайта FiveThirtyEight (http://fivethirtyeight.com/). Как можно видеть на рис. 34.6, он использует жирные шрифты, толстые линии и прозрачные оси координат:

```
In [10]: with plt.style.context('fivethirtyeight'):
             hist_and_lines()
```

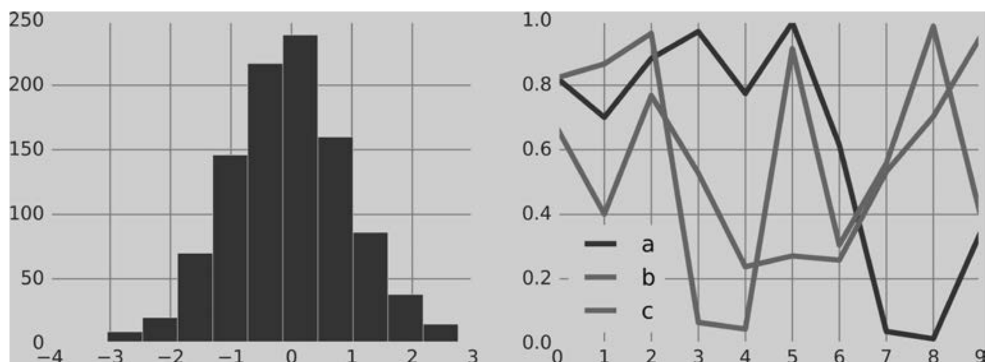


Рис. 34.6. Стиль fivethirtyeight

Стиль ggplot

В языке программирования R пакет `ggplot` — очень популярное средство визуализации. Стиль `ggplot` в библиотеке Matplotlib подражает стилям по умолчанию из этого пакета (рис. 34.7):

```
In [11]: with plt.style.context('ggplot'):
             hist_and_lines()
```

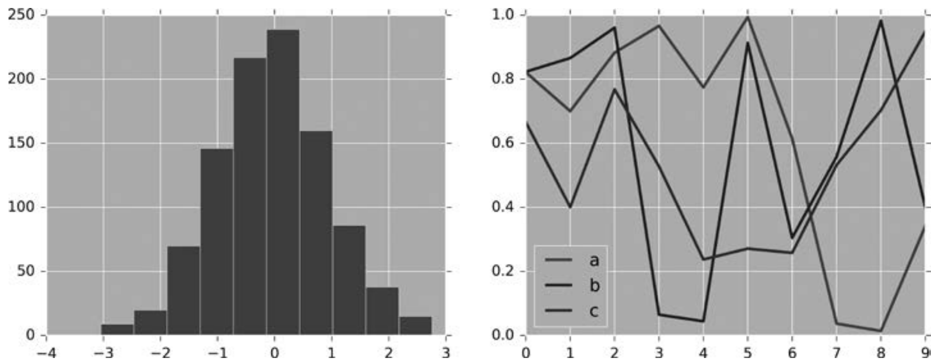


Рис. 34.7. Стиль ggplot

Стиль «байесовские методы для хакеров»

Существует замечательная онлайн-книга *Probabilistic Programming and Bayesian Methods for Hackers* (https://oreil.ly/9JIb7). Она содержит рисунки, созданные с помощью Matplotlib, с использованием набора параметров rc для придания им единообразного внешнего вида. Этот стиль воспроизведен в виде таблицы стилей `bmh` (рис. 34.8):

```
In [12]: with plt.style.context('bmh'):
             hist_and_lines()
```

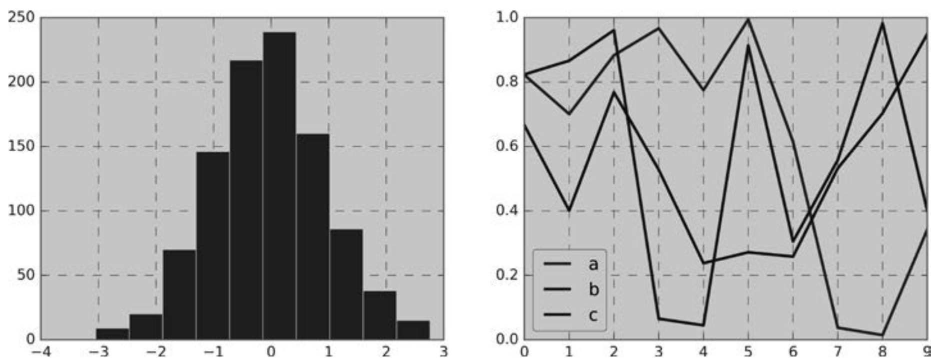


Рис. 34.8. Стиль bmh

Стиль с темным фоном

Для использования в презентациях рисунки с темным фоном часто удобнее, чем со светлым. Эту возможность предоставляет стиль `dark_background` (рис. 34.9):

```
In [13]: with plt.style.context('dark_background'):
             hist_and_lines()
```

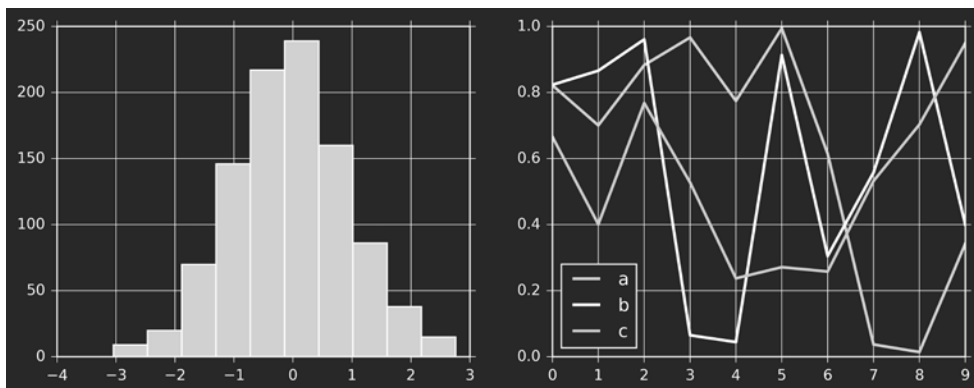


Рис. 34.9. Стиль dark_background

Оттенки серого

Иногда приходится готовить для печатного издания черно-белые рисунки. Для этого может пригодиться стиль `grayscale`, продемонстрированный на рис. 34.10:

```
In [14]: with plt.style.context('grayscale'):
             hist_and_lines()
```

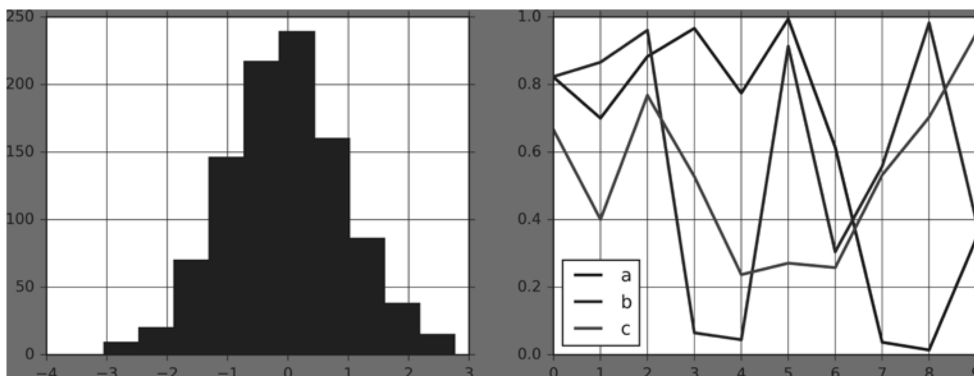


Рис. 34.10. Стиль grayscale

Стиль Seaborn

В библиотеке Matplotlib есть также стили, источником вдохновения для которых послужила библиотека Seaborn (обсуждаемая подробнее в главе 36). Мне эти настройки очень нравятся, и я склонен использовать их как настройки по умолчанию в моих собственных исследованиях данных (рис. 34.11):

```
In [15]: with plt.style.context('seaborn-whitegrid'):
             hist_and_lines()
```

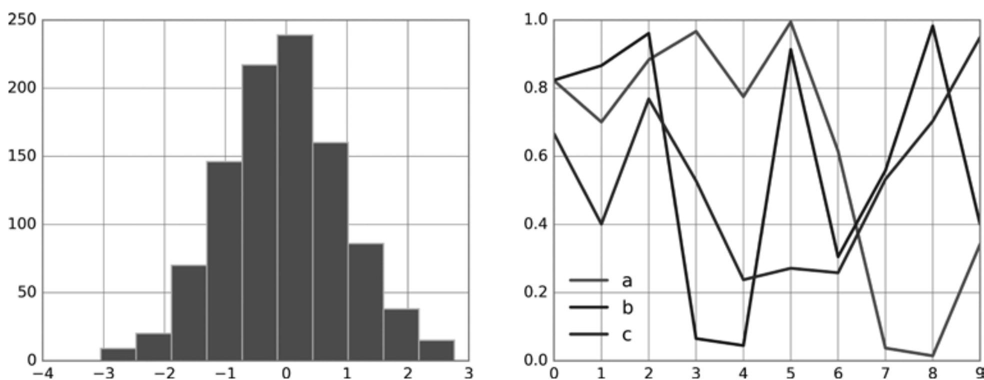


Рис. 34.11. Стиль Seaborn

Обязательно найдите время и отыщите встроенный стиль, который вам понравится! В этой книге я обычно использую при создании графиков один или несколько таких стилей.

ГЛАВА 35

Построение трехмерных графиков в библиотеке Matplotlib

Первоначально библиотека Matplotlib создавалась для построения только двумерных графиков. На момент выпуска версии 1.0 на основе реализованных в библиотеке Matplotlib средств отображения двумерных графиков было создано несколько утилит построения трехмерных графиков. Так появился набор удобных (хотя и несколько ограниченных в возможностях) инструментов для трехмерной визуализации данных. Поддержка построения трехмерных графиков включается импортированием набора инструментов `mplot3d`, входящего в дистрибутив библиотеки Matplotlib:

```
In [1]: from mpl_toolkits import mplot3d
```

После импорта этого модуля появляется возможность создавать трехмерные системы координат путем передачи именованного аргумента `projection='3d'` любой из обычных функций создания систем координат (рис. 35.1):

```
In [2]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

In [3]: fig = plt.figure()
        ax = plt.axes(projection='3d')
```

Используя такую трехмерную систему координат, можно строить различные виды трехмерных графиков. Построение трехмерных графиков — один из видов функциональности, для которых полезнее интерактивный, а не статический просмотр рисунков в блокноте. Напомню, что для работы с интерактивными рисунками необходимо вместо команды `%matplotlib inline` использовать команду `%matplotlib notebook`.

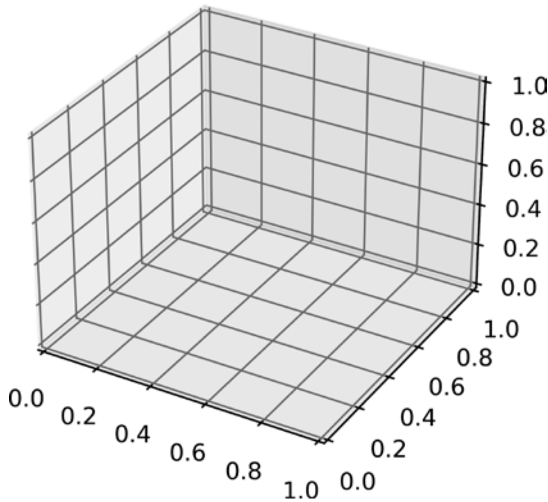


Рис. 35.1. Пустая трехмерная система координат

Трехмерные точки и линии

Линейный график и диаграмма рассеяния — простейшие трехмерные графики, создаваемые на основе множеств трехэлементных кортежей (x, y, z). По аналогии с обсуждавшимися ранее более распространенными двумерными графиками их можно создать с помощью функций `ax.plot3D` и `ax.scatter3D`. Сигнатуры этих функций практически совпадают с двумерными аналогами, поэтому за более подробной информацией по настройке отображения ими данных вы можете обратиться к главам 26 и 27. Следующий пример строит график тригонометрической спирали, а также рисует рядом с кривой несколько точек (рис. 35.2):

```
In [4]: ax = plt.axes(projection='3d')

        # Данные для трехмерной кривой
        zline = np.linspace(0, 15, 1000)
        xline = np.sin(zline)
        yline = np.cos(zline)
        ax.plot3D(xline, yline, zline, 'gray')

        # Данные для трехмерных точек
        zdata = 15 * np.random.random(100)
        xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
        ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
        ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

Обратите внимание, что по умолчанию степень прозрачности точек на диаграмме рассеяния настраивается так, чтобы придать графику эффект глубины.

В статическом изображении этот трехмерный эффект иногда незаметен, но в интерактивном представлении может помочь передать информацию о топологии точек.

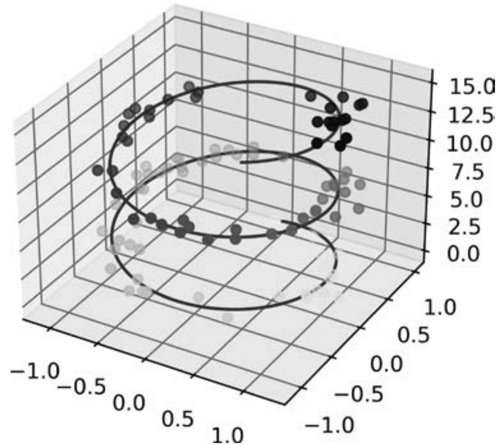


Рис. 35.2. Точки и линии в трех измерениях

Трехмерные контурные графики

Аналогично контурным графикам, рассмотренным нами в главе 28, `mplot3d` содержит инструменты для создания трехмерных рельефных графиков на основе тех же входных данных. Подобно `ax.contour`, функция `ax.contour3D` требует, чтобы все входные данные находились в форме двумерных регулярных сеток, с вычисляемым значением координаты Z в каждой точке. Следующий пример демонстрирует трехмерную контурную диаграмму трехмерной синусоиды (рис. 35.3):

```
In [5]: def f(x, y):
            return np.sin(np.sqrt(x ** 2 + y ** 2))

        x = np.linspace(-6, 6, 30)
        y = np.linspace(-6, 6, 30)

        X, Y = np.meshgrid(x, y)
        Z = f(X, Y)

In [6]: fig = plt.figure()
        ax = plt.axes(projection='3d')
        ax.contour3D(X, Y, Z, 40, cmap='binary')
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z');
```

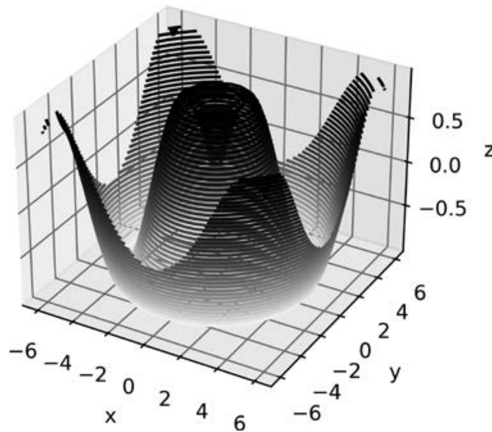


Рис. 35.3. Трехмерный контурный график

Иногда используемый по умолчанию угол зрения неидеален. В этом случае можно вызвать метод `view_init`, чтобы задать азимут и угол возвышения. В нашем примере (результат которого показан на рис. 35.4) используются угол возвышения 60 градусов (то есть 60 градусов над плоскостью $X-Y$) и азимут 35 градусов (то есть график повернут на 35 градусов против часовой стрелки вокруг оси Z):

```
In[7]: ax.view_init(60, 35)
       fig
```

Выполнить такое вращение можно и интерактивно, ухватив и перетащив мышью график Matplotlib, нарисованный в интерактивном режиме.

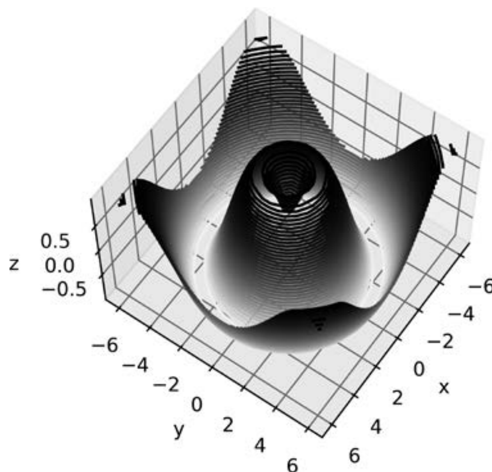


Рис. 35.4. Настройка угла зрения для трехмерного графика

Каркасы и поверхностные графики

Каркасные и поверхностные графики — еще два типа трехмерных графиков, подходящих для отображения данных, привязанных к координатам. Соответствующие функции принимают таблицу значений и проецируют их на заданную трехмерную поверхность, облегчая наглядное представление полученных трехмерных фигур. Вот пример с каркасом (рис. 35.5):

```
In [8]: fig = plt.figure()
        ax = plt.axes(projection='3d')
        ax.plot_wireframe(X, Y, Z)
        ax.set_title('wireframe');
```

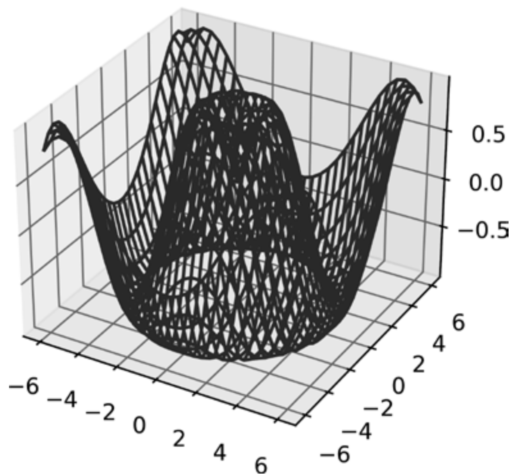


Рис. 35.5. Каркасный график

Поверхностный график выглядит аналогично каркасному, но каждая грань отображается как многоугольник с заливкой. Добавление карты цветов для многоугольников помогает лучше прочувствовать топологию отображаемой поверхности (рис. 35.6):

```
In [9]: ax = plt.axes(projection='3d')
        ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                        cmap='viridis', edgecolor='none')
        ax.set_title('surface');
```

Обратите внимание, что, хотя координатные значения для поверхностного графика должны быть двумерными, он не обязан быть прямолинейным. Вот пример создания неполной сетки в полярной системе координат, которая при построении графика с помощью функции `surface3D` дает нам срез отображаемой функции (рис. 35.7):

```
In [10]: r = np.linspace(0, 6, 20)
         theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi, 40)
         r, theta = np.meshgrid(r, theta)

         X = r * np.sin(theta)
         Y = r * np.cos(theta)
         Z = f(X, Y)

         ax = plt.axes(projection='3d')
         ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                         cmap='viridis', edgecolor='none');
```

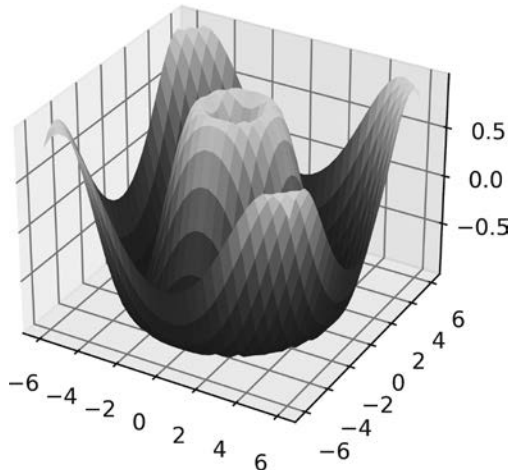


Рис. 35.6. Трехмерный поверхностный график

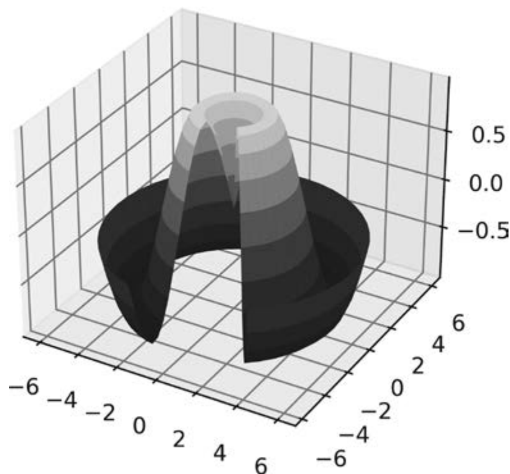


Рис. 35.7. Поверхностный график в полярной системе координат

Триангуляция поверхностей

В некоторых приложениях невозможно или слишком сложно получить регулярную координатную сетку с данными, необходимую функциям, представленным выше. В таких случаях можно с успехом использовать координатную сетку из треугольников. Представьте, что вместо значений, равномерно распределенных в декартовой или полярной системе координат, мы имеем дело с набором значений, выбиравшихся случайно без какой-либо системы.

```
In [11]: theta = 2 * np.pi * np.random.random(1000)
         r = 6 * np.random.random(1000)
         x = np.ravel(r * np.sin(theta))
         y = np.ravel(r * np.cos(theta))
         z = f(x, y)
```

Мы могли бы нарисовать диаграмму рассеяния точек, чтобы представить вид дискретизируемой поверхности (рис. 35.8):

```
In [12]: ax = plt.axes(projection='3d')
         ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5);
```

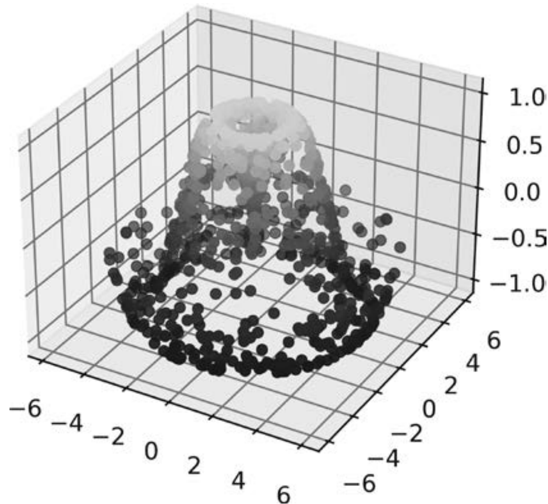


Рис. 35.8. Трехмерная дискретизированная поверхность

Полученное изображение оставляет желать лучшего. В подобных случаях нам может помочь функция `ax.plot_trisurf`, создающая поверхности путем поиска

набора треугольников, образуемых соседними точками (напомню, что x, y и z — это одномерные массивы), и построения поверхности из этих треугольников (результат показан на рис. 35.9):

```
In [13]: ax = plt.axes(projection='3d')
         ax.plot_trisurf(x, y, z,
                         cmap='viridis', edgecolor='none');
```

Результат не такой красивый, как при построении графика с помощью координатной сетки, но гибкость подобной триангуляции дает возможность создавать интересные трехмерные графики. Например, с помощью этого метода можно даже нарисовать трехмерную ленту Мёбиуса.

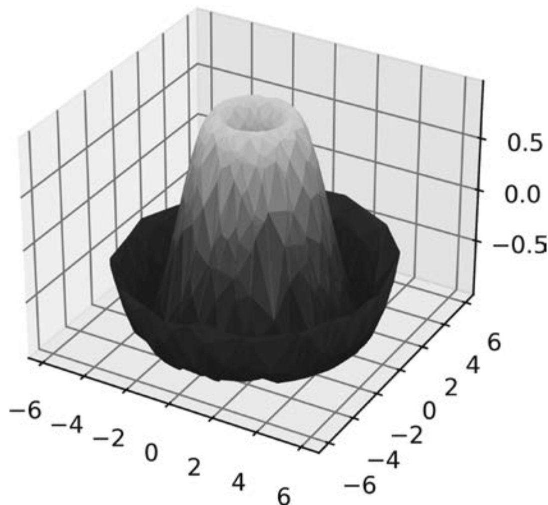


Рис. 35.9. Триангулированный участок поверхности

Пример: визуализация ленты Мёбиуса

Моделью ленты Мёбиуса может служить полоска бумаги, склеенная в кольцо концами, перевернутыми на 180 градусов. Она имеет весьма интересную топологию: несмотря на внешний вид, у нее только одна сторона! Следующий пример создает изображение ленты Мёбиуса с помощью трехмерных инструментов из библиотеки Matplotlib. Ключ к созданию ленты Мёбиуса — ее параметризация. Это двумерная лента, поэтому нам понадобятся для нее две собственные координаты. Назовем

одну из них θ (ее диапазон значений — от 0 до 2π), а вторую — w, с диапазоном значений от –1 на одном краю ленты (по ширине) до 1 на другом:

```
In [14]: theta = np.linspace(0, 2 * np.pi, 30)
         w = np.linspace(-0.25, 0.25, 8)
         w, theta = np.meshgrid(w, theta)
```

Теперь на основе этой параметризации вычислим координаты (x, y, z) ленты.

Немного поразмышляв, можно заметить, что в данном случае происходят два вращательных движения: одно — изменение расположения кольца относительно его центра (координата, которую мы назвали θ), а второе — скручивание полоски относительно ее оси координат (назовем эту координату φ). Чтобы получилась лента Мёбиуса, полоска должна выполнить половину оборота вдоль продольной оси за время полного свертывания в кольцо, то есть $\Delta\varphi = \Delta\theta / 2$:

```
In [15]: phi = 0.5 * theta
```

Теперь вспомним, как в тригонометрии создается трехмерная проекция. Определим переменную r — расстояние каждой точки от центра и воспользуемся ею для поиска внутренних координат (x, y, z):

```
In [16]: # радиус в плоскости X-Y
         r = 1 + w * np.cos(phi)

         x = np.ravel(r * np.cos(theta))
         y = np.ravel(r * np.sin(theta))
         z = np.ravel(w * np.sin(phi))
```

Для построения графика этого объекта нужно обеспечить правильное выполнение триангуляции. Лучший способ — описать триангуляцию *в координатах базовой параметризации*, после чего позволить библиотеке Matplotlib выполнить проекцию полученной триангуляции в трехмерное пространство ленты Мёбиуса. Это можно сделать следующим образом (рис. 35.10):

```
In [17]: # Выполняем триангуляцию в координатах базовой параметризации
         from matplotlib.tri import Triangulation
         tri = Triangulation(np.ravel(w), np.ravel(theta))

         ax = plt.axes(projection='3d')
         ax.plot_trisurf(x, y, z, triangles=tri.triangles,
                         cmap='Greys', linewidths=0.2);

         ax.set_xlim(-1, 1); ax.set_ylim(-1, 1); ax.set_zlim(-1, 1)
         ax.axis('off');
```

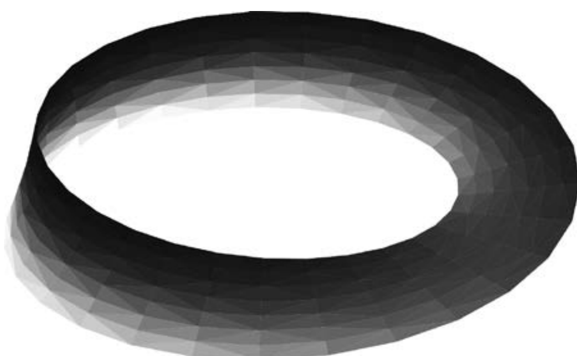


Рис. 35.10. Визуализация ленты Мёбиуса

Сочетая все эти методы, с помощью Matplotlib можно создавать и отображать самые разные трехмерные объекты и пространственные модели.

ГЛАВА 36

Визуализация с помощью библиотеки Seaborn

Библиотека Matplotlib зарекомендовала себя как невероятно удобный и популярный инструмент визуализации в научных исследованиях, но даже заядлые ее пользователи признают, что она зачастую оставляет желать лучшего. Существует несколько часто возникающих жалоб на Matplotlib.

- Типичная жалоба при использовании ранних версий библиотеки, которая ныне потеряла актуальность: до версии 2.0 параметры настройки цвета и стилей в библиотеке были далеки от идеала.
- Прикладной интерфейс библиотеки Matplotlib — относительно низкоуровневый. С его помощью можно создавать сложные статистические визуализации, но это требует *немало* шаблонного кода.
- Matplotlib была выпущена на десятилетие раньше, чем библиотека Pandas, и потому не ориентирована на работу с объектами `DataFrame`. Для визуализации данных из объектов `DataFrame` библиотеки Pandas приходится извлекать все объекты `Series` и преобразовывать их в нужный формат. Хорошо было бы иметь библиотеку для построения графиков, в которой присутствовали бы возможности по интеллектуальному использованию меток `DataFrame` на графиках.

Библиотека Seaborn (http://seaborn.pydata.org/) — решение этих проблем. Seaborn предоставляет API, основанный на API библиотеки Matplotlib, обеспечивающий разумные настройки стилей и цветов по умолчанию, определяющий простые высокоуровневые функции для часто встречающихся типов графиков и хорошо интегрирующийся с функциональностью, предоставляемой библиотекой Pandas.

Справедливости ради должен отметить, что команда разработчиков библиотеки Matplotlib тоже пытается решить эти проблемы: они добавили утилиты `plt.style` (которые мы обсуждали в главе 34) и принимают меры к более органичной обработке данных Pandas. Но по вышеизложенным причинам библиотека Seaborn остается исключительно удобным дополнением.

В соответствии с общепринятыми соглашениями библиотека Seaborn часто импортируется под псевдонимом `sns`:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn as sns
        import numpy as np
        import pandas as pd

        sns.set()  # Этот метод устанавливает стиль оформления
```

Анализируем графики Seaborn

Основная идея библиотеки Seaborn — предоставление высокоуровневых команд для создания множества различных типов графиков, удобных для исследования статистических данных и даже подгонки статистических моделей.

Рассмотрим некоторые из имеющихся в Seaborn наборов данных и типов графиков. Обратите внимание, что все изложенное далее *можно* реализовать и с помощью обычных команд библиотеки Matplotlib (фактически так и действует Seaborn), но Seaborn API намного удобнее.

Гистограммы, KDE и плотности

Зачастую все, что нужно сделать при визуализации статистических данных, — это построить гистограмму и график совместного распределения переменных. Мы уже видели, что в библиотеке Matplotlib сделать это относительно несложно (рис. 36.1):

```
In [2]: data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
        data = pd.DataFrame(data, columns=['x', 'y'])

        for col in 'xy':
            plt.hist(data[col], density=True, alpha=0.5)
```

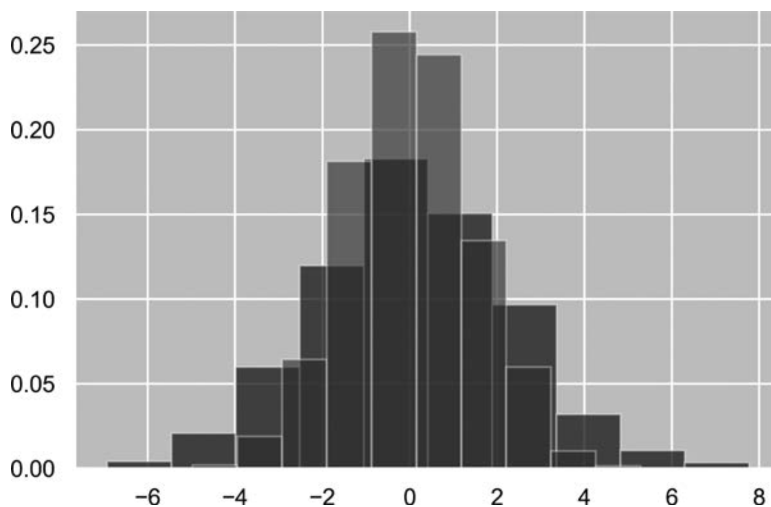


Рис. 36.1. Гистограммы для визуализации распределений

Вместо гистограммы можно построить гладкую аппроксимацию распределения путем ядерной оценки плотности распределения (см. главу 28), которую Seaborn выполняет с помощью функции `sns.kdeplot` (рис. 36.2):

```
In [3]: sns.kdeplot(data=data, shade=True);
```

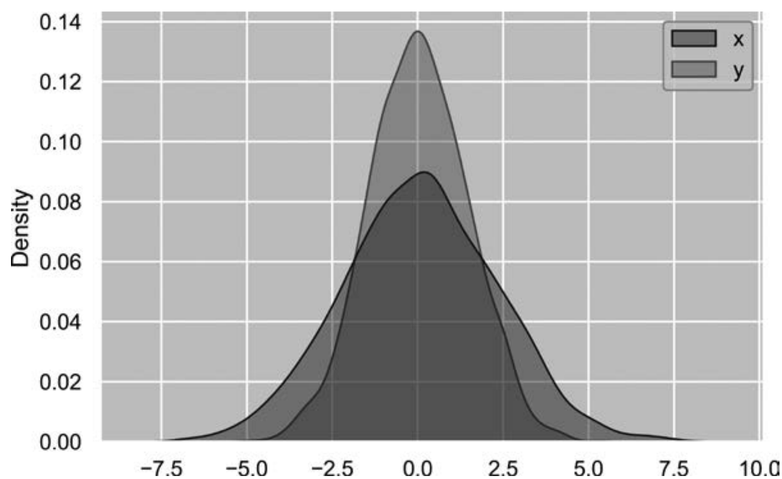


Рис. 36.2. Использование ядерной оценки плотности для визуализации распределений

Если в вызов функции kdeplot передать столбцы x и y, то можно получить двумерную визуализацию данных (рис. 36.3):

```
In [4]: sns.kdeplot(data=data, x='x', y='y');
```

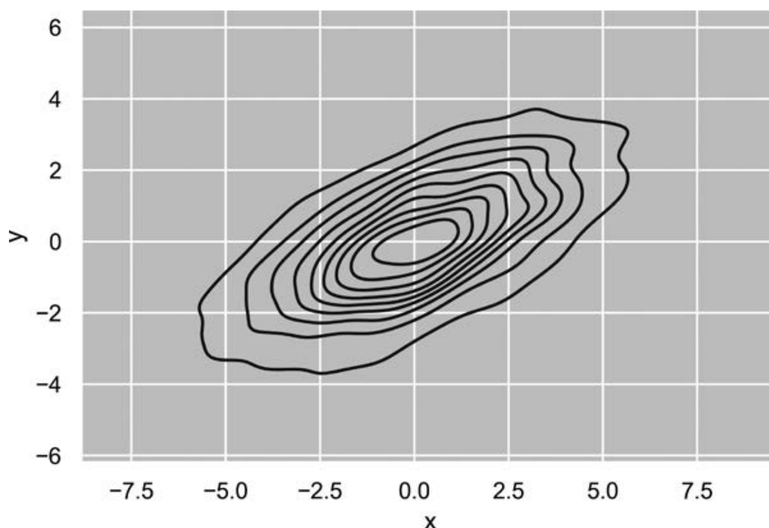


Рис. 36.3. Использование ядерной оценки плотности для визуализации распределений

Отобразить диаграмму совместного распределения и частных распределений можно с помощью функции sns.jointplot, которую мы рассмотрим далее в этой главе.

Графики пар

При обобщении графиков совместных распределений на наборы данных более высоких размерностей мы постепенно приходим к *графикам пар* (pair plots). Они очень удобны для изучения зависимостей между многомерными данными, когда необходимо построить график всех пар значений.

Продемонстрируем это на уже знакомом вам наборе данных Iris, содержащем измерения лепестков и чашелистиков трех видов ирисов:

```
In [5]: iris = sns.load_dataset("iris")
        iris.head()
Out[5]:    sepal_length  sepal_width  petal_length  petal_width species
        0           5.1          3.5           1.4          0.2  setosa
        1           4.9          3.0           1.4          0.2  setosa
```

```
2            4.7          3.2           1.3           0.2  setosa
3            4.6          3.1           1.5           0.2  setosa
4            5.0          3.6           1.4           0.2  setosa
```

Визуализация многомерных зависимостей между выборками сводится к вызову функции `sns.pairplot` (рис. 36.4):

```
In [6]: sns.pairplot(iris, hue='species', height=2.5);   # Вид
```

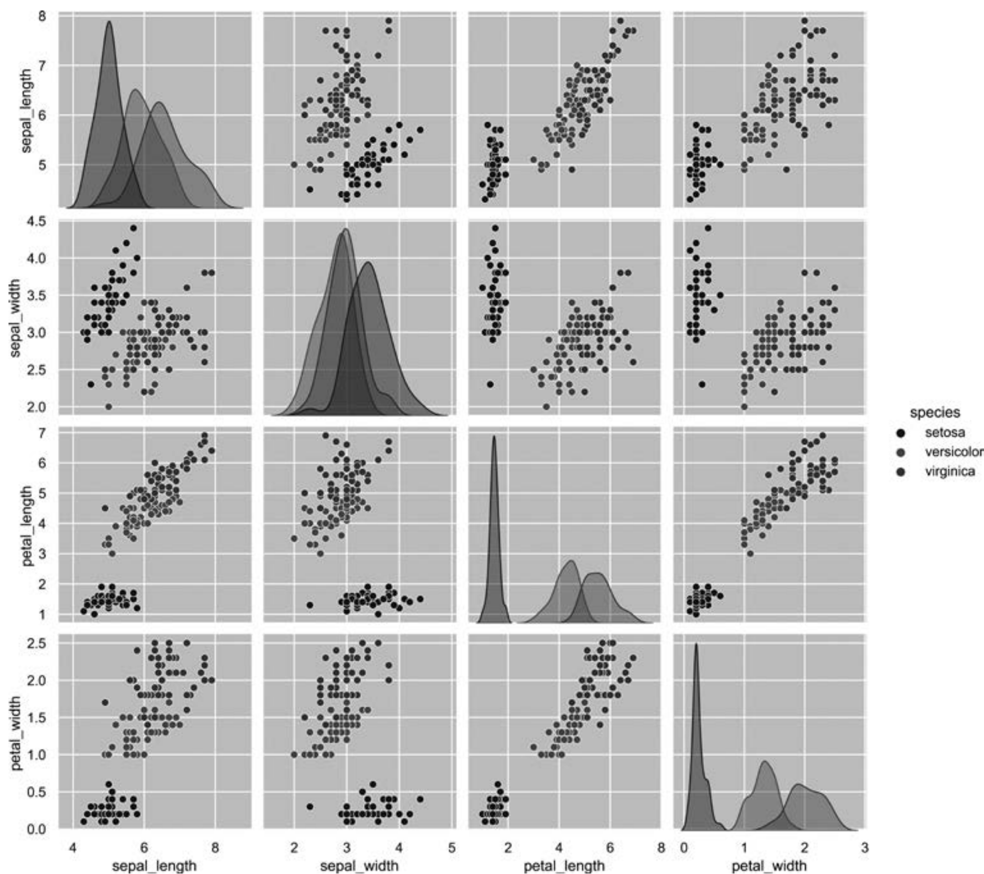


Рис. 36.4. График пар, демонстрирующий зависимости между четырьмя переменными

Фасетные гистограммы

Иногда оптимальный способ представления данных — гистограммы подмножеств, как показано на рис. 36.5. Класс `FacetGrid` из библиотеки Seaborn делает эту задачу элементарно простой. Рассмотрим данные, отображающие суммы,

которые персонал ресторана получает в качестве чаевых, в зависимости от различных признаков:

```
In [7]: tips = sns.load_dataset('tips')
        tips.head()
Out[7]:    total_bill   tip     sex smoker  day    time  size
        0       16.99  1.01  Female     No  Sun  Dinner     2
        1       10.34  1.66    Male     No  Sun  Dinner     3
        2       21.01  3.50    Male     No  Sun  Dinner     3
        3       23.68  3.31    Male     No  Sun  Dinner     2
        4       24.59  3.61  Female     No  Sun  Dinner     4

In [8]: tips['tip_pct'] = 100 * tips['tip'] / tips['total_bill']

        grid = sns.FacetGrid(tips, row="sex", col="time", margin_titles=True)
        grid.map(plt.hist, "tip_pct", bins=np.linspace(0, 40, 15));
```

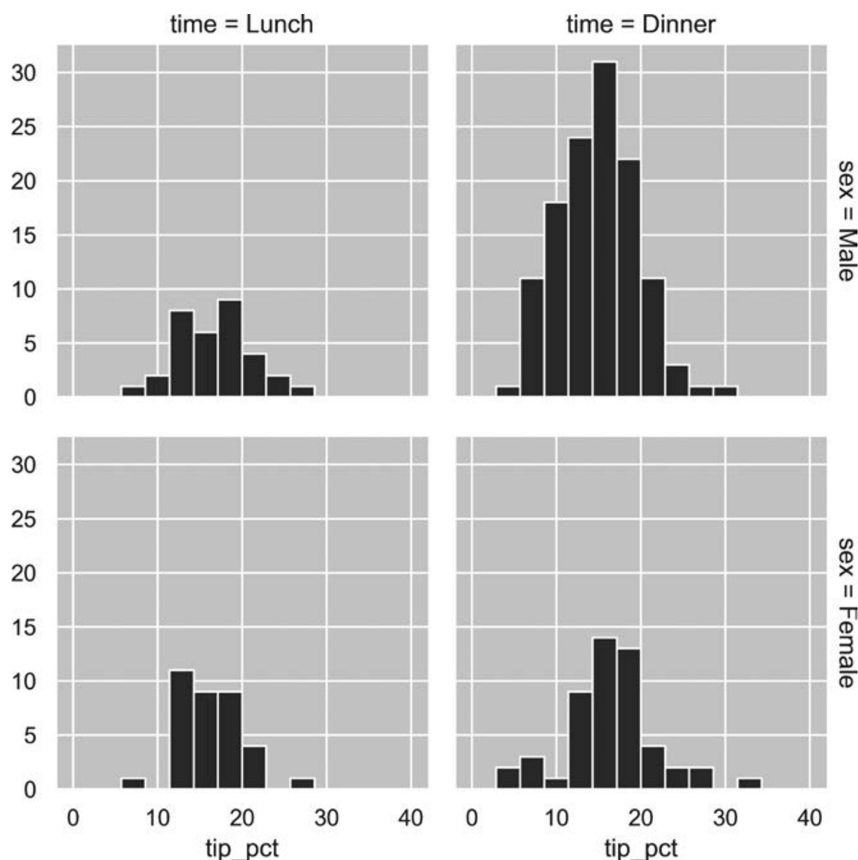


Рис. 36.5. Пример фасетной гистограммы

Фасетная диаграмма дает некоторое представление о наборе данных: например, мы видим, что в вечерние часы посетителей обслуживают преимущественно официанты-мужчины, а типичная сумма чаевых составляет примерно от 10 до 20 %, при этом наблюдаются отдельные выбросы как в меньшую, так и в большую сторону.

Графики факторов

Графики факторов тоже могут пригодиться для подобных исследований. Они позволяют просмотреть распределение параметра по интервалам, задаваемым любым другим параметром (рис. 36.6):

```
In [9]: with sns.axes_style(style='ticks'):
            g = sns.catplot(x="day", y="total_bill", hue="sex",
                            data=tips, kind="box")
            g.set_axis_labels("Day", "Total Bill")  # День, итого
```

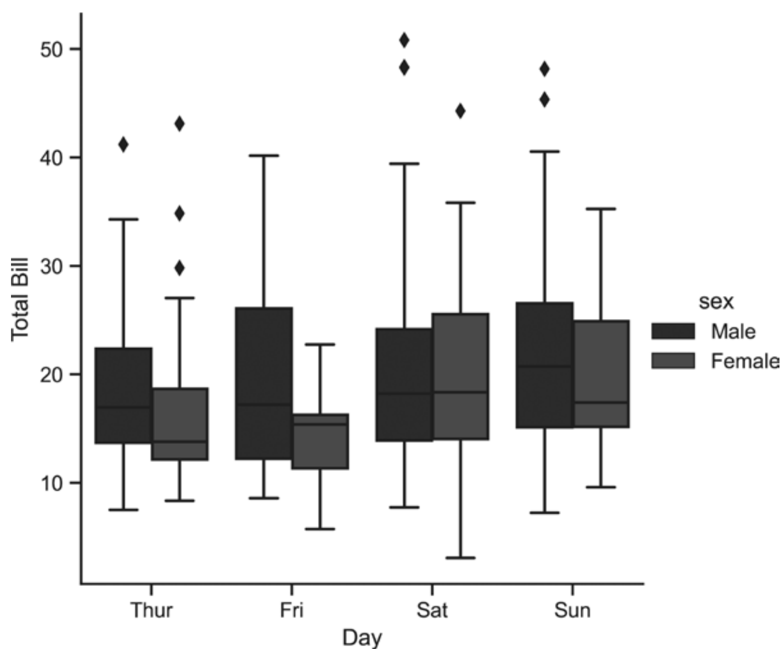


Рис. 36.6. Пример графика факторов со сравнением распределений при различных дискретных факторах

Совместные распределения

Аналогично графикам пар, которые мы рассмотрели выше, мы можем воспользоваться функцией `sns.jointplot` для отображения совместного распределения между различными наборами данных, а также соответствующих частных распределений (рис. 36.7):

```
In [10]: with sns.axes_style('white'):
             sns.jointplot(x="total_bill", y="tip", data=tips, kind='hex')
```

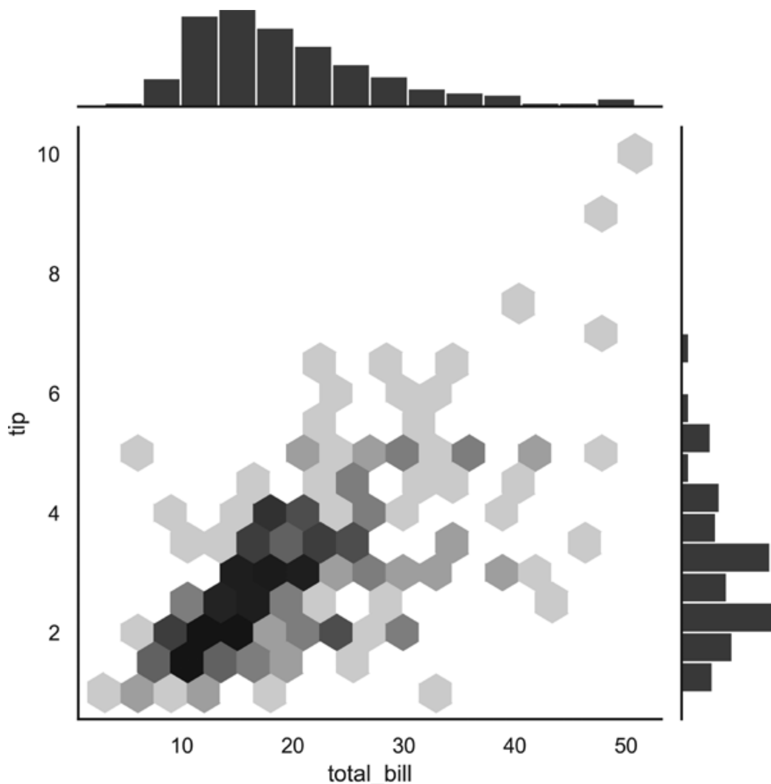


Рис. 36.7. График совместного распределения

Функция `jointplot` позволяет даже автоматически выполнить ядерную оценку плотности распределения и регрессию (рис. 36.8):

```
In [11]: sns.jointplot(x="total_bill", y="tip", data=tips, kind='reg');
```

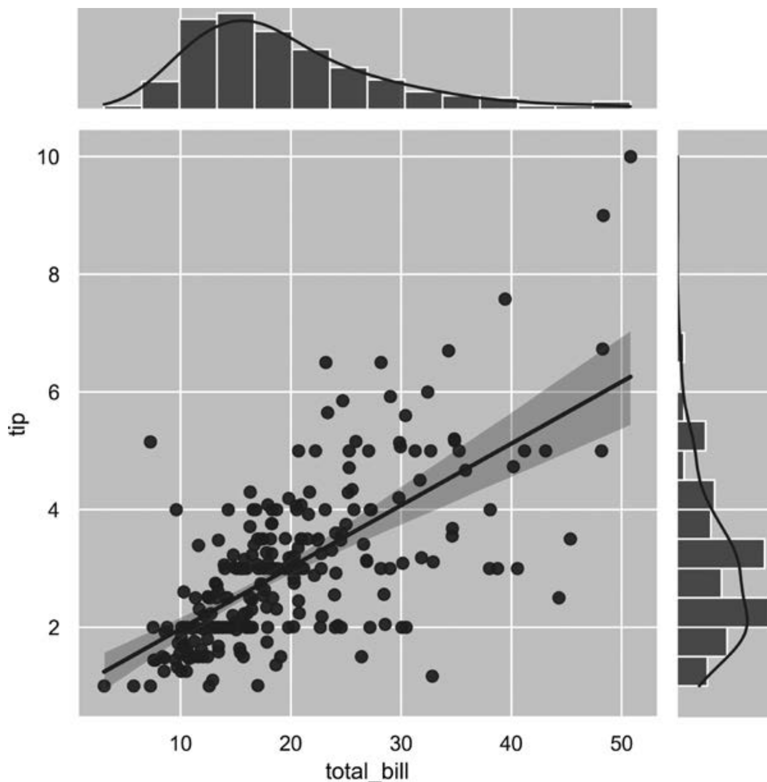


Рис. 36.8. График совместного распределения с подбором регрессии

Столбиковые диаграммы

Графики временных рядов можно строить с помощью функции `sns.factorplot`. В следующем примере, показанном на рис. 36.9, мы воспользуемся данными из набора Planets («Планеты»), которые мы уже анализировали в главе 20:

```
In [12]: planets = sns.load_dataset('planets')
         planets.head()
Out[12]:             method  number  orbital_period   mass  distance  year
         0  Radial Velocity       1         269.300   7.10     77.40  2006
         1  Radial Velocity       1         874.774   2.21     56.95  2008
         2  Radial Velocity       1         763.000   2.60     19.84  2011
         3  Radial Velocity       1         326.030  19.40    110.62  2007
         4  Radial Velocity       1         516.220  10.50    119.47  2009

In [13]: with sns.axes_style('white'):
             g = sns.catplot(x="year", data=planets, aspect=2, # Год
                             kind="count", color='steelblue')  # Количество
             g.set_xticklabels(step=5)
```

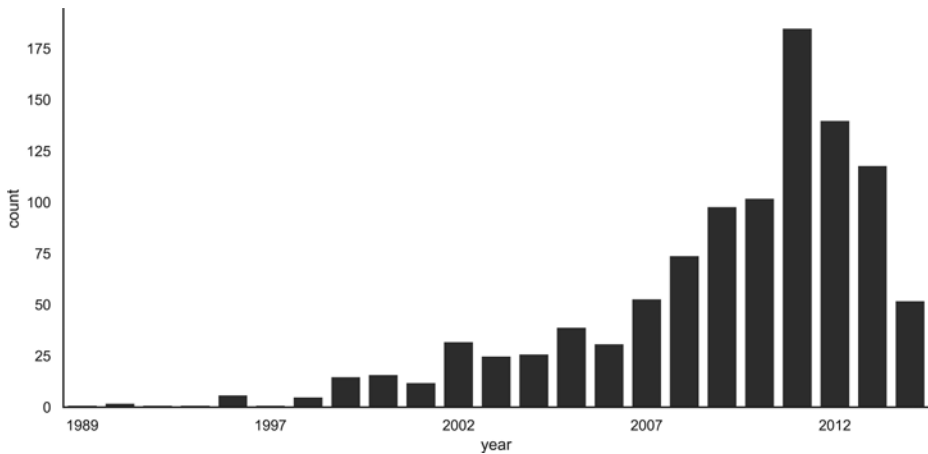


Рис. 36.9. Гистограмма как частный случай графика факторов

Мы можем узнать больше, проанализировав *метод*, с помощью которого была открыта каждая из планет, как показано на рис. 36.10:

```
In [14]: with sns.axes_style('white'):
             g = sns.catplot(x="year", data=planets, aspect=4.0, kind='count',
                             hue='method', order=range(2001, 2015))
             g.set_ylabels('Number of Planets Discovered')
             # Количество обнаруженных планет
```

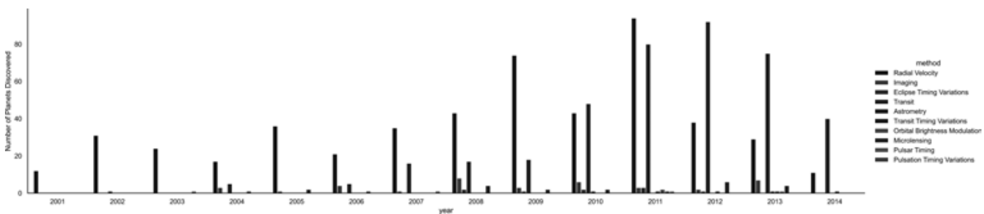


Рис. 36.10. Количество открытых планет по годам и методам

Дополнительную информацию о построении графиков с помощью библиотеки Seaborn можно найти в документации (https://oreil.ly/fCHxn) и галерее примеров (https://oreil.ly/08xGE).

Пример: время прохождения марафона

В этом разделе мы рассмотрим использование библиотеки Seaborn для визуализации и анализа данных по времени прохождения марафонской дистанции. Эти данные я собрал из различных интернет-источников, агрегировал, убрал все

данные, идентифицирующие отдельные личности, и выложил на GitHub, откуда их можно скачать[1].

Начнем со скачивания данных из Интернета и загрузки их в Pandas:

```
In [15]: # url = ('https://raw.githubusercontent.com/jakevdp/'
         #        'marathon-data/master/marathon-data.csv')
         # !cd data && curl -O {url}

In [16]: data = pd.read_csv('data/marathon-data.csv')
         data.head()
Out[16]:    age gender     split     final
         0   33      M  01:05:38  02:08:51
         1   32      M  01:06:26  02:09:28
         2   31      M  01:06:49  02:10:42
         3   38      M  01:06:16  02:13:45
         4   31      M  01:06:32  02:13:59
```

По умолчанию библиотека Pandas загружает столбцы с временем как строки Python (тип `object`), убедиться в этом можно, посмотрев значение атрибута `dtypes` объекта `DataFrame`:

```
In [17]: data.dtypes
Out[17]: age        int64
         gender    object
         split     object
         final     object
         dtype: object
```

Устраним этот недостаток, создав функцию для преобразования значений времени:

```
In [18]: import datetime

         def convert_time(s):
             h, m, s = map(int, s.split(':'))
             return datetime.timedelta(hours=h, minutes=m, seconds=s)

         data = pd.read_csv('data/marathon-data.csv',
                            converters={'split':convert_time, 'final':convert_time})
         data.head()
Out[18]:    age gender           split           final
         0   33      M 0 days 01:05:38 0 days 02:08:51
         1   32      M 0 days 01:06:26 0 days 02:09:28
         2   31      M 0 days 01:06:49 0 days 02:10:42
         3   38      M 0 days 01:06:16 0 days 02:13:45
         4   31      M 0 days 01:06:32 0 days 02:13:59
```

[1] Если вас интересует использование языка Python для веб-скрапинга, то рекомендую прочитать книгу *Web Scraping with Python* (https://oreil.ly/e3Xdg) Райана Митчелла (*Митчелл Р.* Современный скрапинг веб-сайтов с помощью Python. — СПб.: Питер, 2021).

```
In [19]: data.dtypes
Out[19]: age                int64
         gender            object
         split    timedelta64[ns]
         final    timedelta64[ns]
         dtype: object
```

Это упростит манипулирование значениями времени. Добавим столбцы с временем в секундах для использования утилитами Seaborn при построении графиков:

```
In [20]: data['split_sec'] = data['split'].view(int) / 1E9
         data['final_sec'] = data['final'].view(int) / 1E9
         data.head()
Out[20]:    age gender           split           final  split_sec  final_sec
         0   33      M 0 days 01:05:38 0 days 02:08:51     3938.0     7731.0
         1   32      M 0 days 01:06:26 0 days 02:09:28     3986.0     7768.0
         2   31      M 0 days 01:06:49 0 days 02:10:42     4009.0     7842.0
         3   38      M 0 days 01:06:16 0 days 02:13:45     3976.0     8025.0
         4   31      M 0 days 01:06:32 0 days 02:13:59     3992.0     8039.0
```

Чтобы получить некоторое представление о данных, нарисуем график с помощью `jointplot` (рис. 36.11):

```
In [21]: with sns.axes_style('white'):
             g = sns.jointplot(x='split_sec', y='final_sec', data=data, kind='hex')
             g.ax_joint.plot(np.linspace(4000, 16000),
                             np.linspace(8000, 32000), ':k')
```

Пунктирная линия показывает, каким было бы время прохождения всего марафона, если бы бегуны бежали его с неизменной скоростью. Распределение лежит выше этой прямой, и это значит (как и можно было ожидать), что большинство людей снижает скорость по мере прохождения дистанции. Если вы участвовали в марафонах, то знаете, что про бегунов, поступающих наоборот, то есть ускоряющихся во второй части дистанции, говорят, что они применяют обратное распределение сил.

Создадим в данных еще один столбец — коэффициент распределения, показывающий степень прямого и обратного распределения сил каждым бегуном:

```
In [22]: data['split_frac'] = 1 - 2 * data['split_sec'] / data['final_sec']
         data.head()
Out[22]:    age gender           split           final  split_sec  final_sec  \
         0   33      M 0 days 01:05:38 0 days 02:08:51     3938.0     7731.0
         1   32      M 0 days 01:06:26 0 days 02:09:28     3986.0     7768.0
         2   31      M 0 days 01:06:49 0 days 02:10:42     4009.0     7842.0
         3   38      M 0 days 01:06:16 0 days 02:13:45     3976.0     8025.0
         4   31      M 0 days 01:06:32 0 days 02:13:59     3992.0     8039.0

            split_frac
         0   -0.018756
         1   -0.026262
```

```
2   -0.022443
3    0.009097
4    0.006842
```

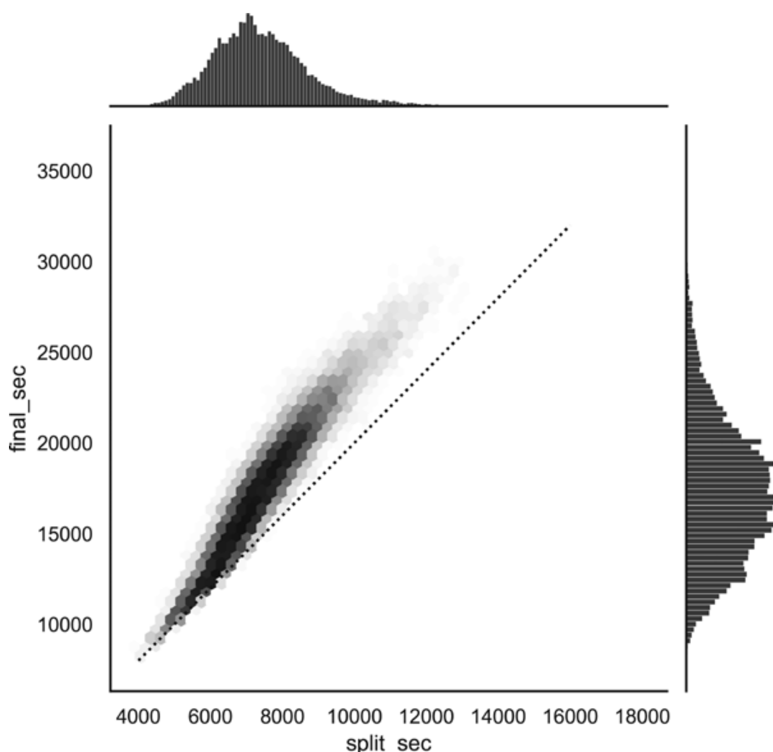


Рис. 36.11. Зависимости между временем прохождения первой половины марафона и всего марафона целиком

Если этот коэффициент меньше нуля, значит, спортсмен распределяет свои силы в обратной пропорции на соответствующую долю. Построим график распределения этого коэффициента (рис. 36.12):

```
In [23]: sns.displot(data['split_frac'], kde=False)
         plt.axvline(0, color="k", linestyle="--");

In [24]: sum(data.split_frac < 0)
Out[24]: 251
```

Из почти 40 000 участников только 250 человек распределяют свои силы на марафонской дистанции в обратной пропорции.

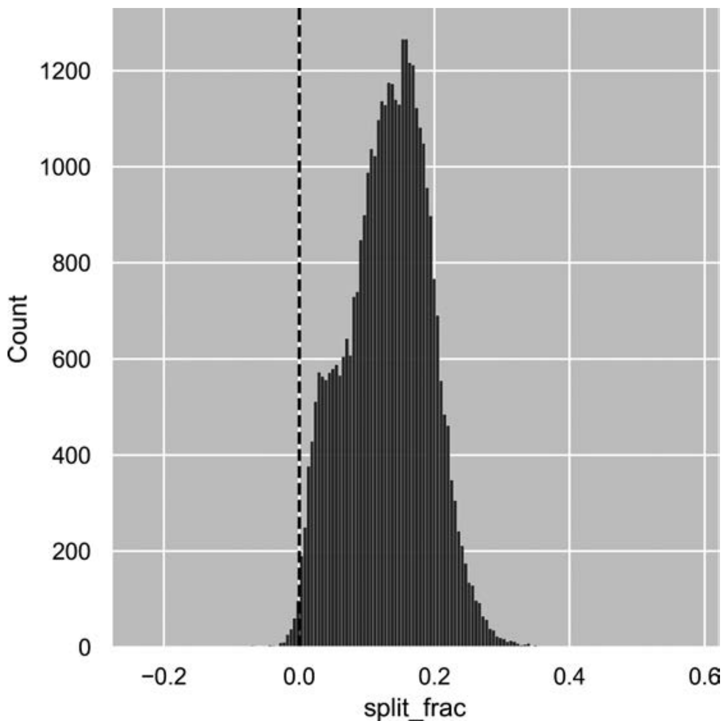


Рис. 36.12. Распределение коэффициентов для всех бегунов; 0.0 соответствует бегуну, пробежавшему первую и вторую половины марафона за одинаковое время

Выясним, существует ли какая-либо корреляция между коэффициентом распределения сил и другими переменными. Для построения графиков всех этих корреляций воспользуемся классом `PairGrid` (рис. 36.13):

```
In [25]: g = sns.PairGrid(data, vars=['age', 'split_sec', 'final_sec', 'split_frac'],
                          hue='gender', palette='RdBu_r')
         g.map(plt.scatter, alpha=0.8)
         g.add_legend();
```

Похоже, что коэффициент распределения сил никак не коррелирует с возрастом, но коррелирует с итоговым временем забега: более быстрые бегуны склонны распределять свои силы поровну. Также было бы интересно посмотреть различие между мужчинами и женщинами. Построим гистограмму коэффициентов распределения сил для этих двух групп (рис. 36.14):

```
In [26]: sns.kdeplot(data.split_frac[data.gender=='M'], label='men', shade=True)
         sns.kdeplot(data.split_frac[data.gender=='W'], label='women', shade=True)
         plt.xlabel('split_frac');
```

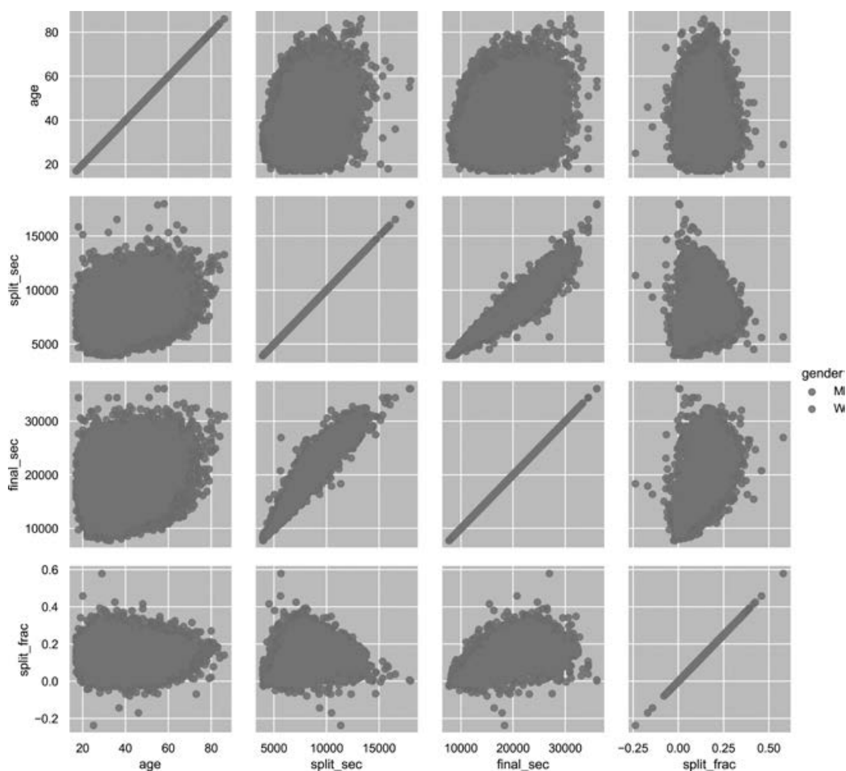


Рис. 36.13. Зависимости между величинами в «марафонском» наборе данных

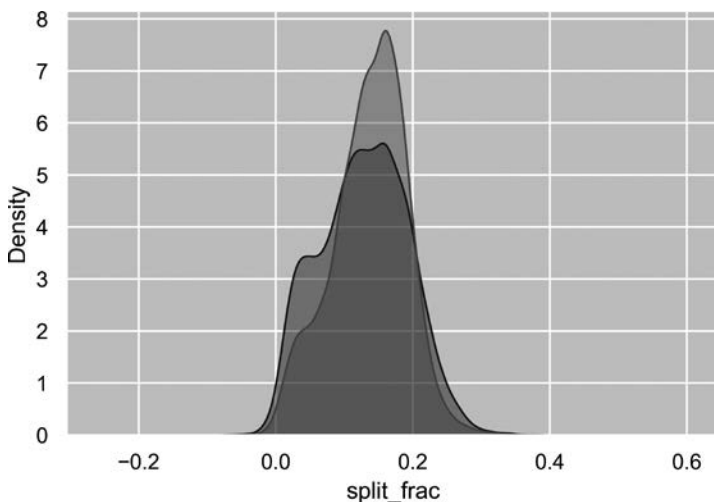


Рис. 36.14. Распределение коэффициентов для бегунов в зависимости от пола

Интересно отметить, что мужчин, распределяющих свои силы почти поровну, намного больше, чем женщин! График выглядит практически как какое-то бимодальное распределение по мужчинам и женщинам. Удастся ли нам разобраться, в чем дело, взглянув на эти распределения как на функцию возраста?

Удобный способ сравнения распределений — использование так называемой *скрипичной диаграммы* (рис. 36.15):

```
In [27]: sns.violinplot(x="gender", y="split_frac", data=data,
                        palette=["lightblue", "lightpink"]);
```

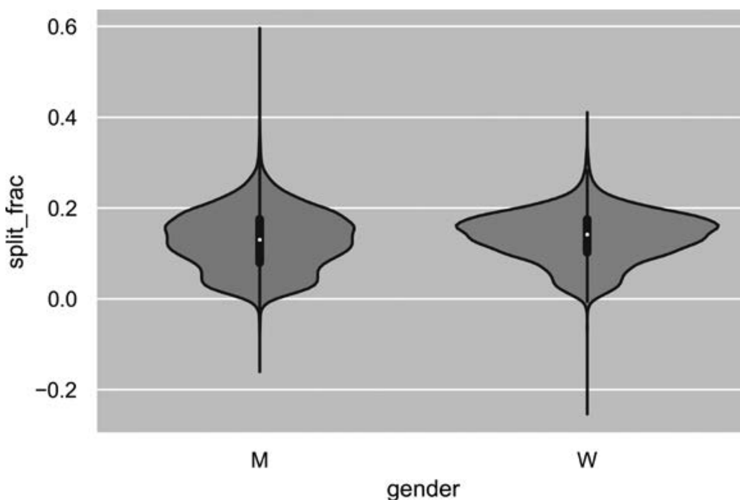


Рис. 36.15. «Скрипичная» диаграмма, показывающая зависимость коэффициента распределения сил от пола

Заглянем чуть глубже и сравним эти «скрипичные» диаграммы как функцию возраста (рис. 36.16). Начнем с создания в массиве нового столбца, отражающего возраст бегуна с точностью до десятилетия:

```
In [28]: data['age_dec'] = data.age.map(lambda age: 10 * (age // 10))
         data.head()
Out[28]:    age gender           split           final  split_sec  final_sec  \
         0   33      M 0 days 01:05:38 0 days 02:08:51     3938.0     7731.0
         1   32      M 0 days 01:06:26 0 days 02:09:28     3986.0     7768.0
         2   31      M 0 days 01:06:49 0 days 02:10:42     4009.0     7842.0
         3   38      M 0 days 01:06:16 0 days 02:13:45     3976.0     8025.0
         4   31      M 0 days 01:06:32 0 days 02:13:59     3992.0     8039.0

            split_frac  age_dec
         0   -0.018756       30
         1   -0.026262       30
```

```
        2   -0.022443       30
        3    0.009097       30
        4    0.006842       30

In [29]: men = (data.gender == 'M')
         women = (data.gender == 'W')

         with sns.axes_style(style=None):
             sns.violinplot(x="age_dec", y="split_frac", hue="gender", data=data,
                            split=True, inner="quartile",
                            palette=["lightblue", "lightpink"]);
```

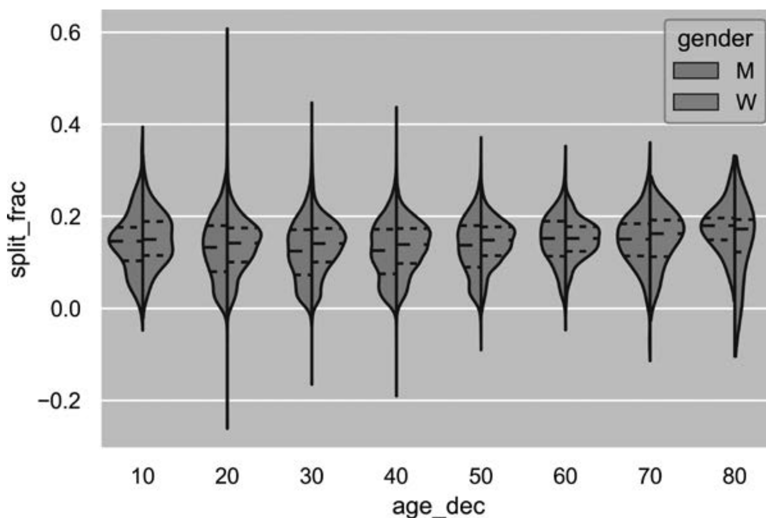


Рис. 36.16. «Скрипичная» диаграмма, показывающая зависимость коэффициента распределения сил от пола и возраста

Этот график наглядно показывает, как распределяют свои силы мужчины и женщины. Мужчины в возрасте от 20 до 50 лет явно склонны более равномерно распределять силы, чем женщины того же возраста (или вообще любого возраста).

И на удивление 80-летние женщины, похоже, обошли *всех* в смысле равномерности распределения сил. Вероятно, дело в том, что мы оцениваем распределение на малых количествах, ведь бегунов такого возраста всего несколько:

```
In [30]: (data.age > 80).sum()
Out[30]: 7
```

Но вернемся к мужчинам с обратным распределением сил: кто эти бегуны? Существует ли корреляция между обратным распределением сил и скоростью преодоления марафонской дистанции в целом? Для ответа на этот вопрос можно построить соот-

ветствующий график. Воспользуемся функцией `regplot`, автоматически выполняющей подгонку параметров линейной регрессии под имеющиеся данные (рис. 36.17):

```
In [31]: g = sns.lmplot(x='final_sec', y='split_frac', col='gender', data=data,
                        markers=".", scatter_kws=dict(color='c'))
         g.map(plt.axhline, y=0.0, color="k", ls=":");
```

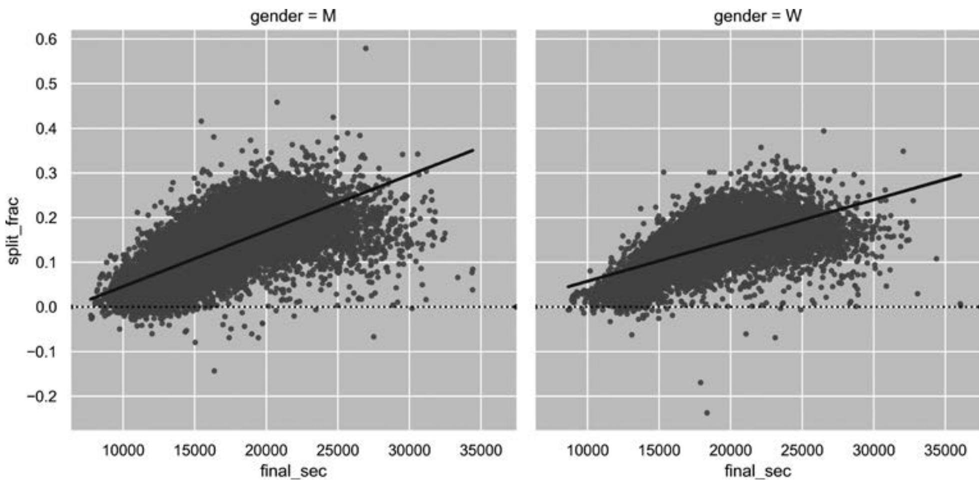


Рис. 36.17. Зависимость коэффициента распределения сил от времени преодоления дистанции

Как видите, люди (и мужчины, и женщины), равномерно распределяющие свои силы, как правило, преодолевают дистанцию быстрее — они финишируют в пределах 15 000 секунд (это примерно 4 часа). Более медленные бегуны менее склонны к подобному распределению сил.

Дополнительные источники информации

Бессмысленно надеяться охватить в одной части книги все имеющиеся в Matplotlib возможности и типы графиков. Как и для других рассмотренных нами пакетов, разумное использование функции автодополнения клавишей Tab и средств получения справки в оболочке IPython (см. главу 1) может принести немалую пользу при изучении Matplotlib API. Кроме того, полезным источником информации может стать онлайн-документация Matplotlib (http://matplotlib.org/), в частности галерея Matplotlib (https://oreil.ly/WNiHP). В ней содержатся миниатюры сотен различных типов графиков, каждая из которых представляет собой ссылку на страницу с фрагментом кода на языке Python, который использовался для ее генерации. Таким образом, вы можете визуально изучить широкий диапазон различных стилей построения графиков и методик визуализации.

В качестве более обширного обзора библиотеки Matplotlib я бы рекомендовал обратить внимание на книгу *Interactive Applications Using Matplotlib*, написанную разработчиком ядра Matplotlib Беном Рутом.

Другие графические библиотеки для Python

Matplotlib — наиболее известная из библиотек визуализации для Python, однако существуют и другие, более современные инструменты, заслуживающие пристального внимания. Я перечислю некоторые из них.

- Bokeh (http://bokeh.pydata.org/) — JavaScript-библиотека визуализации с клиентской частью для Python, предназначенная для создания высокоинтерактивных визуализаций с возможностью обработки очень больших и/или потоковых наборов данных.
- Plotly (http://plot.ly/) — продукт с открытым исходным кодом одноименной компании, аналогичный по духу библиотеке Bokeh. Эта библиотека активно развивается и позволяет создавать широкий диапазон типов диаграмм.
- HoloViews (https://holoviews.org/) — более декларативный и унифицированный API для создания диаграмм с использованием различных библиотек, включая Bokeh и Matplotlib.
- Vega (https://vega.github.io/) и Vega-Lite (https://vega.github.io/vega-lite) — декларативные графические форматы, являющиеся продуктом многолетних исследований в области визуализации и интерактивного взаимодействия с данными. Эталонная реализация написана на языке JavaScript, но вообще их API не зависит от языка, а реализация на Python доступна в виде пакета Altair (http://altair-viz.github.io/).

Ландшафт визуализации в мире Python меняется очень динамично, и я уверен, что этот список устареет сразу после публикации. Кроме того, поскольку Python используется во многих областях, вам встретятся многие другие инструменты визуализации, созданные для более конкретных случаев. Часто бывает трудно уследить за всем, поэтому я рекомендую обратить внимание на PyViz (https://pyviz.org/) — отличный ресурс, который поможет вам в изучении широкого спектра инструментов визуализации. Этот поддерживаемый сообществом сайт содержит учебные пособия и примеры применения различных инструментов визуализации.

ЧАСТЬ V

Машинное обучение

Эта заключительная часть знакомит с очень широкой темой машинного обучения, используя пакет Scikit-Learn (http://scikit-learn.org/) как основной инструмент. Вы можете думать о машинном обучении как о классе алгоритмов, позволяющих программам обнаруживать закономерности в наборе данных и тем самым «учиться» делать выводы на основе данных. Это не всестороннее введение в машинное обучение, поскольку это обширная тема, требующая более формализованного подхода. Это также не исчерпывающее руководство по использованию пакета Scikit-Learn (такие руководства вы можете найти в разделе «Дополнительные источники информации по машинному обучению» главы 50). Задачи этой части — познакомить читателя:

- с базовой терминологией и понятиями машинного обучения;
- с API библиотеки Scikit-Learn и некоторыми примерами его использования;
- с подробностями нескольких наиболее важных методов машинного обучения, помочь понять, как они работают, а также где и когда применимы.

Большая часть материала взята из учебных курсов по Scikit-Learn, а также семинаров, проводившихся мной на PyCon, SciPy, PyData и других конференциях. Многолетние отзывы участников и других докладчиков семинаров позволили сделать изложение материала более доходчивым!

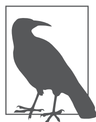
ГЛАВА 37

Что такое машинное обучение

Прежде чем углубиться в детали различных методов машинного обучения, давайте выясним, что такое машинное обучение. Машинное обучение часто рассматривается как подобласть искусственного интеллекта. Однако такая классификация, как мне кажется, вводит в заблуждение. Исследования в области машинного обучения возникли на основе научных разработок в этой сфере, но в контексте применения методов машинного обучения в науке о данных полезнее рассматривать машинное обучение как *средство создания моделей данных*.

Задачи «обучения» начинаются с появлением у этих моделей *настраиваемых параметров*, которые можно адаптировать для отражения наблюдаемых данных, таким образом, программа как бы «обучается» на данных. После обучения модели на имеющихся данных ее можно использовать для предсказания и интерпретации различных аспектов данных новых наблюдений. Оставлю читателю в качестве самостоятельного задания обдумать философский вопрос о том, насколько подобное математическое «обучение», основанное на моделях, схоже с «обучением» человеческого мозга.

Для эффективного использования этих инструментов важно понимать общую формулировку задачи машинного обучения, поэтому начнем с широкой классификации типов подходов, которые мы будем обсуждать.



Все рисунки в этой главе созданы на основе реальных вычислений; код, стоящий за ними, можно найти в онлайн-приложении (https://oreil.ly/o1Zya).

Категории машинного обучения

На базовом уровне машинное обучение можно разделить на два основных типа: обучение с учителем и обучение без учителя.

Машинное обучение с учителем (supervised learning) — включает моделирование отношений между измеренными признаками данных и соответствующими им метками. После выбора модели ее можно использовать для присваивания меток новым, неизвестным ранее данным. Далее этот вид машинного обучения разделяется на задачи классификации и задачи регрессии. В задачах *классификации* метки определяют дискретные категории, а в задачах *регрессии* — непрерывные величины. Мы рассмотрим примеры обоих типов машинного обучения с учителем в следующем разделе.

Машинное обучение без учителя (unsupervised learning) — включает моделирование признаков набора данных без каких-либо меток и описывается фразой «Пусть набор данных говорит сам за себя». Эти модели включают такие задачи, как *кластеризация* (clustering) и *понижение размерности* (dimensionality reduction). Алгоритмы кластеризации служат для выделения отдельных групп данных, а алгоритмы понижения размерности — для поиска более сжатых представлений данных. Мы рассмотрим примеры обоих типов машинного обучения без учителя в следующем разделе.

Кроме того, существуют так называемые методы *частичного обучения* (semi-supervised learning), располагающиеся примерно посередине между машинным обучением с учителем и машинным обучением без учителя. Методы частичного обучения полезны при наличии лишь неполного набора меток.

Качественные примеры прикладных задач машинного обучения

Чтобы конкретизировать вышеописанные понятия, рассмотрим несколько простых примеров задач машинного обучения. Их цель — дать общее представление о разновидностях машинного обучения, с которыми мы столкнемся в этой главе. В следующих главах мы рассмотрим подробнее соответствующие модели и их использование. Чтобы получить представление о технических аспектах, загляните в исходный код на Python, с помощью которого были сгенерированы иллюстрации, который доступен в онлайн-приложении (https://oreil.ly/o1Zya).

Классификация: предсказание дискретных меток

Рассмотрим простую задачу классификации: имеется набор точек с метками и требуется классифицировать некоторое количество точек без меток.

Допустим, у нас есть данные, показанные на рис. 37.1. Наши данные двумерны, то есть для каждой точки заданы два *признака* (feature), которым соответствуют координаты (x, y) точки на плоскости. Кроме того, каждой точке поставлена в соответствие одна из двух *меток класса* (class label), представленная определенным цветом. Нам требуется на основе этих признаков и меток создать модель, с помощью которой мы смогли бы установить, должна ли новая точка быть «синей» или «красной».

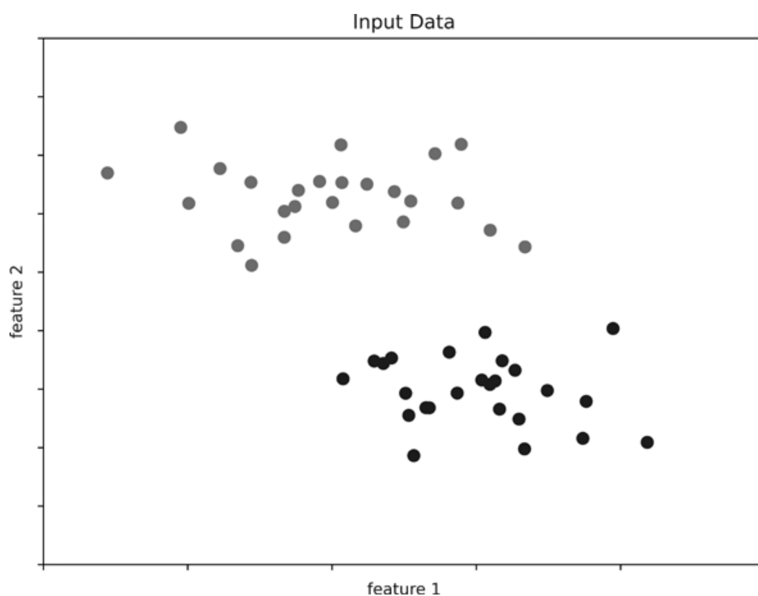


Рис. 37.1. Простой набор данных для классификации

Существует множество возможных моделей для решения подобной задачи классификации, но мы воспользуемся исключительно простой моделью. Будем исходить из допущения, что наши две группы можно разделить прямой линией на плоскости, так что точки с одной стороны прямой будут принадлежать к одной группе. Данная *модель* представляет собой количественное выражение утверждения «прямая линия разделяет классы», в то время как *параметры модели* представляют собой конкретные числа, описывающие местоположение и направленность этой прямой для наших данных. Оптимальные значения этих параметров модели получаются

с помощью обучения на имеющихся данных (это и есть «обучение» в смысле машинного обучения), часто называемого *обучением модели* (training the model).

На рис. 37.2 показано, как выглядит модель, обученная на наших данных.

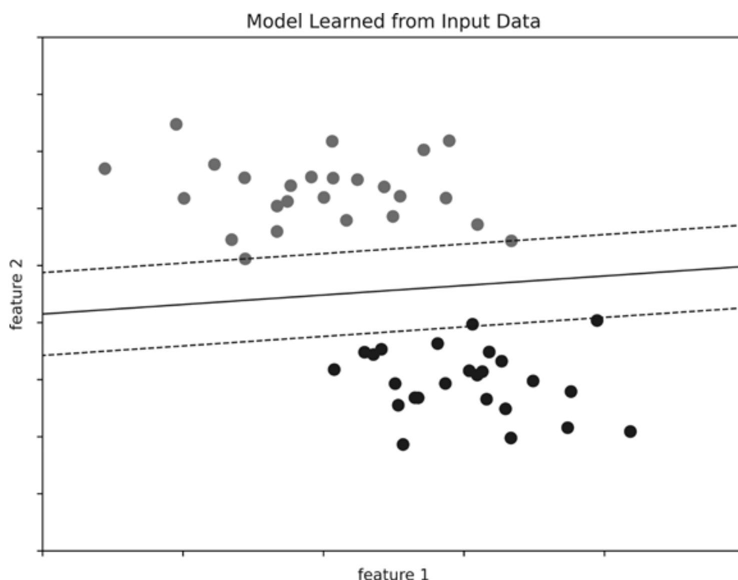


Рис. 37.2. Простая модель классификации

После обучения модель можно попробовать применить для классификации новых, немаркированных данных. Другими словами, можно взять другой набор данных, провести прямую модели через них и на основе этой прямой присвоить метки новым точкам (рис. 37.3). Этот этап обычно называют *предсказанием* (prediction).

Такова основная идея задачи классификации в машинном обучении, причем слово «классификация» означает, что метки классов дискретны. На первый взгляд эта задача может показаться довольно простой: нет ничего сложного в том, чтобы просто посмотреть на данные и провести подобную разделяющую прямую для классификации данных. Достоинство подхода машинного обучения состоит в том, что его результаты можно обобщить на большие наборы данных и большее количество измерений. Например, задачу классификации можно применить для автоматического обнаружения спама в электронной почте. В этом случае можно использовать следующие признаки и метки:

- *признак 1*, *признак 2* и т. д. → нормированные количества ключевых слов или фраз («Виагра», «Расширенная гарантия» и т. д.);
- *метка* → «спам» или «не спам».

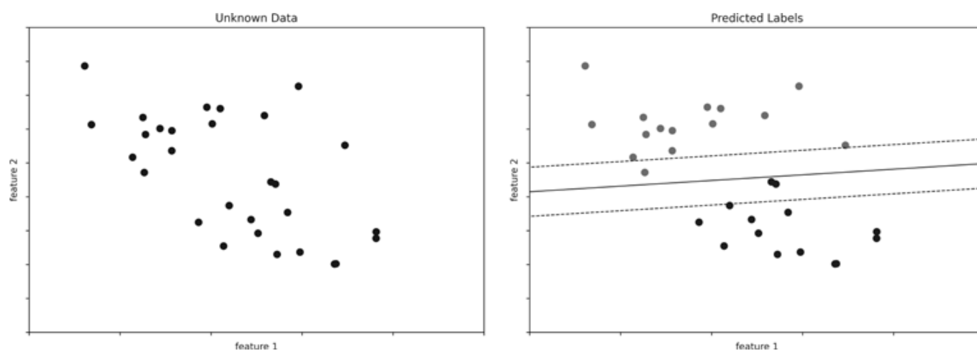


Рис. 37.3. Применение модели для классификации новых данных

В обучающем наборе эти метки задаются путем индивидуального осмотра небольшой репрезентативной выборки сообщений электронной почты, для остальных сообщений электронной почты метка будет определяться с помощью модели. При обученном соответствующим образом алгоритме классификации с достаточно хорошо сконструированными признаками (обычно тысячи или миллионы слов/фраз) этот тип классификации может оказаться весьма эффективным. Мы рассмотрим пример подобной классификации текста в главе 41.

Некоторые важные алгоритмы классификации, которые мы обсудим более подробно, — это гауссов наивный байесовский классификатор (глава 41), метод опорных векторов (глава 43) и классификация на основе случайных лесов (глава 44).

Регрессия: предсказание непрерывных меток

В отличие от дискретных меток, с которыми мы имели дело в алгоритмах классификации, сейчас мы рассмотрим простую задачу регрессии, где метки представляют собой непрерывные величины.

Изучим показанные на рис. 37.4 данные, состоящие из набора точек с непрерывными метками.

Как и в примере классификации, наши данные двумерны, то есть каждая точка описывается двумя признаками. Непрерывные метки точек представлены их цветом.

Существует много возможных моделей регрессии, подходящих для таких данных, но мы для предсказания меток точек воспользуемся простой линейной регрессией. Модель простой линейной регрессии основана на допущении, что, если рассматривать метки как третье пространственное измерение, можно подобрать для этих данных разделяющую плоскость. Это высокоуровневое обобщение хорошо известной задачи подбора разделяющей прямой для данных с двумя координатами.

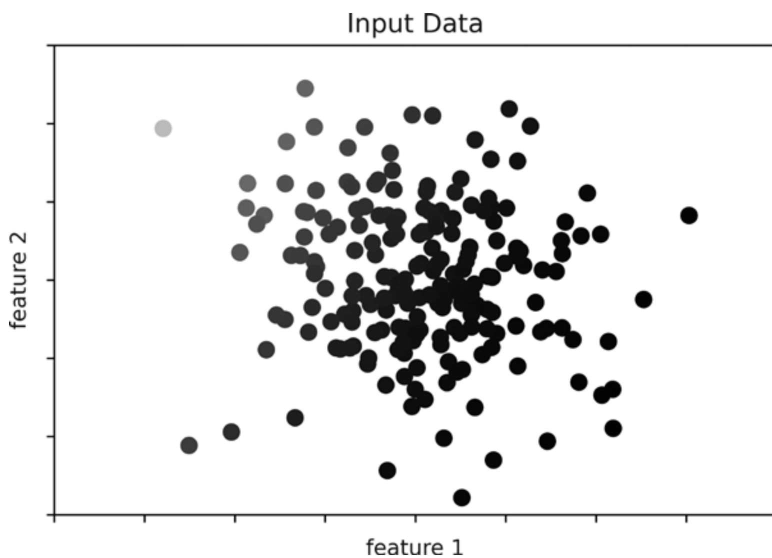


Рис. 37.4. Простой набор данных для регрессии

Визуализировать это можно так, как показано на рис. 37.5.

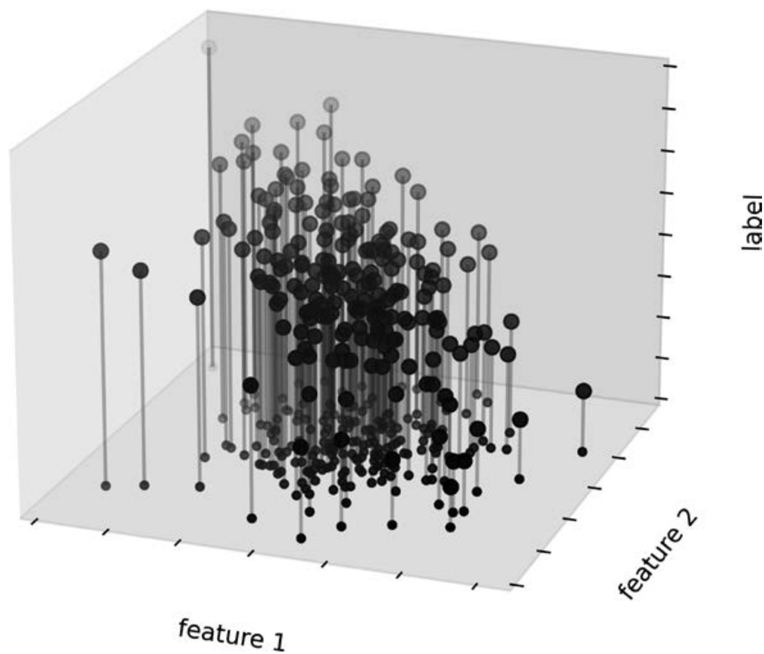


Рис. 37.5. Трехмерное представление данных для демонстрации регрессии

Обратите внимание, что плоскость *«признак 1 — признак 2»* здесь подобна двумерному графику на рис. 37.4. Однако в данном случае метки представлены как цветом, так и местоположением вдоль третьей оси координат. Логично, что подбор разделяющей плоскости для этих трехмерных данных позволит нам предсказывать будущие метки для любых входных параметров. Вернувшись к двумерной проекции, после подбора подобной плоскости мы получим результат, показанный на рис. 37.6.

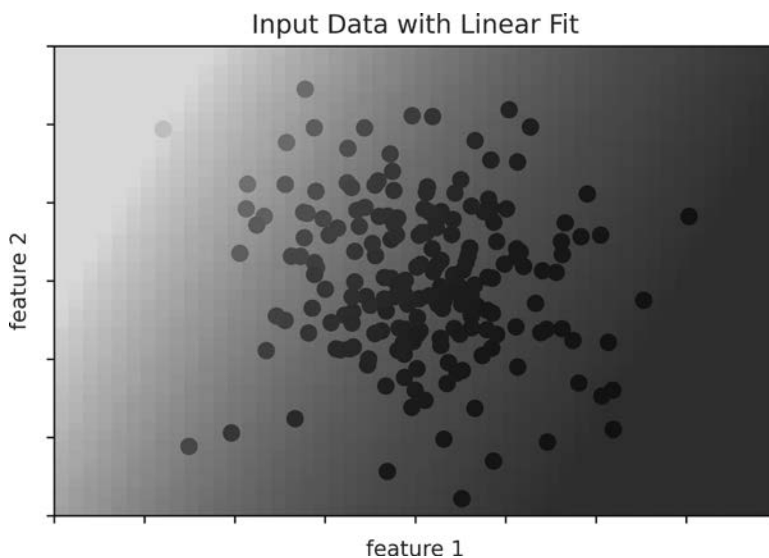


Рис. 37.6. Визуальное представление модели регрессии

Полученная плоскость дает нам все, что нужно для предсказания меток новых точек. Графически наши результаты будут выглядеть так, как показано на рис. 37.7.

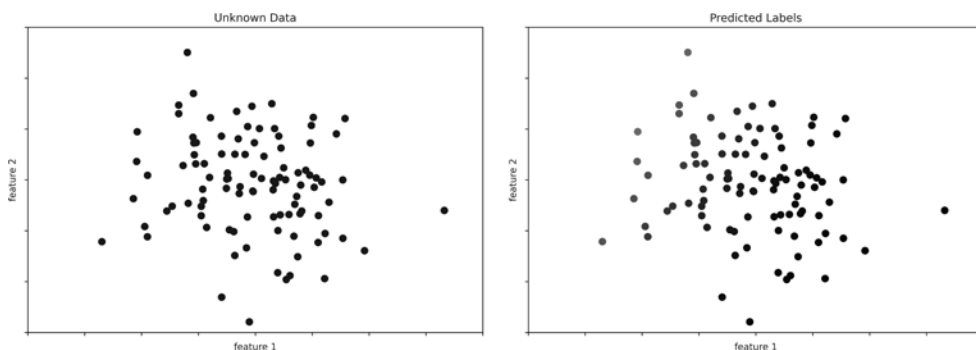


Рис. 37.7. Применение модели регрессии к новым данным

Как и в примере классификации, при небольшом количестве измерений эта задача может показаться тривиально простой. Однако сила этих методов заключается как раз в том, что их можно использовать и для данных со множеством признаков. Например, рассматриваемый пример подобен задаче вычисления расстояний до наблюдаемых в телескоп галактик — в данном случае можно использовать следующие признаки и метки:

- *признак 1*, *признак 2* и т. д. → яркость каждой галактики на одной из нескольких длин волн (цветов);
- *метка* → расстояние до галактики или ее красное смещение.

Можно вычислить расстояния до небольшого числа этих галактик на основании независимого набора (обычно более дорогостоящих) наблюдений. После этого можно оценить расстояния до оставшихся галактик с помощью подходящей модели регрессии, вместо того чтобы использовать более дорогостоящие наблюдения для всего набора галактик. В астрономических кругах эта задача известна под названием «задачи фотометрии красного смещения».

Далее в этой главе мы обсудим такие важные регрессионные алгоритмы, как линейная регрессия (см. главу 42), метод опорных векторов (см. главу 43) и регрессия на основе случайных лесов (см. главу 44).

Кластеризация: определение меток для немаркированных данных

Классификация и регрессия, которые мы только что рассмотрели, — это примеры алгоритмов обучения с учителем, создающих модель на основе имеющихся данных для предсказания меток новых данных. Обучение без учителя строит модели для описания данных безотносительно каких-либо известных меток.

Часто встречающийся случай машинного обучения без учителя — «кластеризация», когда данные автоматически распределяются по некоторому количеству дискретных групп. Например, наши двумерные данные могли бы оказаться такими, как показаны на рис. 37.8.

Визуально очевидно, что каждая из этих точек относится к одной из нескольких групп. При таких входных данных модель кластеризации определит на основе их внутренней структуры, какие точки и какой группе принадлежат. Воспользовавшись очень быстрым и интуитивно понятным алгоритмом кластеризации методом k средних (см. главу 47), мы получаем кластеры, изображенные на рис. 37.9.

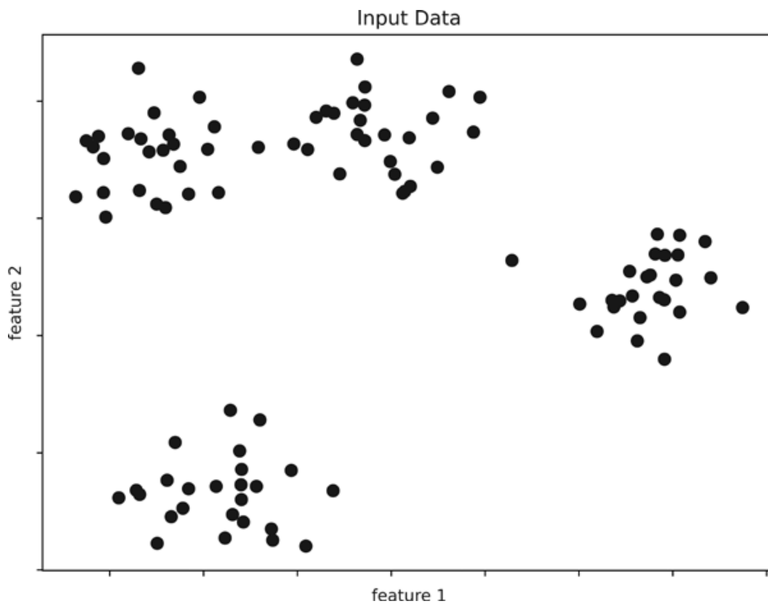


Рис. 37.8. Пример данных для демонстрации кластеризации

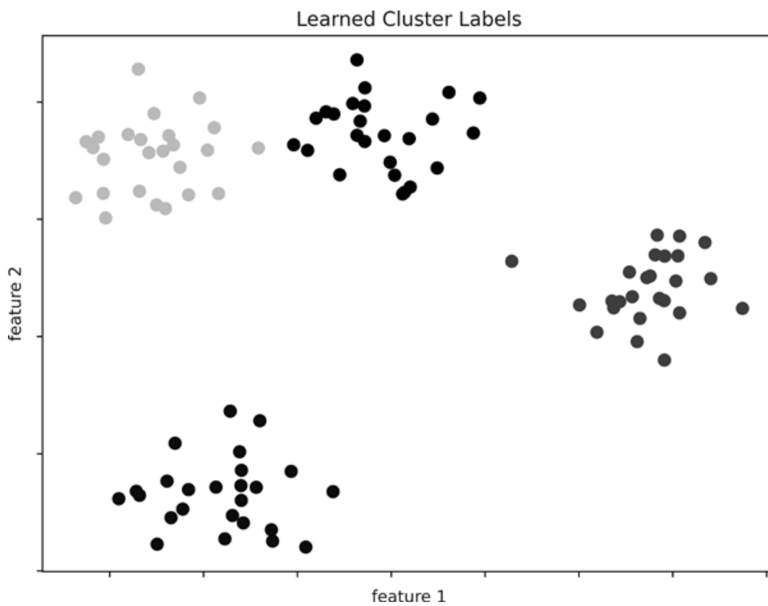


Рис. 37.9. Данные, маркированные
с помощью модели кластеризации методом k средних

Метод k средних обучает модель, состоящую из k центров кластеров. Наилучшими считаются центры, расстояние от которых до точек, соответствующих им, минимально. В случае двух измерений эта задача может показаться тривиальной, но по мере усложнения данных и увеличения их объема подобные алгоритмы кластеризации можно использовать для извлечения из набора данных полезной информации.

Более подробно алгоритм k средних мы обсудим в главе 47. Среди других важных алгоритмов кластеризации можно назвать: смесь гауссовых распределений (см. главу 48) и спектральную кластеризацию (см. документацию библиотеки Scikit-Learn, https://oreil.ly/9FHKO).

Понижение размерности: определение структуры немаркированных данных

Понижение размерности — еще один пример алгоритма обучения без учителя, который определяет метки и другую информацию, исходя из структуры самого набора данных. Алгоритм понижения размерности несколько труднее для понимания, чем рассмотренные нами ранее примеры. Его суть заключается в попытке получить представление меньшей размерности, сохраняющее основные качества полного набора данных. Различные алгоритмы понижения размерности оценивают существенность этих качеств по-разному, как мы увидим далее в главе 46.

В качестве примера рассмотрим данные, показанные на рис. 37.10.

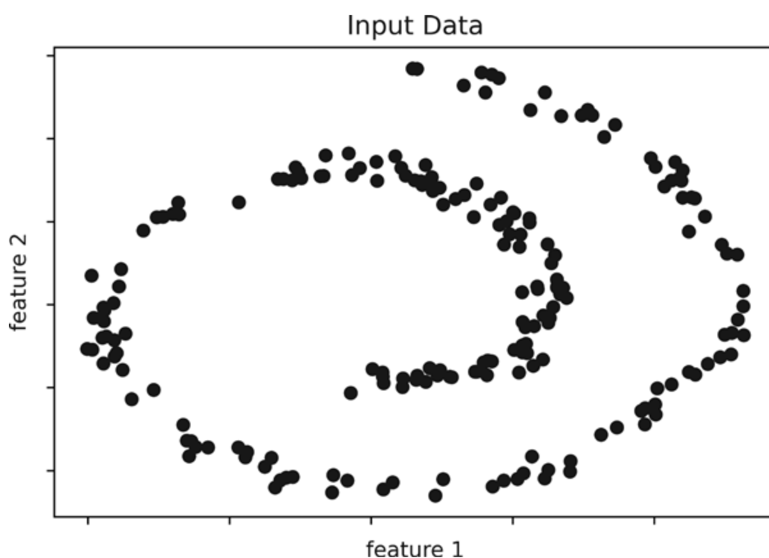


Рис. 37.10. Пример данных для демонстрации понижения размерности

Зрительно очевидно, что эти данные имеют внутреннюю структуру: они получены из одномерной прямой, закрученной в двумерном пространстве в спираль. В некотором смысле можно сказать, что эти данные по своей внутренней сути одномерны, но вложены в пространство большей размерности. Подходящая модель понижения размерности в таком случае должна учитывать эту нелинейную вложенную структуру, чтобы получить из нее представление более низкой размерности.

На рис. 37.11 показаны результаты работы алгоритма обучения на базе многообразий Isomap, который именно это и делает.

Обратите внимание, что цвета, отражающие значения полученной одномерной скрытой переменной, меняются равномерно по ходу спирали, а значит, алгоритм действительно распознает видимую на глаз структуру. Масштаб возможностей алгоритмов понижения размерности становится очевиднее в случаях более высоких размерностей. Например, нам может понадобиться визуализировать важные зависимости в наборе данных, имеющих от 100 до 1000 признаков. Визуализация 1000-мерных данных — непростая задача, один из способов решения которой — использование методик понижения размерности, чтобы свести набор признаков к двум или трем измерениям.

Среди других важных алгоритмов понижения размерности — метод главных компонент (см. главу 45) и различные методы обучения на базе многообразий, включая Isomap и локально линейное вложение (см. главу 46).

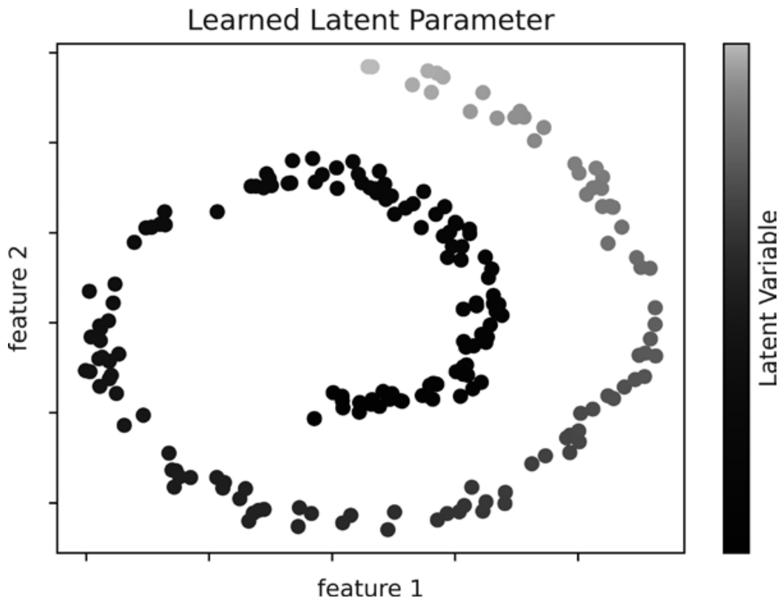


Рис. 37.11. Данные с метками, полученными с помощью алгоритма понижения размерности

Резюме

Мы рассмотрели несколько простых примеров основных методов машинного обучения. На практике существует множество важных нюансов, которые мы обошли вниманием, однако главной целью этой главы было дать лишь общий обзор задач, решаемых с помощью методов машинного обучения.

Мы рассмотрели:

- *обучение с учителем* — модели для предсказания меток на основе маркированных обучающих данных:
  - *классификацию* — модели для предсказания меток из двух или более отдельных категорий;
  - *регрессию* — модели для предсказания непрерывных меток;
- *обучение без учителя* — модели для выявления структуры немаркированных данных:
  - *кластеризацию* — модели для выявления и распознавания в данных отдельных групп;
  - *понижение размерности* — модели для выявления и распознавания низкоразмерной структуры в многомерных данных.

В следующих главах мы изучим данные категории подробнее и рассмотрим более интересные примеры использования этих принципов.

ГЛАВА 38

Знакомство с библиотекой Scikit-Learn

Существует несколько библиотек для Python с надежными реализациями широкого диапазона алгоритмов машинного обучения. Одна из самых известных — Scikit-Learn (http://scikit-learn.org/), пакет, предоставляющий эффективные версии многих известных алгоритмов. Пакет Scikit-Learn отличают аккуратный, единообразный и довольно простой API, а также удобная и всеохватывающая онлайн-документация. Преимущество этого единообразия в том, что, разобравшись в основах использования и синтаксисе Scikit-Learn для одного типа моделей, вы сможете легко перейти к другой модели или алгоритму.

В этой главе вы найдете обзор API библиотеки Scikit-Learn. Полное понимание элементов API — основа для более углубленного практического обсуждения алгоритмов и методов машинного обучения в следующих главах.

Начнем с представления данных в библиотеке Scikit-Learn, затем рассмотрим Estimator API (API статистического оценивания) и, наконец, взглянем на интересный пример использования этих инструментов для исследования набора изображений рукописных цифр.

Представление данных в Scikit-Learn

Машинное обучение связано с созданием моделей на основе данных, поэтому начнем с обсуждения понятного компьютеру представления данных. Лучше всего представлять используемые в библиотеке Scikit-Learn данные в виде *таблиц*.

Простейшая таблица — двумерная сетка данных, в которой строки представляют отдельные элементы набора данных, а столбцы — признаки, характеризующие каждый из этих элементов. Например, рассмотрим набор данных Iris (https://oreil.ly/TeWYs), проанализированный Рональдом Фишером в 1936 году. Скачаем его

в виде объекта `DataFrame` библиотеки Pandas с помощью библиотеки Seaborn (http://seaborn.pydata.org/) и рассмотрим несколько первых элементов:

```
In [1]: import seaborn as sns
        iris = sns.load_dataset('iris')
        iris.head()
Out[1]:    sepal_length  sepal_width  petal_length  petal_width species
        0           5.1          3.5           1.4          0.2  setosa
        1           4.9          3.0           1.4          0.2  setosa
        2           4.7          3.2           1.3          0.2  setosa
        3           4.6          3.1           1.5          0.2  setosa
        4           5.0          3.6           1.4          0.2  setosa
```

Каждая строка относится к одному из измеренных цветков, а количество строк равно полному количеству цветков в наборе данных. Мы будем называть строки этой матрицы *образцами* (samples), а количество строк полагать равным `n_samples`.

Каждый столбец данных относится к конкретному количественному показателю, описывающему каждый образец. Мы будем называть столбцы матрицы *признаками* (features), а количество столбцов полагать равным `n_features`.

Матрица признаков

Из устройства таблицы очевидно, что информацию можно рассматривать как двумерный числовой массив или матрицу, которую мы будем называть *матрицей признаков* (features matrix). По традиции матрицу признаков часто хранят в переменной `X`. Предполагается, что матрица признаков — двумерная, с формой `[n_samples, n_features]`, и хранят ее чаще всего в массиве NumPy или объекте `DataFrame` библиотеки Pandas, хотя некоторые модели библиотеки Scikit-Learn допускают использование также разреженных матриц из библиотеки SciPy.

Образцы (то есть строки) всегда соответствуют отдельным объектам, описываемым набором данных. Например, образец может быть цветком, человеком, документом, изображением, звуковым файлом, видеофайлом, астрономическим объектом или чем-то еще, что можно описать с помощью набора количественных измерений.

Признаки (то есть столбцы) всегда соответствуют конкретным наблюдениям, количественно описывающим каждый из образцов. Значения признаков обычно представлены вещественными числами, но в некоторых случаях они могут быть булевыми или дискретными значениями.

Целевой массив

Помимо матрицы признаков `X`, обычно мы имеем дело с *целевым* массивом (массивом *меток*), который принято обозначать `y`. Целевой массив обычно одномерный, имеет длину `n_samples` и хранится в массиве NumPy или объекте `Series` библиотеки

Pandas. Значения в целевом массиве могут быть непрерывными числовыми или дискретными классами/метками. Хотя некоторые оцениватели библиотеки Scikit-Learn умеют работать с несколькими целевыми величинами в виде двумерного целевого массива `[n_samples, n_targets]`, мы в основном будем работать с более простым случаем одномерного целевого массива.

Отличительная черта целевого массива от остальных столбцов признаков в том, что он представляет собой величину, значения которой мы хотим *предсказать на основе имеющихся данных*. Говоря статистическим языком, это зависимая переменная (dependent variable). Например, проанализировав предыдущие данные, мы могли бы сконструировать модель для предсказания вида цветка на основе остальных измерений. В таком случае столбец `species` рассматривался бы как целевой массив.

Имея такой целевой массив, можно воспользоваться библиотекой Seaborn (которую мы рассматривали в главе 36), чтобы без труда визуализировать данные (рис. 38.1):

```
In [2]: %matplotlib inline
        import seaborn as sns
        sns.pairplot(iris, hue='species', height=1.5);
```

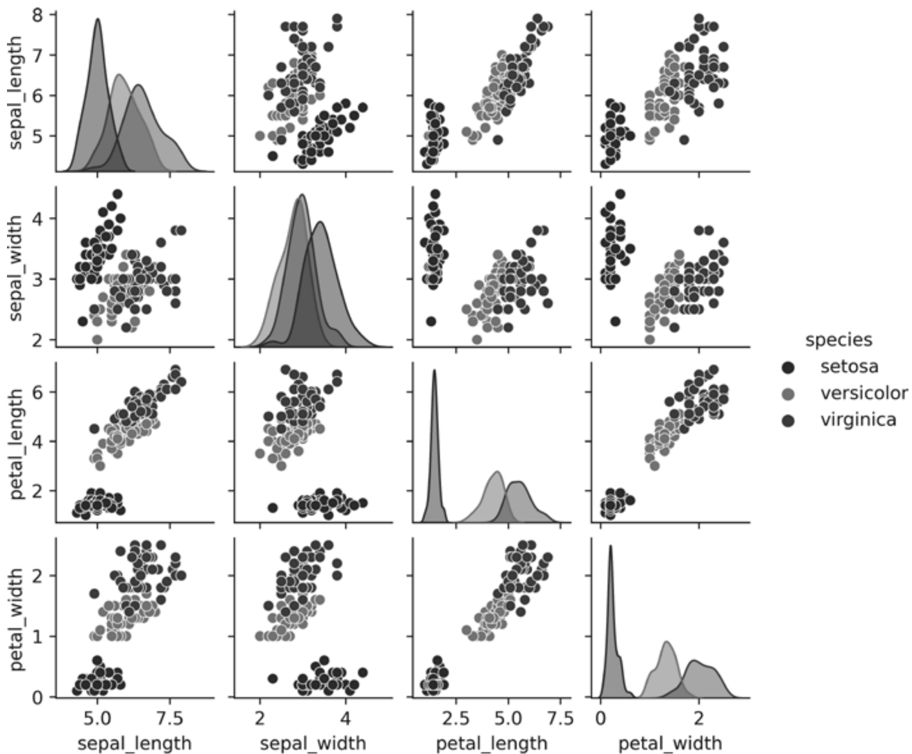


Рис. 38.1. Визуализация набора данных Iris

Для демонстрации приемов анализа набора данных Iris из Scikit-Learn извлечем матрицу признаков и целевой массив из объекта `DataFrame`. Сделать это можно с помощью операций, поддерживаемых объектом `DataFrame` и обсуждавшихся в части III:

```
In [3]: X_iris = iris.drop('species', axis=1)
        X_iris.shape
Out[3]: (150, 4)

In [4]: y_iris = iris['species']
        y_iris.shape
Out[4]: (150,)
```

Полученные массивы данных должны иметь структуру, как показано на рис. 38.2.

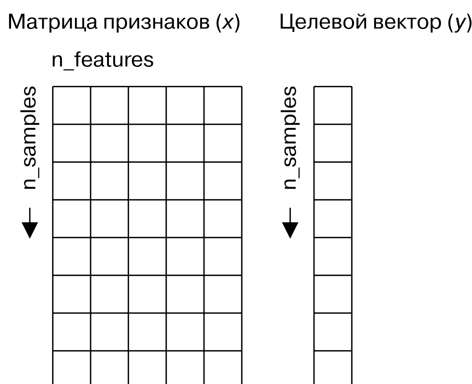


Рис. 38.2. Структура данных Scikit-Learn[1]

Теперь, получив данные в нужном формате, можно переходить к знакомству с Estimator API (инструментами вычисления статистических оценок) в библиотеке Scikit-Learn.

API статистического оценивания в Scikit-Learn

В статье, описывающей архитектуру библиотеки Scikit-Learn (http://arxiv.org/abs/1309.0238), говорится, что она основывается на следующих принципах:

- *единообразие* — интерфейс всех объектов идентичен и основан на ограниченном наборе методов, причем документация тоже единообразна;

[1] Код, сгенерировавший этот рисунок, можно найти в онлайн-приложении (https://oreil.ly/J8V6U).

- *контроль* — все задаваемые параметры определены как общедоступные атрибуты;
- *ограниченная иерархия объектов* — классы Python используются только для представления алгоритмов; наборы данных представлены в стандартных форматах (массивы NumPy, объекты `DataFrame` библиотеки Pandas, разреженные матрицы библиотеки SciPy), а для имен параметров используются стандартные строки Python;
- *объединение* — многие задачи машинного обучения можно выразить в виде последовательностей алгоритмов более низкого уровня, и библиотека Scikit-Learn использует этот факт при любой возможности;
- *разумные значения по умолчанию* — для любых параметров моделей, доступных для настройки пользователем, библиотека определяет разумные значения по умолчанию.

На практике эти принципы очень облегчают изучение Scikit-Learn. Все алгоритмы машинного обучения в библиотеке реализуются через Estimator API — единообразный интерфейс для широкого диапазона прикладных задач машинного обучения.

Основы API статистического оценивания

Применение инструментов вычисления статистических оценок из библиотеки Scikit-Learn обычно выполняется в несколько этапов.

1. Выбирается класс модели, и импортируется соответствующий класс оценивателя из библиотеки Scikit-Learn.
2. Выбираются гиперпараметры модели путем создания экземпляра этого класса с соответствующими значениями.
3. Данные помещаются в матрицу признаков и целевой вектор, как было описано выше.
4. Производится обучение модели на данных вызовом метода `fit()` экземпляра модели.
5. Обученная модель применяется к новым данным:
    - в случае обучения с учителем метки для неизвестных данных обычно предсказываются с помощью метода `predict`;
    - в случае обучения без учителя обычно выполняется преобразование свойств данных или вычисление их значений с помощью методов `transform` или `predict`.

Рассмотрим несколько простых примеров применения методов обучения без учителя и с учителем.

Пример обучения с учителем: простая линейная регрессия

В качестве примера возьмем простую линейную регрессию, то есть часто встречающийся случай подбора разделяющей прямой для данных вида (x, y). Для этого примера возьмем следующие простые данные (рис. 38.3):

```
In [5]: import matplotlib.pyplot as plt
        import numpy as np

        rng = np.random.RandomState(42)
        x = 10 * rng.rand(50)
        y = 2 * x - 1 + rng.randn(50)
        plt.scatter(x, y);
```

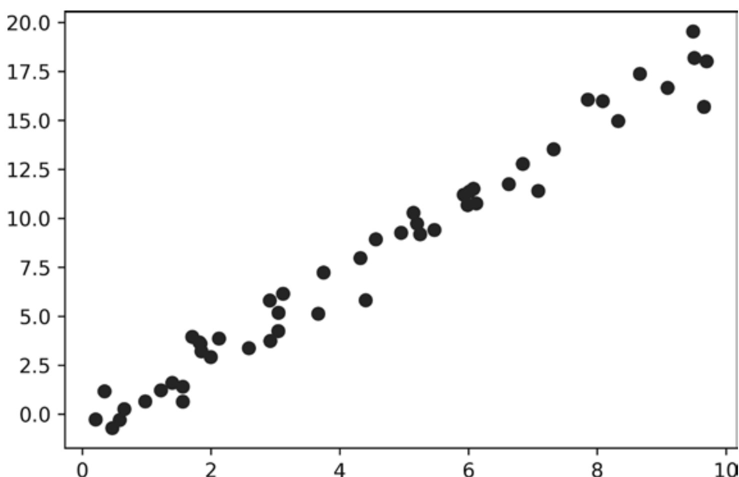


Рис. 38.3. Данные для обучения модели линейной регрессии

Заполучив данные, воспользуемся описанным выше рецептом. Пройдемся по всем шагам этого процесса, описанным в следующих разделах.

1. Выбор класса модели

Каждый класс модели в библиотеке Scikit-Learn представлен соответствующим классом языка Python. Например, для расчета модели простой линейной регрессии можно импортировать класс `LinearRegression`:

```
In[6]: from sklearn.linear_model import LinearRegression
```

Обратите внимание, что существуют и другие, более общие модели линейной регрессии, узнать больше о них можно в документации модуля `sklearn.linear_model` (https://oreil.ly/YVOFd).

2. Выбор гиперпараметров модели

Важно отметить, что *класс модели — это не то же самое, что экземпляр модели*.

После выбора класса модели у нас все еще остаются некоторые возможности для выбора. В зависимости от выбранного класса модели может понадобиться ответить на один или несколько следующих вопросов.

- Требуется ли выполнить подбор сдвига прямой (то есть точки пересечения с осью y)?
- Требуется ли нормализовать модель?
- Требуется ли выполнить предварительную обработку признаков для придания модели большей гибкости?
- Какая степень регуляризации должна быть у нашей модели?
- Сколько компонент модели предполагается использовать?

Это примеры важных решений, которые придется принять *после выбора класса модели*. Результаты этих решений часто называют *гиперпараметрами*, то есть параметрами, задаваемыми до обучения модели на данных. Выбор гиперпараметров в библиотеке Scikit-Learn осуществляется путем передачи значений при создании экземпляра модели. Мы рассмотрим количественные обоснования выбора гиперпараметров в главе 39.

Создадим экземпляр класса `LinearRegression` и укажем с помощью гиперпараметра `fit_intercept`, что требуется выполнить подбор точки пересечения с осью y:

```
In [7]: model = LinearRegression(fit_intercept=True)
        model
Out[7]: LinearRegression()
```

Имейте в виду, что операция создания экземпляра модели просто сохраняет значения гиперпараметров. В этот момент она еще не применяется ни к каким данным: API библиотеки Scikit-Learn очень четко разделяет *выбор модели* и *применение модели к данным*.

3. Формирование матрицы признаков и целевого вектора

Выше мы подробно рассмотрели представление данных в библиотеке Scikit-Learn в форме двумерной матрицы признаков и одномерного целевого вектора. Наша целевая переменная `y` уже имеет нужный вид (массив длиной `n_samples`), но нам придется проделать небольшие манипуляции с данными `x`, чтобы получить матрицу `[n_samples, n_features]`. В данном случае манипуляции сводятся к простому изменению формы одномерного массива:

```
In [8]: X = x[:, np.newaxis]
        X.shape
Out[8]: (50, 1)
```

4. Обучение модели на наших данных

Пришло время применить модель к данным. Сделать это можно с помощью метода `fit` модели:

```
In [9]: model.fit(X, y)
Out[9]: LinearRegression()
```

Команда `fit` вызывает множество «закулисных» вычислений в зависимости от модели и сохранение результатов этих вычислений в атрибутах модели, доступных для просмотра пользователем. В библиотеке Scikit-Learn по традиции все параметры модели, полученные в процессе выполнения команды `fit`, содержат в конце имен знак подчеркивания. Например, в данной линейной модели:

```
In [10]: model.coef_
Out[10]: array([1.9776566])

In [11]: model.intercept_
Out[11]: -0.9033107255311146
```

Эти два параметра определяют угловой коэффициент и точку пересечения аппроксимирующей прямой с осью y. Сравнивая результаты с описанием данных, мы видим, что они очень близки к параметрам, использовавшимся для генерирования данных: исходному угловому коэффициенту 2 и точке пересечения −1.

Часто возникает вопрос относительно погрешностей в подобных внутренних параметрах модели. В целом библиотека Scikit-Learn не предоставляет инструментов, позволяющих делать выводы непосредственно из внутренних параметров модели: интерпретация параметров скорее вопрос *статистического моделирования*, а не *машинного обучения*. Машинное обучение концентрируется в основном на том, что *предсказывает* модель. Для тех, кто хочет узнать больше о смысле подбираемых параметров модели, существуют другие инструменты, включая пакет `statsmodels` для Python (https://oreil.ly/adDFZ).

5. Предсказание меток для новых данных

После обучения модели главная задача машинного обучения с учителем — вычисление с ее помощью значений для новых данных, не являющихся частью обучающей последовательности. Сделать это в библиотеке Scikit-Learn можно с помощью метода `predict`. В этом примере наши новые данные будут сеткой x-значений и нас будет интересовать, какие y-значения предсказывает модель:

```
In [12]: xfit = np.linspace(-1, 11)
```

Как и ранее, эти x-значения требуется преобразовать в матрицу признаков `[n_samples, n_features]`, после чего можно подать их на вход модели:

```
In [13]: Xfit = xfit[:, np.newaxis]
         yfit = model.predict(Xfit)
```

Наконец, визуализируем результаты, нарисовав сначала график исходных данных, а затем обученную модель (рис. 38.4):

```
In [14]: plt.scatter(x, y)
         plt.plot(xfit, yfit);
```

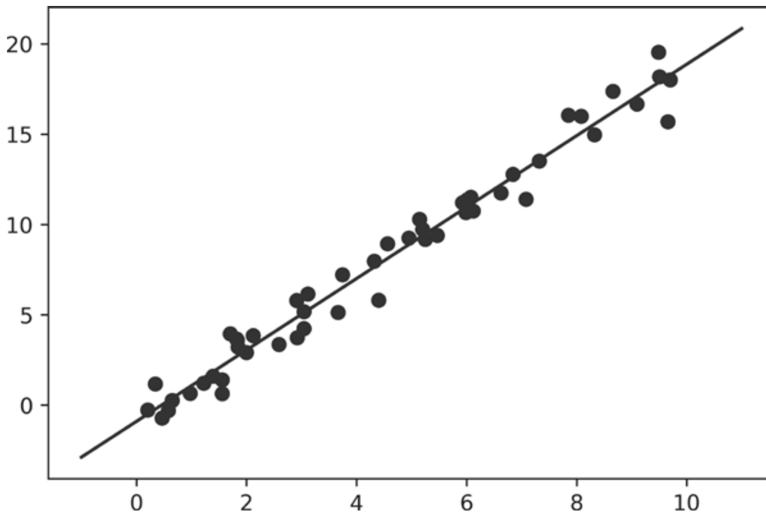


Рис. 38.4. Простая линейная регрессионная аппроксимация наших данных

Обычно эффективность модели оценивают, сравнивая ее результаты с эталоном, как мы увидим в следующем примере.

Пример обучения с учителем: классификация набора данных Iris

Рассмотрим другой пример того же процесса, воспользовавшись обсуждавшимся ранее набором данных Iris. Зададим себе такой вопрос: насколько хорошо можно предсказать метки остальных данных с помощью модели, обученной на некоторой части данных набора Iris?

Для ответа на вопрос воспользуемся чрезвычайно простой обобщенной моделью, известной под названием *«гауссов наивный байесовский классификатор»*, исходящей из допущения, что все классы взяты из выровненного по осям координат гауссова распределения (см. главу 41). Гауссов наивный байесовский классификатор

в силу отсутствия гиперпараметров и высокой производительности — хороший кандидат на роль эталонной модели классификации. Давайте немного поэкспериментируем с ним, прежде чем выяснять, можно ли получить лучшие результаты с помощью более сложных моделей.

Было бы желательно оценить работу модели на неизвестных ей данных, поэтому разделим данные на обучающую (training set) и контрольную последовательности (testing set). Это можно сделать вручную, но удобнее воспользоваться вспомогательной функцией `train_test_split`:

```
In [15]: from sklearn.model_selection import train_test_split
         Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
                                                         random_state=1)
```

После упорядочения данных последуем нашему рецепту классификации:

```
In [16]: from sklearn.naive_bayes import GaussianNB # 1) выбор класса модели
         model = GaussianNB()                       # 2) создание модели
         model.fit(Xtrain, ytrain)                  # 3) обучение модели
         y_model = model.predict(Xtest)             # 4) получение предсказаний
                                                    #    для новых данных
```

Воспользуемся утилитой `accuracy_score` для выяснения доли предсказанных меток, совпадающих с истинным значением:

```
In [17]: from sklearn.metrics import accuracy_score
         accuracy_score(ytest, y_model)
Out[17]: 0.9736842105263158
```

Как видите, точность превысила 97 %, поэтому для этого конкретного набора данных даже очень простой алгоритм классификации оказывается эффективным!

Пример обучения без учителя: понижение размерности набора данных Iris

В качестве примера задачи обучения без учителя рассмотрим задачу понижения размерности набора данных Iris с целью упрощения его визуализации. Напомню, что данные Iris четырехмерны: для каждого образца зафиксированы четыре признака.

Задача понижения размерности заключается в выявлении подходящего представления более низкой размерности, сохраняющего существенные признаки данных. Зачастую понижение размерности используется для облегчения визуализации данных, в конце концов, гораздо проще строить график данных в двух измерениях, чем в четырех или более!

В этом разделе мы используем *метод главных компонент* (PCA; см. главу 45) — быстрый линейный метод понижения размерности. Наша модель должна будет возвращать две компоненты, то есть двумерное представление данных.

Следуя вышеописанной последовательности шагов, получаем:

```
In [18]: from sklearn.decomposition import PCA  # 1) выбор класса модели
         model = PCA(n_components=2)            # 2) создание модели
         model.fit(X_iris)                      # 3) обучение модели
         X_2D = model.transform(X_iris)         # 4) преобразование данных
```

Построим график полученных результатов. Сделать это быстрее всего можно, вставив результаты в исходный объект `DataFrame` с данными Iris и воспользовавшись функцией `lmplot` для отображения результатов:

```
In [19]: iris['PCA1'] = X_2D[:, 0]
         iris['PCA2'] = X_2D[:, 1]
         sns.lmplot(x="PCA1", y="PCA2", hue='species', data=iris, fit_reg=False);
```

Как показано на рис. 38.5, в двумерном представлении виды цветков четко разделены, хотя алгоритм PCA ничего не знает о метках видов цветов! Этот эксперимент показывает, что даже относительно простая классификация на этом наборе данных, вероятно, будет работать достаточно хорошо.

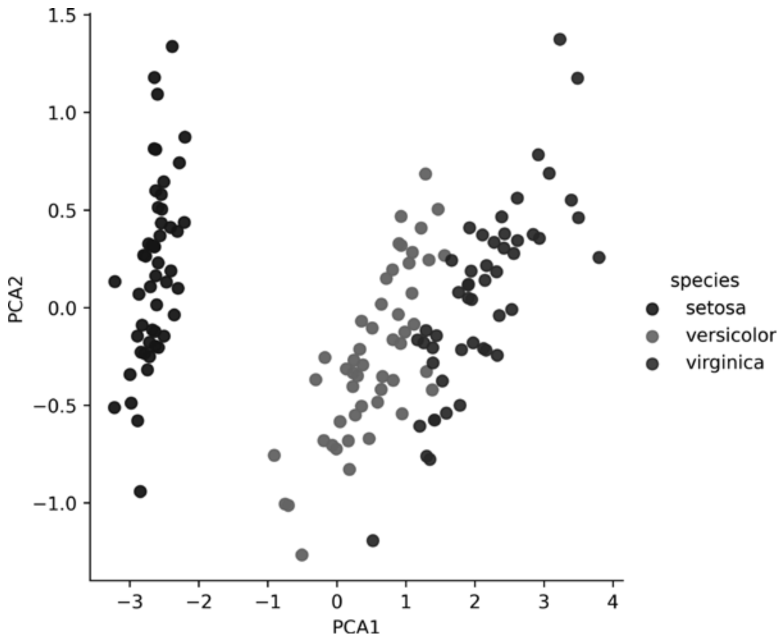


Рис. 38.5. Проекция данных набора Iris на двумерное пространство

Обучение без учителя: кластеризация набора данных Iris

Теперь рассмотрим кластеризацию набора данных Iris. Алгоритм кластеризации пытается выделить группы данных безотносительно к каким-либо меткам. Здесь мы используем мощный алгоритм кластеризации под названием *смесь гауссовых распределений* (Gaussian mixture model, GMM), который изучим подробнее в главе 48. Метод GMM пытается смоделировать данные в виде набора гауссовых пятен.

Ниже показан процесс обучения модели смеси гауссовых распределений:

```
In [20]: from sklearn.mixture import GaussianMixture      # 1) выбор класса модели
         model = GaussianMixture(n_components=3,
                                 covariance_type='full')  # 2) создание модели
         model.fit(X_iris)                                 # 3) обучение модели
         y_gmm = model.predict(X_iris)                     # 4) определение меток
                                                                кластеров
```

Как и ранее, добавим столбец `cluster` в `DataFrame` с данными Iris и воспользуемся библиотекой Seaborn для построения графика результатов (рис. 38.6):

```
In [21]: iris['cluster'] = y_gmm
         sns.lmplot(x="PCA1", y="PCA2", data=iris, hue='species',
                    col='cluster', fit_reg=False);  # Кластер
```

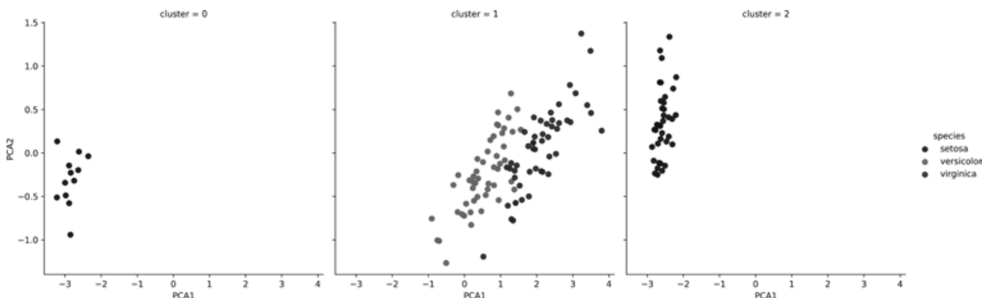


Рис. 38.6. Проекция данных набора Iris на двумерное пространство

Разбив данные в соответствии с номерами кластеров, мы видим, насколько хорошо алгоритм GMM восстановил требуемые метки: вид *setosa* идеально выделен в кластер 0, правда, небольшое количество экземпляров видов *versicolor* и *virginica* смешались между собой. Следовательно, даже в отсутствие эксперта, который мог бы сообщить нам, к каким видам относятся отдельные цветки, одних измерений вполне достаточно для *автоматического* распознавания этих разновидностей

цветков с помощью простого алгоритма кластеризации! Подобный алгоритм может в дальнейшем помочь специалистам в предметной области выяснить связи между исследуемыми образцами.

Прикладная задача: анализ рукописных цифр

Продемонстрируем эти принципы на более интересной задаче, рассмотрев один из аспектов задачи оптического распознавания символов — распознавание рукописных цифр. Традиционно эта задача включает определение местоположения на рисунке и определение значений символов. Мы пойдем самым коротким путем и воспользуемся встроенным в библиотеку Scikit-Learn набором предварительно отформатированных цифр.

Загрузка и визуализация цифр

Воспользуемся интерфейсом доступа к данным библиотеки Scikit-Learn и посмотрим на эти данные:

```
In [22]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.images.shape
Out[22]: (1797, 8, 8)
```

Изображения хранятся в трехмерном массиве: 1797 образцов, представленных сетками пикселов размером 8 × 8. Визуализируем первую сотню (рис. 38.7):

```
In [23]: import matplotlib.pyplot as plt

         fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                                  subplot_kw={'xticks':[], 'yticks':[]},
                                  gridspec_kw=dict(hspace=0.1, wspace=0.1))

         for i, ax in enumerate(axes.flat):
             ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
             ax.text(0.05, 0.05, str(digits.target[i]),
                     transform=ax.transAxes, color='green')
```

Для работы с этими данными нужно получить их двумерное `[n_samples, n_features]` представление. Для этого будем трактовать каждый пиксел как признак, то есть «расплющим» массивы пикселов так, чтобы каждую цифру представлял одномерный массив с 64 пикселами. Нам также понадобится целевой массив, задающий

для каждой цифры предопределенную метку. Эти два параметра встроены в набор данных в виде атрибутов `data` и `target` соответственно:

```
In [24]: X = digits.data
         X.shape
Out[24]: (1797, 64)
In [25]: y = digits.target
         y.shape
Out[25]: (1797,)
```

Итого получаем 1797 экземпляров и 64 признака.

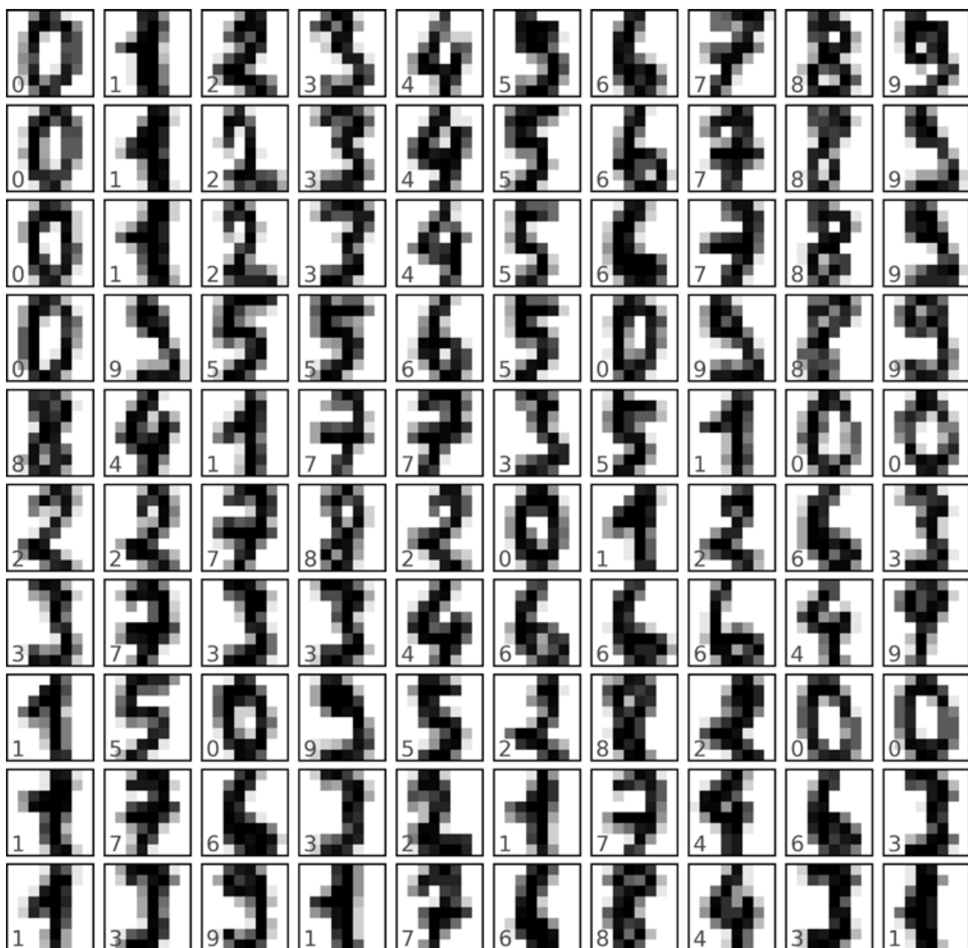


Рис. 38.7. Набор данных с изображениями рукописных цифр; каждый образец представлен сеткой пикселов размером 8 × 8

Обучение без учителя: понижение размерности

Хотелось бы визуализировать наши точки в 64-мерном параметрическом пространстве, но сделать это эффективно в пространстве такой высокой размерности непросто. Поэтому уменьшим количество измерений до 2, воспользовавшись методом обучения без учителя. Здесь мы будем применять алгоритм обучения на базе многообразий под названием Isomap (см. главу 46) и преобразуем данные в двумерный вид:

```
In [26]: from sklearn.manifold import Isomap
         iso = Isomap(n_components=2)
         iso.fit(digits.data)
         data_projected = iso.transform(digits.data)
         print(data_projected.shape)
Out[26]: (1797, 2)
```

Теперь, преобразовав данные в двумерное представление, построим график, чтобы увидеть, можно ли что-то понять из их структуры (рис. 38.8):

```
In [27]: plt.scatter(data_projected[:, 0], data_projected[:, 1], c=digits.target,
                     edgecolor='none', alpha=0.5,
                     cmap=plt.cm.get_cmap('viridis', 10))
         plt.colorbar(label='digit label', ticks=range(10))
         plt.clim(-0.5, 9.5);
```

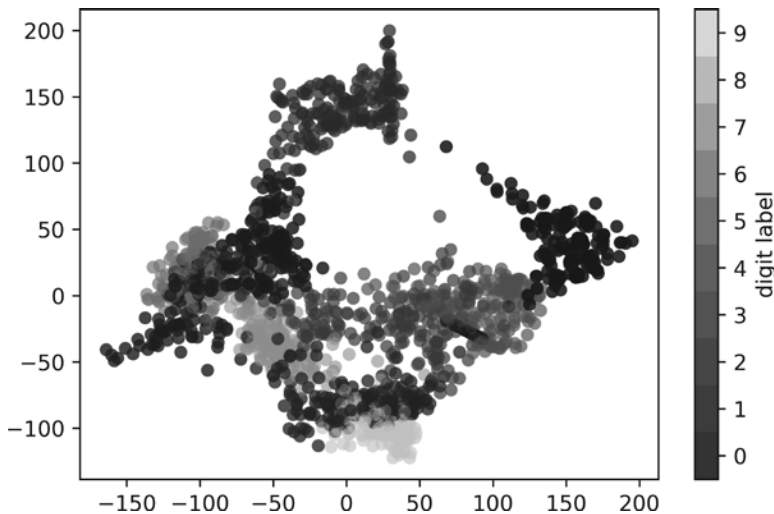


Рис. 38.8. Isomap-представление набора данных цифр

Этот график дает нам представление о распределении различных цифр в 64-мерном пространстве. Например, нули (отображаемые черным цветом) и единицы (отображаемые фиолетовым) практически не пересекаются в параметрическом пространстве. На интуитивном уровне это представляется вполне логичным: нули содержат пустое место в середине изображения, а у единиц там, наоборот, чернила. В то же время единицы и четверки на графике располагаются сплошным спектром, что понятно, ведь некоторые люди рисуют единицы со «шляпками», из-за чего они становятся похожи на четверки.

В целом, однако, различные группы образуют достаточно компактные группы в параметрическом пространстве. Это значит, что даже простой алгоритм классификации с учителем должен достаточно хорошо классифицировать их.

Классификация цифр

Применим алгоритм классификации к нашим данным. Как и в случае с набором данных Iris, разобьем данные на обучающую и контрольную последовательности, после чего на первой обучим гауссову наивную байесовскую модель:

```
In [28]: Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
In [29]: from sklearn.naive_bayes import GaussianNB
         model = GaussianNB()
         model.fit(Xtrain, ytrain)
         y_model = model.predict(Xtest)
```

Затем оценим ее точность, выполнив предсказания с помощью обученной модели и сравнив истинные значения из контрольной последовательности с предсказанными:

```
In [30]: from sklearn.metrics import accuracy_score
         accuracy_score(ytest, y_model)
Out[30]: 0.8333333333333334
```

Даже такая исключительно простая модель показала более чем 80%-ную точность классификации цифр! Однако из одного числа сложно понять, где наша модель ошиблась. Для этой цели удобно использовать так называемую *матрицу различий* (confusion matrix), вычислить которую можно с помощью библиотеки Scikit-Learn, а нарисовать посредством Seaborn (рис. 38.9):

```
In [31]: from sklearn.metrics import confusion_matrix

         mat = confusion_matrix(ytest, y_model)

         sns.heatmap(mat, square=True, annot=True, cbar=False, cmap='Blues')
         plt.xlabel('predicted value') # Прогнозируемое значение
         plt.ylabel('true value');     # Истинное значение
```

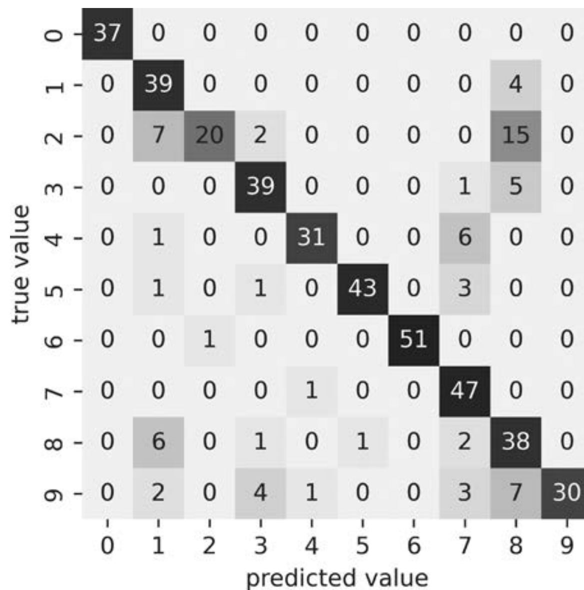


Рис. 38.9. Матрица различий, демонстрирующая частоты ошибочных классификаций

Этот рисунок демонстрирует места, в которых наш классификатор склонен ошибаться, например, значительное количество двоек ошибочно классифицировано как единицы или восьмерки.

Другой способ получения информации о характеристиках модели — построить график входных данных вместе с предсказанными метками. Используем зеленый цвет для обозначения правильных меток и красный — для ошибочных (рис. 38.10):

```
In [32]: fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                                  subplot_kw={'xticks':[], 'yticks':[]},
                                  gridspec_kw=dict(hspace=0.1, wspace=0.1))

         test_images = Xtest.reshape(-1, 8, 8)

         for i, ax in enumerate(axes.flat):
             ax.imshow(test_images[i], cmap='binary', interpolation='nearest')
             ax.text(0.05, 0.05, str(y_model[i]),
                     transform=ax.transAxes,
                     color='green' if (ytest[i] == y_model[i]) else 'red')
```

Из этого подмножества данных можно почерпнуть полезную информацию относительно мест, в которых алгоритм ошибается. Чтобы поднять точность выше достигнутых 83 %, можно воспользоваться более сложным алгоритмом, таким как метод опорных векторов (см. главу 43), случайные леса (см. главу 44), или другим методом классификации.

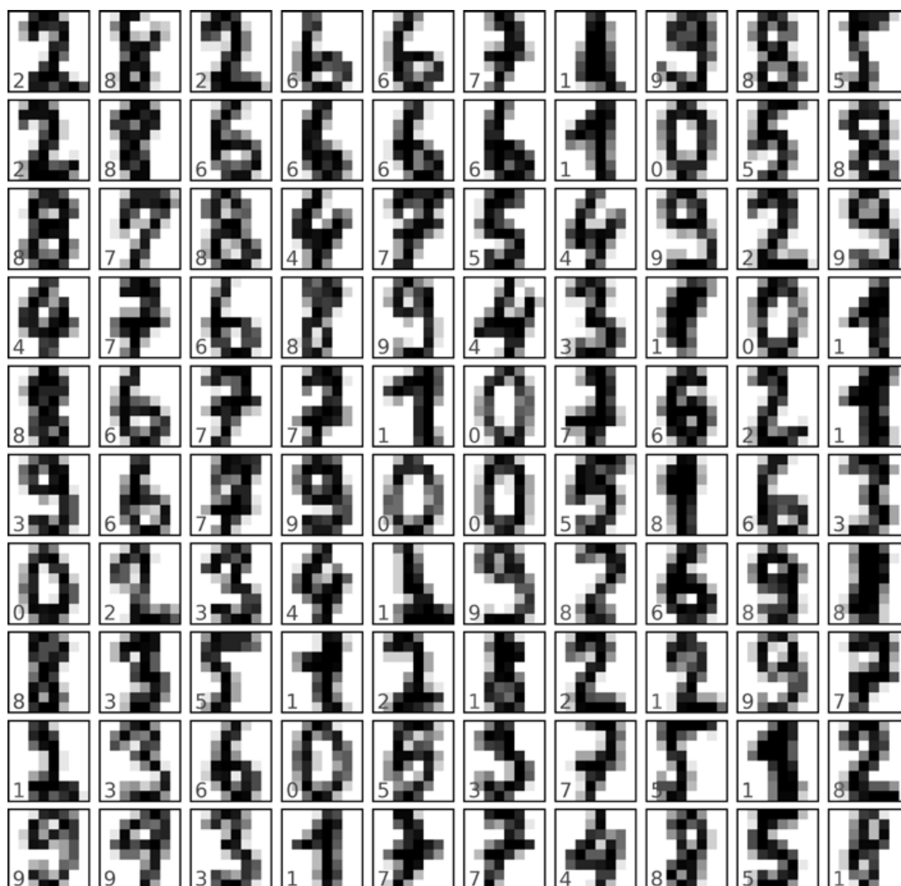


Рис. 38.10. Данные, показывающие правильные (зеленым цветом) и ошибочные (красным) метки. Цветную версию этого графика можно найти в онлайн-приложении (https://oreil.ly/PDSH_GitHub)

Резюме

В этой главе мы рассмотрели основные возможности библиотеки Scikit-Learn для представления данных, а также API статистического оценивания. Независимо от типа оценивателя применяется одна и та же схема: импорт/создание экземпляра/обучение/предсказание. Вооруженные этой информацией, вы сможете, изучив документацию библиотеки Scikit-Learn, начать экспериментировать, используя различные модели для своих данных.

В следующей главе мы рассмотрим, вероятно, самый важный вопрос машинного обучения: выбор и проверку модели.

ГЛАВА 39

Гиперпараметры и проверка модели

В предыдущей главе описана основная схема использования моделей машинного обучения с учителем.

1. Выбрать класс модели.
2. Выбрать гиперпараметры модели.
3. Обучить модель на данных из обучающей последовательности.
4. Использовать модель для предсказания меток новых данных.

Первые два пункта — выбор модели и выбор гиперпараметров — вероятно, важнее всего для эффективного использования этих инструментов и методов. Чтобы сделать оптимальный выбор, необходим способ, позволяющий *проверить*, насколько хорошо конкретная модель с конкретными гиперпараметрами аппроксимирует конкретные данные. На первый взгляд это простая задача, но в ней полно подводных камней, которые необходимо обойти.

Соображения относительно проверки модели

В принципе, проверить модель просто: достаточно после выбора модели и гиперпараметров оценить ее качество, применив ее к части обучающей последовательности и сравнив предсказания с известными значениями.

В этом разделе я сначала покажу «наивный» подход к проверке модели и объясню его недостатки, а затем продемонстрирую приемы, основанные на использовании отложенных наборов данных и перекрестной проверке, позволяющие оценить модель более точно.

Плохой способ проверки модели

Рассмотрим «наивный» подход к проверке на наборе данных Iris, с которым мы работали в предыдущем разделе. Начнем с загрузки данных:

```
In [1]: from sklearn.datasets import load_iris
        iris = load_iris()
        X = iris.data
        y = iris.target
```

Затем выберем модель и гиперпараметры. В этом примере мы используем классификатор на основе метода k средних с `n_neighbors=1`. Это очень простая и интуитивно понятная модель, которую можно описать фразой «Неизвестная точка имеет ту же метку, что и ближайшая к ней точка из обучающего набора»:

```
In [2]: from sklearn.neighbors import KNeighborsClassifier
        model = KNeighborsClassifier(n_neighbors=1)
```

Далее обучим модель и используем ее для предсказания меток уже известных данных:

```
In [3]: model.fit(X, y)
        y_model = model.predict(X)
```

Наконец, вычислим долю правильно классифицированных точек:

```
In [4]: from sklearn.metrics import accuracy_score
        accuracy_score(y, y_model)
Out[4]: 1.0
```

Как видите, показатель точности равен 1,0, то есть наша модель правильно классифицировала 100 % точек! Но действительно ли это правильная оценка ожидаемой точности? Действительно ли нам попалась модель, которая будет работать правильно в 100 % случаев?

Как вы наверняка догадались, ответ на этот вопрос — нет. На самом деле этот подход имеет фундаментальный изъян: *обучение и оценка модели выполняются на одних и тех же данных*! Более того, модель ближайшего соседа, работающая путем *обучения на примерах* (instance-based estimator), попросту сохраняет обучающие данные и предсказывает метки, сравнивая новые данные с этими сохраненными точками. За исключением некоторых специально сконструированных случаев ее точность будет всегда равна 100 %!

Хороший способ проверки модели: отложенные данные

Для более точного выяснения качества модели воспользуемся так называемыми *отложенными наборами данных* (holdout sets), то есть отложим некоторое подмножество данных из обучающей последовательности модели, после чего используем

его для проверки модели. Это разделение в Scikit-Learn можно произвести с помощью утилиты `train_test_split`:

```
In [5]: from sklearn.model_selection import train_test_split
        # Делим данные на два равных набора
        X1, X2, y1, y2 = train_test_split(X, y, random_state=0,
                                          train_size=0.5)

        # Обучаем модель на одном из наборов данных
        model.fit(X1, y1)

        # Оцениваем модель на другом наборе
        y2_model = model.predict(X2)
        accuracy_score(y2, y2_model)
Out[5]: 0.9066666666666666
```

Мы получили более правдоподобный результат: классификатор на основе метода ближайшего соседа демонстрирует точность около 90 % на отложенном наборе данных. Отложенный набор данных схож с неизвестными данными, поскольку модель «не видела» их ранее.

Перекрестная проверка модели

Отделение части данных, которые можно было бы использовать для обучения модели, — один из недостатков приема проверки на основе отложенного набора данных. В предыдущем случае половина набора данных не участвует в обучении модели! Это неоптимально и может стать причиной проблем, особенно если исходный набор данных невелик.

Один из способов решения этой проблемы — *перекрестная проверка* (cross-validation), то есть выполнение последовательности аппроксимаций, в которых каждое подмножество данных используется в качестве как обучающей последовательности, так и контрольного набора. Наглядно его можно представить, как показано на рис. 39.1.

Рис. 39.1. Двухблочная перекрестная проверка[1]

[1] Код, генерирующий рисунки из этой главы, можно найти в онлайн-приложении (https://oreil.ly/jv0wb).

Мы выполняем две попытки проверки, попеременно используя каждую половину данных в качестве отложенного набора данных. Разделив данные, как показано выше, это можно реализовать так:

```
In [6]: y2_model = model.fit(X1, y1).predict(X2)
        y1_model = model.fit(X2, y2).predict(X1)
        accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)
Out[6]: (0.96, 0.9066666666666666)
```

Полученные числа — две оценки точности, которые можно обобщить (например, взяв среднее значение) для получения лучшей меры качества модели. Этот конкретный вид перекрестной проверки, используя который мы разбили данные на два набора и по очереди использовали каждый из них для проверки, называется *двухблочной перекрестной проверкой* (two-fold cross-validation).

Эту идею можно распространить на случай большего числа попыток и большего количества блоков данных, например, на рис. 39.2 показан вариант пятиблочной перекрестной проверки.

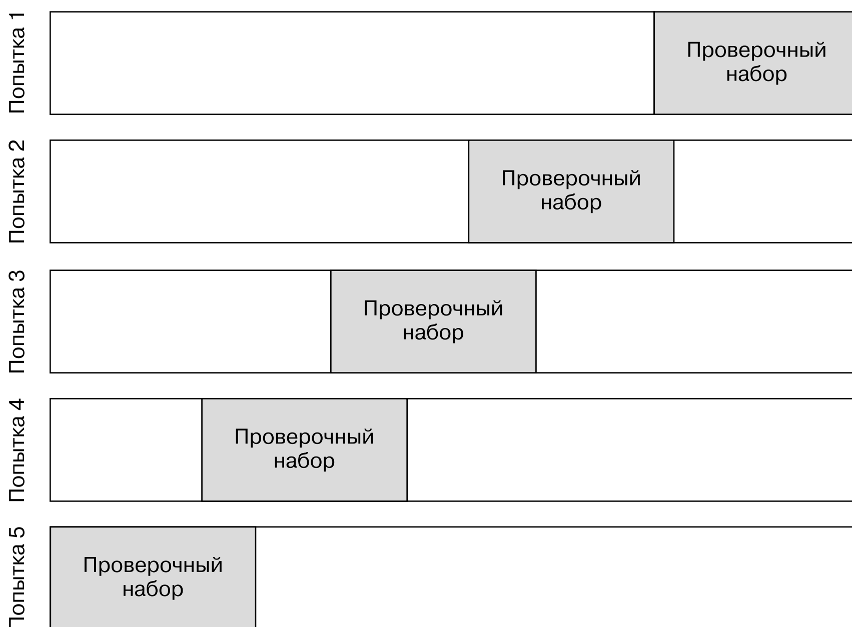


Рис. 39.2. Пятиблочная перекрестная проверка

Здесь мы разбиваем данные на пять групп и по очереди используем каждую из них для оценки качества обучения модели на остальных 4/5 данных. Делать

это вручную довольно утомительно, поэтому воспользуемся удобной утилитой `cross_val_score` из библиотеки Scikit-Learn:

```
In [7]: from sklearn.model_selection import cross_val_score
        cross_val_score(model, X, y, cv=5)
Out[7]: array([0.96666667, 0.96666667, 0.93333333, 0.93333333, 1.        ])
```

Повторение проверки по различным подмножествам данных дает более полное представление о качестве работы алгоритма.

Библиотека Scikit-Learn реализует множество схем перекрестной проверки, удобных в конкретных ситуациях. Они реализованы в виде итераторов в модуле `model_selection`. Например, мы взяли предельный случай, при котором количество блоков равно количеству точек данных, и обучаем модель в каждой попытке на всех точках, кроме одной. Такой тип перекрестной проверки известен под названием *перекрестной проверки по отдельным объектам* (leave-one-out cross-validation — дословно «перекрестная проверка по всем без одного»), ее можно использовать следующим образом:

```
In [8]: from sklearn.model_selection import LeaveOneOut
        scores = cross_val_score(model, X, y, cv=LeaveOneOut())
        scores
Out[8]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 0., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
               0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1.,
               1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Поскольку мы имеем 150 образцов, перекрестная проверка по отдельным объектам дает результаты по 150 попыткам, причем каждый результат показывает, правильно (1,0) или неправильно (0,0) была предсказана принадлежность проверяемого образца. Среднее значение этих результатов дает нам общую оценку частоты ошибок:

```
In [9]: scores.mean()
Out[9]: 0.96
```

Аналогично реализуются и другие схемы перекрестной проверки. Чтобы узнать, какие еще варианты есть в библиотеке Scikit-Learn, воспользуйтесь оболочкой IPython и исследуйте содержимое модуля `sklearn.model_selection` или обратитесь к онлайн-документации по перекрестной проверке библиотеки Scikit-Learn (https://oreil.ly/rITkn).

Выбор оптимальной модели

Теперь, познакомившись с основными идеями простой и перекрестной проверки, углубимся немного в вопросы выбора модели и гиперпараметров. Это один из наиболее важных аспектов машинного обучения, причем во вводных курсах машинного обучения эти вопросы зачастую рассматриваются лишь вскользь.

Важнейшим является следующий вопрос: что делать, *если модель показывает недостаточно хорошие результаты*? Существует несколько возможных ответов:

- использовать более сложную/гибкую модель;
- использовать менее сложную/гибкую модель;
- собрать больше образцов для обучения;
- собрать больше данных для добавления новых признаков в образцы.

Ответы на этот вопрос часто выглядят странно. В частности, иногда использование более сложной модели приводит к худшим результатам, а добавление новых образцов в обучающую последовательность не приводит к их улучшению! Успешных специалистов-практиков в области машинного обучения как раз и отличает умение определять, какие действия улучшат характеристики модели.

Компромисс между систематической ошибкой и дисперсией

По существу, выбор «оптимальной модели» сводится к поиску наилучшего компромисса между *систематической ошибкой* (bias) и *дисперсией* (variance). Рассмотрим рис. 39.3, где представлены два случая регрессионной аппроксимации одного и того же набора данных.

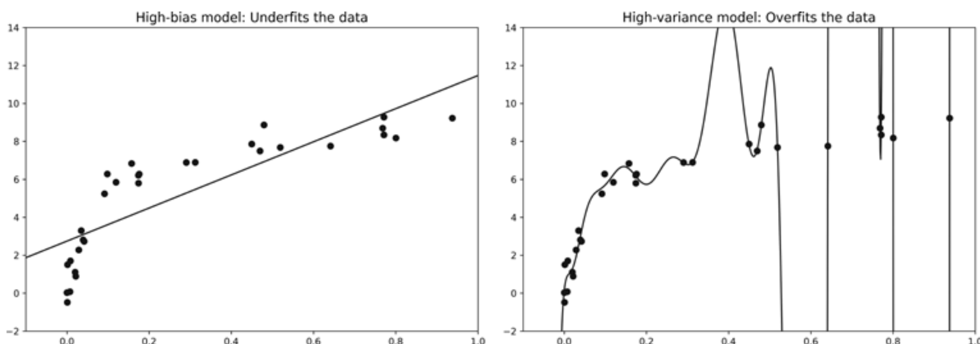


Рис. 39.3. Модели регрессии со значительной систематической ошибкой и высокой дисперсией

Очевидно, что обе модели не слишком хорошо аппроксимируют данные, но по разным причинам.

Модель слева пытается найти прямолинейное приближение к данным. Но в силу того, что внутренняя структура данных сложнее прямой линии, прямолинейная модель не может описать этот набор данных достаточно точно. О подобной модели говорят, что она *недообучена* (underfit), то есть гибкости модели недостаточно для удовлетворительного учета всех признаков в данных. Другими словами, эта модель страдает значительной систематической ошибкой.

Модель справа пытается аппроксимировать данные многочленом высокой степени. В этом случае модель достаточно гибкая, чтобы практически идеально соответствовать всем нюансам данных, но, несмотря на то что она описывает обучающую последовательность очень точно, ее форма больше напоминает шум, чем истинную форму процесса, сгенерировавшего данные. О подобной модели говорят, что она *переобучена* (overfit), то есть гибкость модели настолько высока, что она учитывает не только исходное распределение данных, но и случайные ошибки в них. Другими словами, эта модель страдает высокой дисперсией.

Чтобы взглянуть с другой стороны, посмотрим, что получится, если воспользоваться этими двумя моделями для предсказания *y*-значения для каких-либо новых данных. В диаграммах на рис. 39.4 красные (более светлые в черно-белом варианте книги) точки обозначают данные, исключенные из обучающей последовательности.

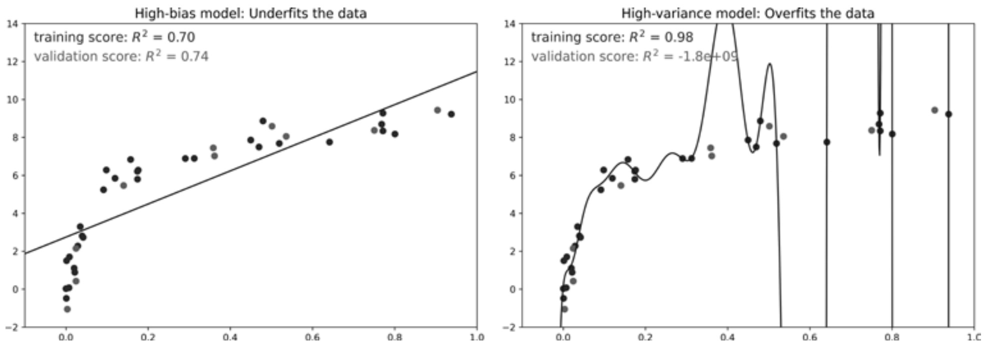


Рис. 39.4. Оценки эффективности моделей со значительной систематической ошибкой и высокой дисперсией на обучающих и контрольных данных

В качестве оценки эффективности здесь используется R^2 — коэффициент детерминации или коэффициент смешанной корреляции (подробнее о нем можно прочитать по адресу https://ru.wikipedia.org/wiki/Коэффициент_детерминации). Он представляет меру, насколько наша модель точнее по сравнению с простым средним значением целевых величин. $R^2 = 1$ означает идеальное совпадение предсказаний,

а $R^2 = 0$ показывает, что модель оказалась ничем не лучше простого среднего значения, а отрицательные значения указывают на модели, точность которых ниже простого среднего значения. На основе оценок эффективности двух моделей мы можем сделать следующие выводы.

- Для моделей со значительной систематической ошибкой эффективность модели на контрольном наборе данных сопоставима с ее эффективностью на обучающей последовательности.
- Для моделей с высокой дисперсией эффективность модели на контрольном наборе данных существенно хуже ее эффективности на обучающей последовательности.

Если бы мы умели управлять сложностью модели, то оценки эффективности на обучающих и контрольных данных вели бы себя, как показано на рис. 39.5. Эту кривую часто называют *кривой проверки* (validation curve). На ней можно наблюдать следующие важные особенности.

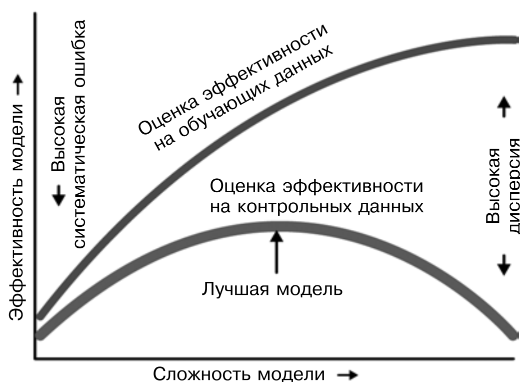


Рис. 39.5. Схематическое изображение зависимости между сложностью модели и оценками эффективности на обучающих и контрольных данных

- Оценка эффективности на обучающих данных всегда превышает оценку эффективности на контрольных данных. Это логично: модель лучше соответствует данным, которые она уже видела, чем данным, которые она еще не видела.
- Модели с очень низкой сложностью (со значительной систематической ошибкой) недостаточно обучены, то есть они будут плохо предсказывать и данные из обучающей последовательности, и данные, не виденные ранее.
- Модели с очень высокой сложностью (с высокой дисперсией) переобучены, то есть они будут очень хорошо предсказывать данные из обучающей последовательности, но плохо любые ранее не виденные данные.

- Кривая проверки достигает максимума в какой-то промежуточной точке. Этот уровень сложности означает приемлемый компромисс между систематической ошибкой и дисперсией.

Средства регулирования сложности модели различаются в зависимости от модели. Как осуществлять подобную регулировку для каждой из моделей, мы увидим в следующих разделах, когда будем обсуждать конкретные модели подробнее.

Кривые проверки в библиотеке Scikit-Learn

Рассмотрим пример расчета кривой проверки для класса моделей. Мы будем использовать *модель полиномиальной регрессии* (polynomial regression model): это обобщенная линейная модель с параметризованной степенью многочлена. Например, многочлен 1-й степени аппроксимирует наши данные прямой линией; при параметрах модели a и b:

$$y = ax + b.$$

Многочлен 3-й степени аппроксимирует данные кубической кривой; при параметрах модели a, b, c, d:

$$y = ax^3 + bx^2 + cx + d.$$

Это можно обобщить на любое количество полиномиальных признаков. В библиотеке Scikit-Learn реализовать это можно с помощью простой линейной регрессии в сочетании с полиномиальным препроцессором. Мы воспользуемся *конвейером* (pipeline) для соединения этих операций в единую цепочку (мы обсудим полиномиальные признаки и конвейеры подробнее в главе 40):

```
In [10]: from sklearn.preprocessing import PolynomialFeatures
         from sklearn.linear_model import LinearRegression
         from sklearn.pipeline import make_pipeline

         def PolynomialRegression(degree=2, **kwargs):
             return make_pipeline(PolynomialFeatures(degree),
                                  LinearRegression(**kwargs))
```

Теперь создадим данные, на которых будем обучать нашу модель:

```
In [11]: import numpy as np

         def make_data(N, err=1.0, rseed=1):
             # Создаем случайные выборки данных
             rng = np.random.RandomState(rseed)
             X = rng.rand(N, 1) ** 2
             y = 10 - 1. / (X.ravel() + 0.1)
             if err > 0:
```

```
        y += err * rng.randn(N)
    return X, y

X, y = make_data(40)
```

Визуализируем наши данные вместе с несколькими аппроксимациями в виде их многочленов различной степени (рис. 39.6):

```
In [12]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')

         X_test = np.linspace(-0.1, 1.1, 500)[:, None]

         plt.scatter(X.ravel(), y, color='black')
         axis = plt.axis()
         for degree in [1, 3, 5]:
             y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)
             plt.plot(X_test.ravel(), y_test, label='degree={0}'.format(degree))
         plt.xlim(-0.1, 1.0)
         plt.ylim(-2, 12)
         plt.legend(loc='best');
```

Параметром, служащим для управления сложностью модели, в данном случае является степень многочлена, которая может быть любым неотрицательным целым числом. Не помешает задать себе вопрос: какая степень многочлена обеспечивает подходящий компромисс между систематической ошибкой (недообучение) и дисперсией (переобучение)?

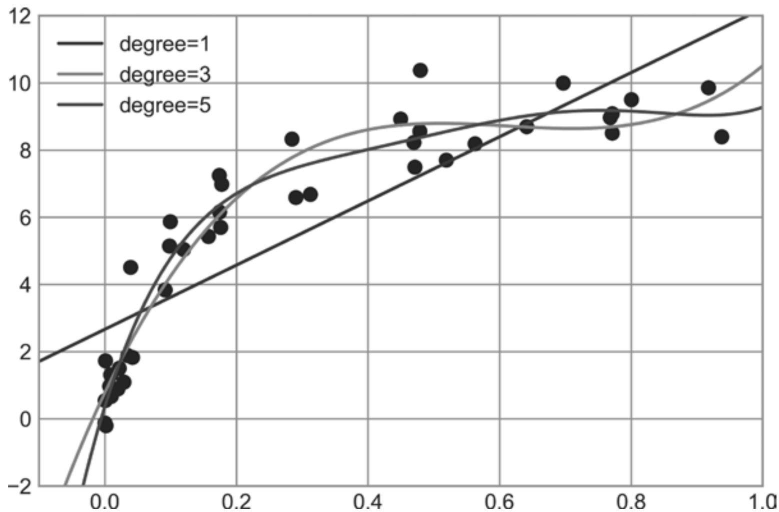


Рис. 39.6. Аппроксимации набора данных тремя различными полиномиальными моделями

Чтобы решить этот вопрос, построим кривую проверки для этих конкретных данных и моделей. Проще всего сделать это с помощью удобной утилиты `validation_curve`, предоставляемой библиотекой Scikit-Learn. Эта функция принимает модель, данные, название параметра и диапазон для анализа и автоматически вычисляет в этом диапазоне значение оценки эффективности на обучающих и контрольных данных (рис. 39.7):

```
In [13]: from sklearn.model_selection import validation_curve
         degree = np.arange(0, 21)
         train_score, val_score = validation_curve(
             PolynomialRegression(), X, y,
             param_name='polynomialfeatures__degree',
             param_range=degree, cv=7)

         plt.plot(degree, np.median(train_score, 1),
                  color='blue', label='training score')
         plt.plot(degree, np.median(val_score, 1),
                  color='red', label='validation score')
         plt.legend(loc='best')
         plt.ylim(0, 1)
         plt.xlabel('degree')
         plt.ylabel('score');
```

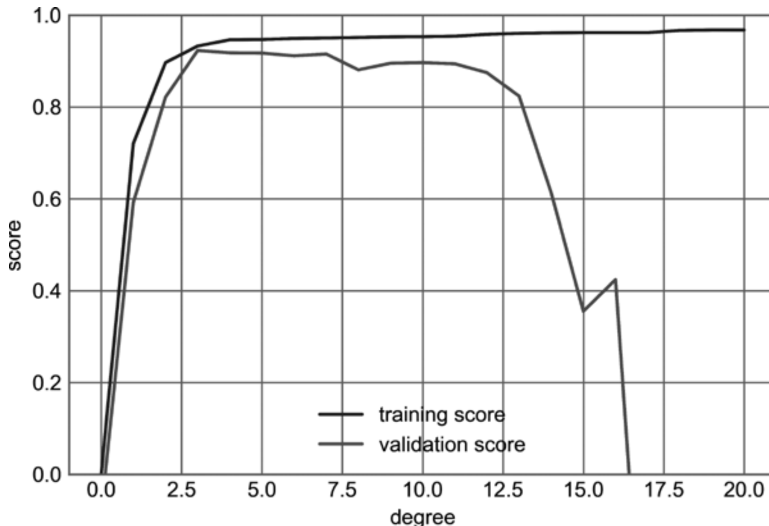


Рис. 39.7. Кривая проверки для приведенных на рис. 39.9 данных

Этот график в точности демонстрирует ожидаемое нами качественное поведение: оценка эффективности на обучающих данных на всем диапазоне превышает оценку эффективности на контрольных данных; первая монотонно растет с увеличением

сложности модели, а вторая достигает максимума и затем резко падает в точке, где наступает переобучение модели.

Как можно понять из приведенной кривой проверки, оптимальный компромисс между систематической ошибкой и дисперсией достигается при использовании многочлена третьей степени. Вычислить и отобразить на графике эту аппроксимацию исходных данных можно следующим образом (рис. 39.8):

```
In [14]: plt.scatter(X.ravel(), y)
         lim = plt.axis()
         y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
         plt.plot(X_test.ravel(), y_test);
         plt.axis(lim);
```

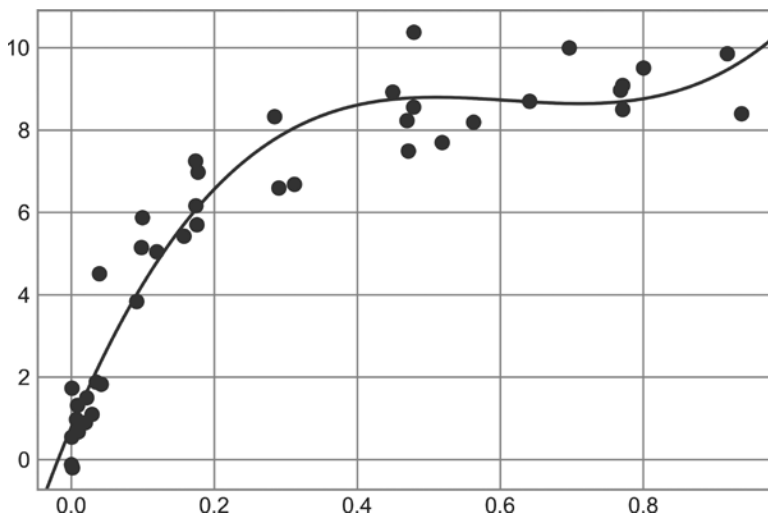


Рис. 39.8. Результат перекрестной проверки оптимальной модели для аппроксимации данных на рис. 39.6

Отмечу, что для нахождения оптимальной модели не требуется вычислять оценку эффективности на обучающих данных, но изучение зависимости между оценками на обучающих и контрольных данных дает нам полезную информацию относительно эффективности модели.

Кривые обучения

Важный нюанс сложности моделей состоит в том, что оптимальность модели обычно зависит от размера обучающей последовательности. Например, сгенерируем новый набор данных с количеством точек в пять раз больше (рис. 39.9):

```
In [15]: X2, y2 = make_data(200)
         plt.scatter(X2.ravel(), y2);
```

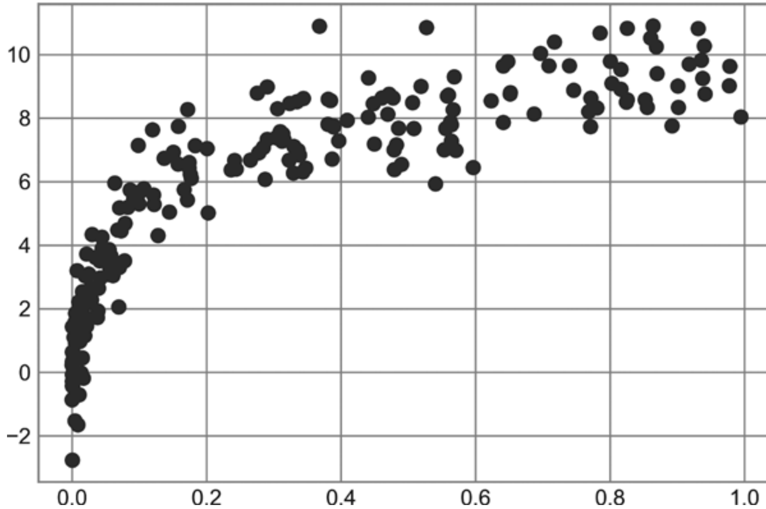


Рис. 39.9. Данные для демонстрации кривых обучения

Вновь построим график кривой обучения для этого большего набора данных. Для сравнения выведем также предыдущие результаты (рис. 39.10):

```
In [16]: degree = np.arange(21)
         train_score2, val_score2 = validation_curve(
             PolynomialRegression(), X2, y2,
             param_name='polynomialfeatures__degree',
             param_range=degree, cv=7)

         plt.plot(degree, np.median(train_score2, 1),
                  color='blue', label='training score')
         plt.plot(degree, np.median(val_score2, 1),
                  color='red', label='validation score')
         plt.plot(degree, np.median(train_score, 1),
                  color='blue', alpha=0.3, linestyle='dashed')
         plt.plot(degree, np.median(val_score, 1),
                  color='red', alpha=0.3, linestyle='dashed')
         plt.legend(loc='lower center')
         plt.ylim(0, 1)
         plt.xlabel('degree')
         plt.ylabel('score');
```

Сплошные линии показывают новые результаты, а более бледные пунктирные линии — результаты, полученные на меньшем наборе данных. Как можно заключить из кривой проверки, этот больший набор данных позволяет использовать намного

более сложную модель: максимум находится где-то в районе степени 6, но даже модель со степенью 20 не выглядит сильно переобученной — оценки эффективности на обучающих и контрольных данных остаются очень близкими друг к другу.

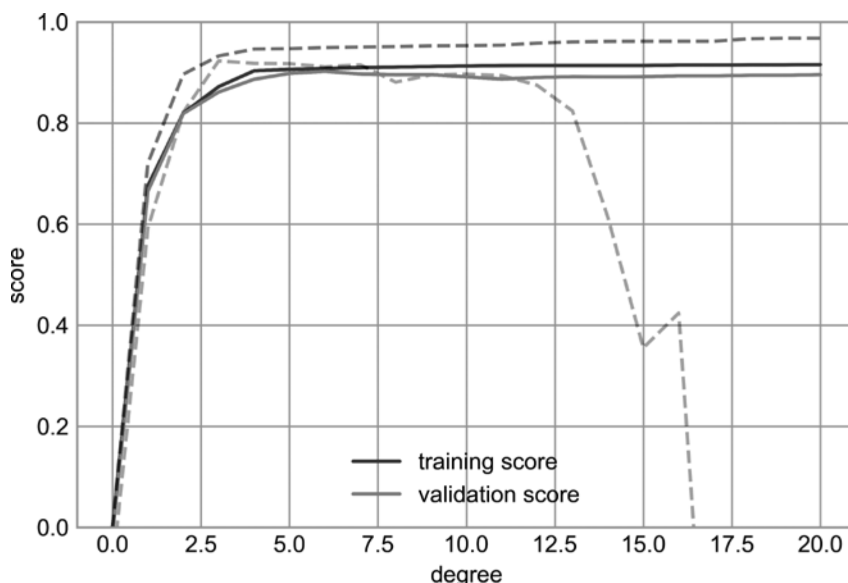


Рис. 39.10. Кривые обучения полиномиальной модели на данных, приведенных на рис. 39.9

Очевидно, что поведение кривой проверки зависит не от одного, а от двух важных факторов: сложности модели и количества точек обучения. Зачастую бывает полезно исследовать поведение модели как функции от количества точек обучения. Сделать это можно путем использования постепенно увеличивающихся подмножеств данных для обучения модели. График оценок на обучающих/контрольных данных с учетом размера обучающей последовательности называют *кривой обучения* (learning curve).

Ожидается следующее поведение кривой обучения.

- Если модель заданной сложности оказывается *переобученной*, то это означает, что набор данных слишком мал. При этом оценка эффективности на обучающих данных будет относительно высокой, а на контрольных данных — относительно низкой.
- Если модель заданной сложности остается *недообученной*, то это означает, что набор данных слишком велик. При этом оценка эффективности на обучающих данных будет снижаться, а на контрольных данных — повышаться с увеличением размера набора данных.

- Если модель никогда (разве что случайно) не показывает на контрольном наборе результат лучше, чем на обучающей последовательности, то кривые будут сближаться, но никогда не пересекутся.

Учитывая эти особенности, можно ожидать, что кривая обучения будет выглядеть качественно схожей с изображенной на рис. 39.11.

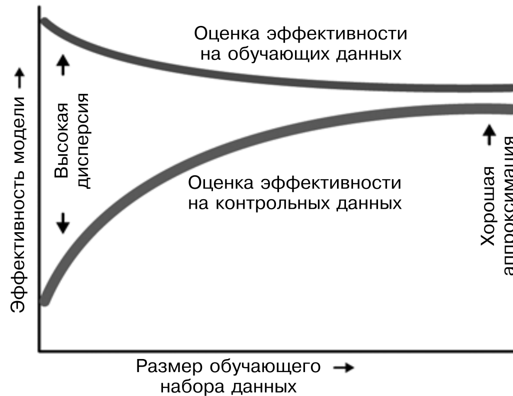


Рис. 39.11. Схематическое изображение типичной кривой обучения

Заметная особенность кривой обучения — сходимость к конкретному значению оценки при росте числа обучающих выборок. В частности, если количество точек достигло значения, при котором данная конкретная модель сошлась, то *добавление новых обучающих данных не поможет*! Единственным способом улучшить качество модели в этом случае будет использование другой (зачастую более сложной) модели.

Библиотека Scikit-Learn предоставляет удобные утилиты для вычисления кривых обучения моделей. Далее мы вычислим кривую обучения для нашего исходного набора данных с полиномиальными моделями второй и девятой степени (рис. 39.12):

```
In [17]: from sklearn.model_selection import learning_curve

         fig, ax = plt.subplots(1, 2, figsize=(16, 6))
         fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

         for i, degree in enumerate([2, 9]):
             N, train_lc, val_lc = learning_curve(
                 PolynomialRegression(degree), X, y, cv=7,
                 train_sizes=np.linspace(0.3, 1, 25))

             ax[i].plot(N, np.mean(train_lc, 1),
                        color='blue', label='training score')
```

```
    ax[i].plot(N, np.mean(val_lc, 1),
               color='red', label='validation score')
    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0],
                 N[-1], color='gray', linestyle='dashed')

    ax[i].set_ylim(0, 1)

    ax[i].set_xlim(N[0], N[-1])
    ax[i].set_xlabel('training size')
    ax[i].set_ylabel('score')
    ax[i].set_title('degree = {0}'.format(degree), size=14)
    ax[i].legend(loc='best')
```

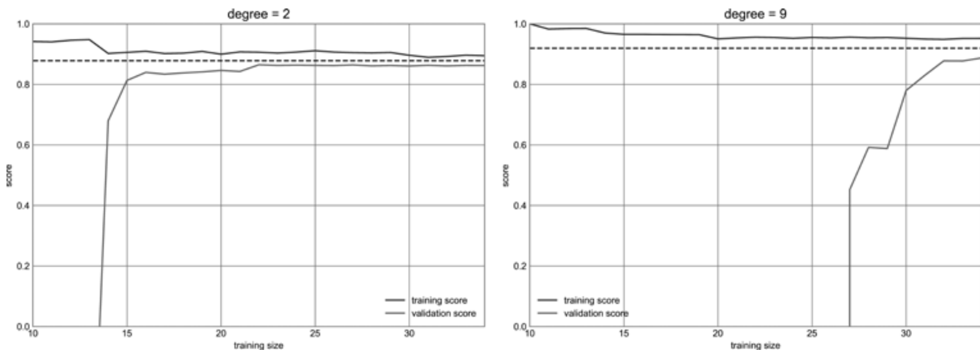


Рис. 39.12. Кривые обучения для модели низкой *(слева)* и высокой *(справа)* сложности

Это ценный показатель, поскольку он наглядно демонстрирует нам реакцию модели на увеличение объема обучающих данных. В частности, после того момента, когда кривая обучения уже сошлась к какому-то значению (то есть когда кривые эффективности, полученные на обучающих и контрольных данных, близки друг к другу), *добавление дополнительных обучающих данных не улучшит аппроксимацию существенно*! Эта ситуация отражена на левом рисунке с кривой обучения для модели второй степени.

Единственный способ улучшения оценки уже сошедшейся кривой — использовать другую (обычно более сложную) модель. Это видно на правом рисунке: перейдя к более сложной модели, мы улучшаем оценку для точки сходимости (отмеченную штриховой линией) за счет более высокой дисперсии модели (соответствующей расстоянию между оценками эффективности обучающих и контрольных данных). Если бы нам пришлось добавить еще больше точек, кривая обучения для более сложной из этих моделей все равно в итоге бы сошлась.

Построение графика кривой обучения для конкретных модели и набора данных облегчает принятие решения о том, как продвинуться еще дальше на пути улучшения анализа данных.

Проверка на практике: поиск по сетке

Из предшествующего обсуждения вы должны были понять смысл компромисса между систематической ошибкой и дисперсией и его зависимость от сложности модели и размера обучающей последовательности. На практике у моделей обычно больше одного параметра, поэтому графики кривых проверки и обучения превращаются из двумерных линий в многомерные поверхности. Отобразить такие поверхности визуально — непростая задача, поэтому лучше отыскать конкретную модель, при которой оценка эффективности на контрольных данных достигает максимума.

Библиотека Scikit-Learn предоставляет несколько инструментов, упрощающих такой поиск: далее мы разберем поиск по сетке и отыщем оптимальную полиномиальную модель. Рассмотрим двумерную сетку признаков модели — степени многочлена и флага, указывающего, нужно ли подбирать точку пересечения с осью координат. Выполнить эти настройки можно с помощью класса `GridSearchCV` из библиотеки Scikit-Learn:

```
In [18]: from sklearn.model_selection import GridSearchCV

         param_grid = {'polynomialfeatures__degree': np.arange(21),
                       'linearregression__fit_intercept': [True, False]}

         grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
```

Отмечу, что здесь этот класс еще не был применен к каким-либо данным. Обучение модели наряду с отслеживанием промежуточных оценок эффективности в каждой из точек сетки производится путем вызова метода `fit`:

```
In [19]: grid.fit(X, y);
```

После обучения можно узнать значения оптимальных параметров:

```
In [20]: grid.best_params_
Out[20]: {'linearregression__fit_intercept': False, 'polynomialfeatures__degree': 4}
```

При необходимости можно воспользоваться этой оптимальной моделью и продемонстрировать аппроксимацию с помощью уже использовавшегося нами кода (рис. 39.13):

```
In [21]: model = grid.best_estimator_

         plt.scatter(X.ravel(), y)
         lim = plt.axis()
         y_test = model.fit(X, y).predict(X_test)
         plt.plot(X_test.ravel(), y_test);
         plt.axis(lim);
```

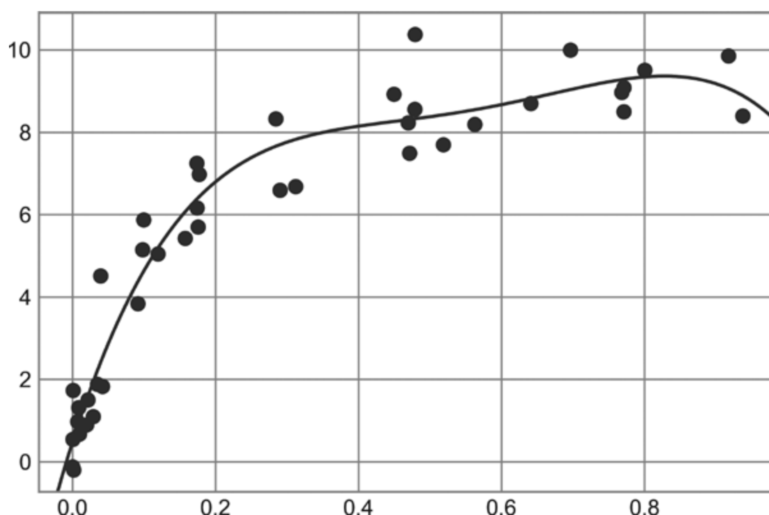


Рис. 39.13. Оптимальная модель, определенная посредством автоматического поиска по сетке

Поиск по сетке обладает множеством других возможностей, позволяя, кроме всего прочего, задавать свою функцию оценки эффективности, распараллеливать вычисления, выполнять случайный поиск и др. Для получения дополнительной информации см. примеры в главах 49 и 50 или обратитесь к документации библиотеки Scikit-Learn, посвященной поиску по сетке (https://oreil.ly/xft8j).

Резюме

В этой главе мы приступили к изучению понятий проверки модели и оптимизации гиперпараметров, фокусируя внимание на наглядных аспектах компромисса между систематической ошибкой и дисперсией и его работы при подгонке моделей к данным. В частности, мы обнаружили, что крайне важно использовать контрольный набор или перекрестную проверку при настройке параметров, чтобы избежать переобучения более сложных/гибких моделей.

В следующих главах мы детально рассмотрим некоторые особенно примечательные модели, попутно обсуждая имеющиеся возможности их настройки и влияние этих свободных параметров на сложность модели. Не забывайте уроки этой главы при дальнейшем чтении книги и изучении упомянутых методов машинного обучения!

ГЛАВА 40

Проектирование признаков

В предыдущих главах мы познакомились с базовыми понятиями машинного обучения, но во всех примерах предполагалось, что наши данные находятся строго в формате `[n_samples, n_features]`. На практике же данные редко поступают к нам в подобном виде. Поэтому одним из важнейших этапов использования машинного обучения на практике становится *проектирование признаков* (feature engineering), то есть преобразование всей информации, касающейся задачи, в числа, пригодные для построения матрицы признаков.

В этой главе мы рассмотрим несколько примеров часто встречающихся задач проектирования признаков: признаки для представления категориальных данных (categorical data), признаки для представления текста и признаки для представления изображений. Кроме того, мы обсудим использование производных признаков (derived features) для повышения сложности модели и заполнение отсутствующих данных. Этот процесс часто называют векторизацией, так как он включает преобразование данных из произвольной формы в аккуратные векторы.

Категориальные признаки

Категориальные данные — один из распространенных типов нечисловых данных. Например, допустим, что мы анализируем какие-то данные по ценам на жилье и, помимо числовых признаков, таких как «цена» и «количество комнат», в них имеется информация о микрорайоне (neighborhood). Например, пусть наши данные выглядят следующим образом:

```
In [1]: data = [
            {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
            {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
            {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
            {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
        ]
```

Может показаться соблазнительным закодировать эти данные, задав прямое числовое соответствие:

```
In [2]: {'Queen Anne': 1, 'Fremont': 2, 'Wallingford': 3};
```

Но оказывается, что в библиотеке Scikit-Learn такой подход не очень удобен: модели, реализованные в ней, исходят из базового допущения о том, что числовые признаки отражают алгебраические величины. Следовательно, подобное соответствие будет подразумевать, например, что *Queen Anne* < *Fremont* < *Wallingford* или даже что *Wallingford* – *Queen Anne* = *Fremont*, что, не считая сомнительных демографических шуток, не имеет никакого смысла.

Испытанным методом для такого случая является *прямое кодирование* (one-hot encoding), означающее создание дополнительных столбцов-индикаторов наличия/отсутствия категории с помощью значений 1 и 0 соответственно. При наличии данных в виде списка словарей для этой цели можно воспользоваться утилитой `DictVectorizer` из библиотеки Scikit-Learn:

```
In [3]: from sklearn.feature_extraction import DictVectorizer
        vec = DictVectorizer(sparse=False, dtype=int)
        vec.fit_transform(data)
Out[3]: array([[     0,      1,      0, 850000,      4],
               [     1,      0,      0, 700000,      3],
               [     0,      0,      1, 650000,      3],
               [     1,      0,      0, 600000,      2]])
```

Обратите внимание, что столбец `neighborhood` (микрорайон) превратился в три отдельных столбца, соответствующих названиям трех микрорайонов, и в каждой строке стоит 1 в столбце, относящемся к ее микрорайону. После подобного кодирования категориальных признаков можно продолжить обучение модели Scikit-Learn как обычно.

Чтобы узнать, что означает каждый столбец, можно посмотреть названия признаков:

```
In [4]: vec.get_feature_names_out()
Out[4]: array(['neighborhood=Fremont', 'neighborhood=Queen Anne',
               'neighborhood=Wallingford', 'price', 'rooms'], dtype=object)
```

У этого подхода имеется один очевидный недостаток: если количество значений категории велико, размер набора данных может значительно вырасти. Однако, поскольку кодированные данные состоят в основном из нулей, эффективным решением будет разреженный формат вывода:

```
In [5]: vec = DictVectorizer(sparse=True, dtype=int)
        vec.fit_transform(data)
Out[5]: <4x5 sparse matrix of type '<class 'numpy.int64'>'
                with 12 stored elements in Compressed Sparse Row format>
```

Многие (но не все) оцениватели в библиотеке Scikit-Learn готовы принять подобные разреженные входные данные для обучения и применения моделей. Для поддержки подобного кодирования библиотека Scikit-Learn включает две дополнительные утилиты: `sklearn.preprocessing.OneHotEncoder` и `sklearn.feature_extraction.FeatureHasher`.

Текстовые признаки

При проектировании признаков часто требуется преобразовывать текст в набор репрезентативных числовых значений. Например, в основе автоматических процедур извлечения данных из социальных медиа лежит определенный вид кодирования текста числовыми значениями. Один из простейших методов кодирования данных — по *количеству слов*: для каждого фрагмента текста подсчитывается количество вхождений в него каждого из слов, после чего результаты помещаются в таблицу.

Рассмотрим следующий набор из трех фраз:

```
In [6]: sample = ['problem of evil',
                  'evil queen',
                  'horizon problem']
```

Для векторизации этих данных на основе числа слов можно создать столбцы, соответствующие словам «problem», «evil», «horizon» и т. д. Это можно сделать вручную, но лучше использовать класс `CountVectorizer` из библиотеки Scikit-Learn, который избавит нас от нудной работы:

```
In [7]: from sklearn.feature_extraction.text import CountVectorizer

        vec = CountVectorizer()
        X = vec.fit_transform(sample)
        X
Out[7]: <3x5 sparse matrix of type '<class 'numpy.int64'>'
                with 7 stored elements in Compressed Sparse Row format>
```

Результат представляет собой разреженную матрицу, содержащую количество вхождений каждого из слов. Для удобства мы преобразуем ее в объект `DataFrame` с маркированными столбцами:

```
In [8]: import pandas as pd
        pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out())
Out[8]:    evil  horizon  of  problem  queen
        0     1        0   1        1      0
        1     1        0   0        0      1
        2     0        1   0        1      0
```

Однако этот подход страдает некоторыми проблемами: использование непосредственно количеств слов ведет к признакам, которые придают слишком большое значение словам, встречающимся очень часто, а это в некоторых алгоритмах классификации может отрицательно сказаться на оптимальности. Один из подходов к решению этой проблемы известен под названием *«частота слова — обратная частота документа»* (term frequency — inverse document frequency) или TF-IDF. При его использовании слова получают вес с учетом частоты их появления во всех документах. Синтаксис вычисления этих признаков аналогичен предыдущему примеру:

```
In [9]: from sklearn.feature_extraction.text import TfidfVectorizer
        vec = TfidfVectorizer()
        X = vec.fit_transform(sample)
        pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out())
Out[9]:        evil   horizon        of   problem     queen
        0  0.517856  0.000000  0.680919  0.517856  0.000000
        1  0.605349  0.000000  0.000000  0.000000  0.795961
        2  0.000000  0.795961  0.000000  0.605349  0.000000
```

Пример использования TF-IDF в задачах классификации вы найдете в главе 41.

Признаки для изображений

Достаточно часто для задач машинного обучения требуется соответствующим образом закодировать изображения. Простейший подход — тот, который мы применяли для набора данных рукописных цифр в главе 38, — использовать значения самих пикселов. Но подобные подходы в зависимости от прикладной задачи могут оказаться неоптимальными.

Всесторонний обзор методов выделения признаков для изображений выходит далеко за рамки данной главы, но вы можете найти отличные реализации стандартных подходов в проекте Scikit-Image (http://scikit-image.org/). Впрочем, один пример совместного использования библиотеки Scikit-Learn и пакета Scikit-Image вы найдете в главе 50.

Производные признаки

Еще один удобный тип признаков — выведенные математически из каких-либо входных признаков. Мы уже встречались с ними в главе 39, когда создавали *полиномиальные признаки* из входных данных. Мы видели, что линейную регрессию можно преобразовать в полиномиальную не путем изменения модели, а преобразованием входных данных!

Например, очевидно, что следующие данные нельзя адекватно аппроксимировать прямой линией (рис. 40.1):

```
In [10]: %matplotlib inline
         import numpy as np
         import matplotlib.pyplot as plt

         x = np.array([1, 2, 3, 4, 5])
         y = np.array([4, 2, 1, 3, 7])
         plt.scatter(x, y);
```

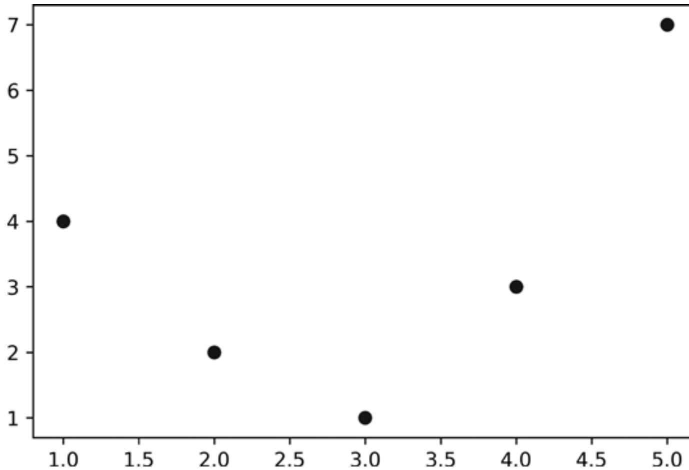


Рис. 40.1. Данные, которые нельзя адекватно аппроксимировать прямой линией

Но мы можем подобрать разделяющую прямую для этих данных с помощью функции `LinearRegression` и получить оптимальный результат (рис. 40.2).

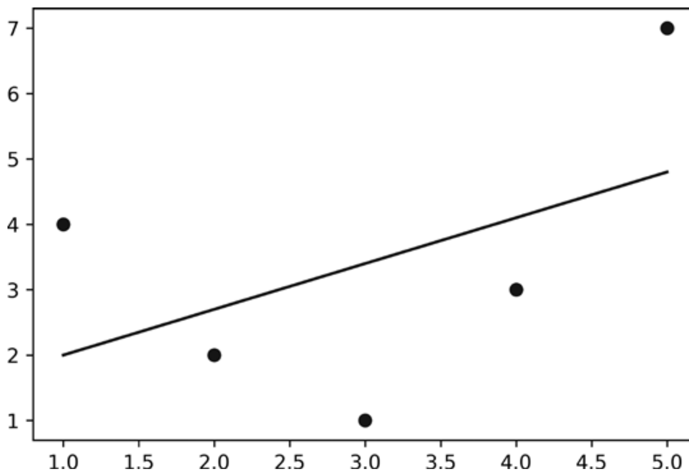


Рис. 40.2. Неудачная аппроксимация прямой линией

```
In [11]: from sklearn.linear_model import LinearRegression
            X = x[:, np.newaxis]
            model = LinearRegression().fit(X, y)
            yfit = model.predict(X)
            plt.scatter(x, y)
            plt.plot(x, yfit);
```

Очевидно, что для описания зависимости между x и y требуется более сложная модель. Добиться желаемого можно путем преобразования данных, добавив дополнительные столбцы признаков для увеличения гибкости модели. Добавить в данные полиномиальные признаки можно так:

```
In [12]: from sklearn.preprocessing import PolynomialFeatures
         poly = PolynomialFeatures(degree=3, include_bias=False)
         X2 = poly.fit_transform(X)
         print(X2)
Out[12]: [[  1.   1.   1.]
          [  2.   4.   8.]
          [  3.   9.  27.]
          [  4.  16.  64.]
          [  5.  25. 125.]]
```

В матрице производных признаков один столбец соответствует x, второй — x^2, а третий — x^3. Расчет линейной регрессии для этих расширенных входных данных позволяет получить намного лучшую аппроксимацию (рис. 40.3).

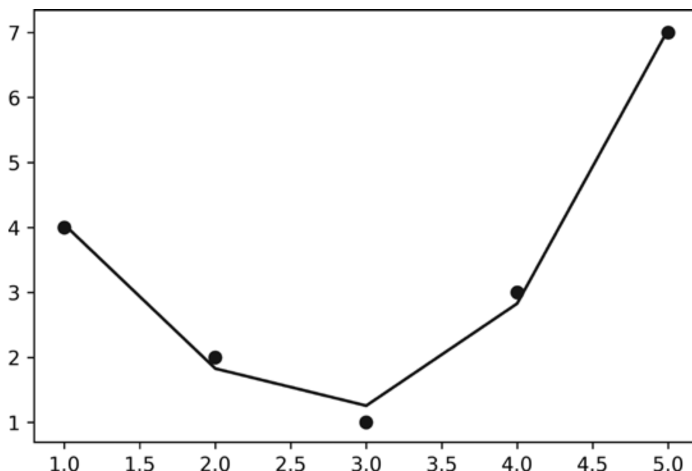


Рис. 40.3. Линейная аппроксимация по производным полиномиальным признакам

```
In [13]: model = LinearRegression().fit(X2, y)
            yfit = model.predict(X2)
            plt.scatter(x, y)
            plt.plot(x, yfit);
```

Идея улучшения модели путем не изменения самой модели, а преобразования входных данных является базовой для многих более продвинутых методов машинного обучения. Мы обсудим эту идею подробнее в главе 42 в контексте *регрессии по комбинации базисных функций*. В общем случае этот путь ведет к набору методик, обладающих огромными возможностями и известных под общим названием *«ядерные методы»* (kernel methods), которые мы рассмотрим в главе 43.

Подстановка отсутствующих данных

Еще одна часто встречающаяся задача в проектировании признаков — обработка отсутствующих данных. Мы уже обсуждали этот вопрос, когда исследовали объекты `DataFrame` в главе 16. Там мы видели, что отсутствующие данные часто отмечаются значением `NaN`. Например, наш набор данных может выглядеть следующим образом:

```
In [14]: from numpy import nan
         X = np.array([[ nan, 0,   3  ],
                       [ 3,   7,   9  ],
                       [ 3,   5,   2  ],
                       [ 4,   nan, 6  ],
                       [ 8,   8,   1  ]])
         y = np.array([14, 16, -1,  8, -5])
```

При использовании для подобных данных типичной модели машинного обучения необходимо сначала заменить отсутствующие данные каким-либо подходящим значением. Это действие называется *подстановкой* (imputation) пропущенных значений, и методики его выполнения варьируются от простых (например, замены пропущенных значений средним значением по столбцу) до сложных (например, с использованием матриц восстановления (matrix completion) или устойчивого к ошибкам алгоритма обработки подобных данных).

Сложные подходы, как правило, сильно зависят от конкретной прикладной задачи, и мы не станем углубляться в них. Для реализации стандартного подхода к подстановке пропущенных значений (с использованием среднего значения, медианы или часто встречающегося значения) библиотека Scikit-Learn предоставляет класс `SimpleImputer`:

```
In [15]: from sklearn.impute import SimpleImputer
         imp = SimpleImputer(strategy='mean')
         X2 = imp.fit_transform(X)
         X2
Out[15]: array([[4.5, 0. , 3. ],
                [3. , 7. , 9. ],
                [3. , 5. , 2. ],
                [4. , 5. , 6. ],
                [8. , 8. , 1. ]])
```

В результате мы получили данные, где вместо пропущенных значений подставлены средние значения остальных элементов столбца. Эти данные можно передать, например, непосредственно классу `LinearRegression`:

```
In [16]: model = LinearRegression().fit(X2, y)
         model.predict(X2)
Out[16]: array([13.14869292, 14.3784627 , -1.15539732, 10.96606197, -5.33782027])
```

Конвейеры признаков

Во всех предыдущих примерах может быстро надоесть выполнять преобразования вручную, особенно если нужно связать в цепочку несколько шагов. Например, нам может понадобиться следующий конвейер обработки.

1. Заменить отсутствующие данные средними значениями.
2. Преобразовать простые признаки в квадратичные.
3. Обучить модель линейной регрессии.

Для организации потоковой обработки подобного конвейера библиотека Scikit-Learn предоставляет объект `Pipeline`, который можно использовать так:

```
In [17]: from sklearn.pipeline import make_pipeline

         model = make_pipeline(SimpleImputer(strategy='mean'),
                               PolynomialFeatures(degree=2),
                               LinearRegression())
```

Этот конвейер выглядит и функционирует подобно обычному объекту библиотеки Scikit-Learn и выполняет все заданные шаги для любых входных данных.

```
In [18]: model.fit(X, y)  # Вышеприведенный массив X с пропущенными значениями
         print(y)
         print(model.predict(X))
Out[18]: [14 16 -1  8 -5]
         [14. 16. -1.  8. -5.]
```

Все шаги этой модели выполняются автоматически. Обратите внимание, что для простоты демонстрации мы применили модель к тем данным, на которых она была обучена, именно поэтому она сумела столь хорошо предсказать результат (более подробное обсуждение этого вопроса можно найти в главе 39).

Некоторые примеры работы конвейеров библиотеки Scikit-Learn вы увидите в следующей главе, посвященной наивной байесовской классификации, а также в главах 42 и 43.

ГЛАВА 41

Заглянем глубже: наивная байесовская классификация

Предыдущие четыре главы были посвящены общему обзору принципов машинного обучения. В остальных главах части V мы сначала рассмотрим первые четыре алгоритма обучения с учителем, а затем четыре алгоритма обучения без учителя. Первым мы исследуем алгоритм наивной байесовской классификации.

Наивные байесовские модели — группа исключительно быстрых и простых алгоритмов классификации, зачастую подходящих для наборов данных очень высоких размерностей. В силу их быстроты и небольшого количества настраиваемых параметров они оказываются очень удобны для получения первого приближенного решения задач классификации. В этой главе мы наглядно объясним работу наивных байесовских классификаторов вместе на нескольких примерах их применения к некоторым наборам данных.

Байесовская классификация

Наивные байесовские классификаторы основаны на байесовских методах классификации, в основе которых лежит теорема Байеса — уравнение, описывающее связь условных вероятностей статистических величин. В байесовской классификации нас интересует поиск вероятности метки (категории) L при определенных заданных признаках, которую можно записать как $P(L \mid \text{признаков})$. Теорема Байеса позволяет выразить это в терминах величин, которые можно вычислить напрямую:

$$P(L \mid \text{признаков}) = \frac{P(\text{признаков} \mid L)\, P(L)}{P(\text{признаков})}.$$

Один из способов выбора между двумя метками (L_1 и L_2) — вычислить отношение апостериорных вероятностей для каждой из них:

$$\frac{P(L_1 \mid \text{признаков})}{P(L_2 \mid \text{признаков})} = \frac{P(\text{признаков} \mid L_1)\; P(L_1)}{P(\text{признаков} \mid L_2)\; P(L_2)}.$$

Все, что нам теперь нужно, — модель, с помощью которой можно было бы вычислить $P(\text{признаков} \mid L_i)$ для каждой из меток. Подобная модель называется *генеративной моделью* (generative model), поскольку определяет гипотетический случайный процесс генерации данных. Задание генеративной модели для каждой из меток/категорий — основа обучения подобного байесовского классификатора. Обобщенная версия подобного шага обучения — непростая задача, но мы упростим ее, приняв некоторые упрощающие допущения о виде модели.

Именно на этом этапе возникает слово «наивный» в названии «наивный байесовский классификатор»: сделав очень «наивное» допущение относительно генеративной модели для каждой из меток/категорий, можно отыскать грубое приближение генеративной модели для каждого класса, после чего перейти к байесовской классификации. Различные виды наивных байесовских классификаторов основываются на различных «наивных» допущениях относительно данных, мы рассмотрим несколько из них в следующих разделах.

Начнем с импорта необходимых пакетов:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        plt.style.use('seaborn-whitegrid')
```

Гауссов наивный байесовский классификатор

Вероятно, самый простой для понимания наивный байесовский классификатор — гауссов. В этом классификаторе допущение состоит в том, что *данные всех категорий взяты из простого нормального распределения*. Пускай у нас имеются следующие данные (рис. 41.1):

```
In [2]: from sklearn.datasets import make_blobs
        X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
```

Идея одной из самых простых гауссовых моделей заключается в допущении, что данные подчиняются нормальному распределению без ковариации между измере-

ниями. Для обучения этой модели достаточно найти среднее значение и стандартное отклонение точек внутри каждой из категорий — это все, что требуется для описания подобного распределения. Результат этого наивного гауссова допущения показан на рис. 41.2.

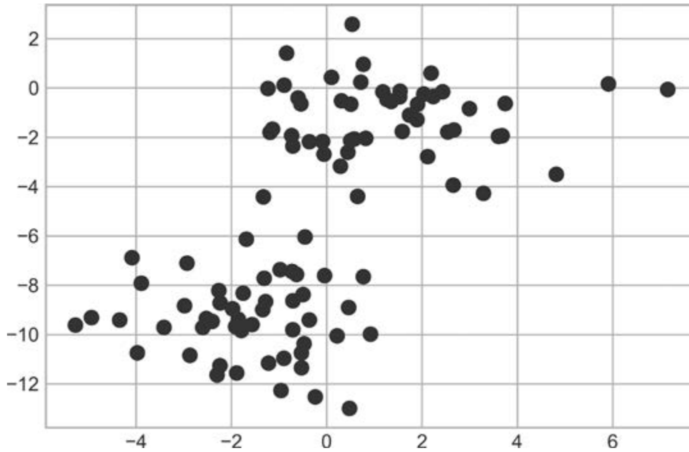


Рис. 41.1. Данные для наивной байесовской классификации

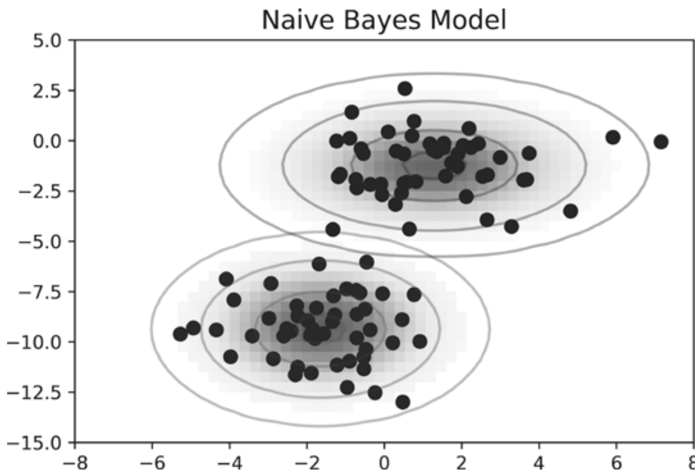


Рис. 41.2. Визуализация гауссовой наивной байесовской модели[1]

[1] Код, генерирующий рисунки этой главы, можно найти в онлайн-приложении (https://oreil.ly/jv0wb).

Эллипсы на этом рисунке представляют гауссову генеративную модель для каждой из меток с ростом вероятности по мере приближения к центру эллипса. С помощью этой генеративной модели для каждого класса мы можем легко вычислить вероятность $P(\text{признаков} \mid L_i)$ для каждой точки данных, а следовательно, быстро рассчитать соотношение для апостериорной вероятности и определить, какая из меток с большей вероятностью соответствует конкретной точке.

Эта процедура реализована в классе `sklearn.naive_bayes.GaussianNB`:

```
In [3]: from sklearn.naive_bayes import GaussianNB
        model = GaussianNB()
        model.fit(X, y);
```

Давайте сгенерируем какие-нибудь новые данные и выполним предсказание метки:

```
In [4]: rng = np.random.RandomState(0)
        Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
        ynew = model.predict(Xnew)
```

Теперь можно построить график, отражающий эти новые данные, и понять, где пролегает граница принятия решений (рис. 41.3):

```
In [5]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
        lim = plt.axis()
        plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu', alpha=0.1)
        plt.axis(lim);
```

Мы видим, что граница слегка изогнута. В целом граница, получаемая гауссовым наивным байесовским классификатором, соответствует кривой второго порядка.

Положительная сторона этого байесовского формального представления заключается в возможности естественной вероятностной классификации, рассчитать которую можно с помощью метода `predict_proba`:

```
In [6]: yprob = model.predict_proba(Xnew)
        yprob[-8:].round(2)
Out[6]: array([[0.89, 0.11],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [0.  , 1.  ],
               [0.15, 0.85]])
```

Столбцы отражают апостериорные вероятности первой и второй меток соответственно. Подобные байесовские методы могут оказаться неплохой отправной точкой для оценки погрешностей в классификации.

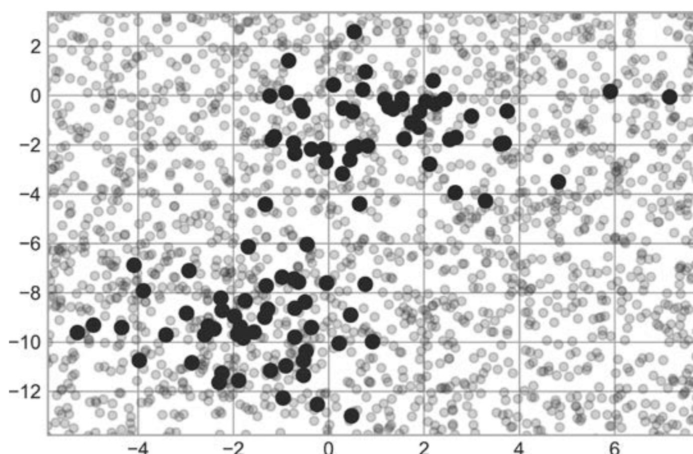


Рис. 41.3. Визуализация гауссовой наивной байесовской классификации

Качество получаемой в итоге классификации не может превышать качества исходных допущений модели, поэтому гауссов наивный байесовский классификатор зачастую не демонстрирует слишком хороших результатов. Тем не менее во многих случаях — особенно при значительном количестве признаков — исходные допущения не настолько плохи, чтобы нивелировать удобство гауссова наивного байесовского классификатора.

Полиномиальный наивный байесовский классификатор

Гауссово допущение — далеко не единственное простое допущение, которое можно использовать для описания генеративного распределения для всех меток. Еще один интересный пример — полиномиальный наивный байесовский классификатор с допущением, что признаки сгенерированы на основе простого полиномиального распределения. Полиномиальное распределение описывает вероятность наблюдения количеств вхождений в несколько категорий, поэтому полиномиальный наивный байесовский классификатор лучше всего подходит для признаков, отражающих количество или частоту вхождения.

Основная идея остается той же, но вместо моделирования распределения данных с оптимальной гауссовой функцией мы моделируем распределение данных с оптимальным полиномиальным распределением.

Пример: классификация текста

Полиномиальный наивный байесовский классификатор нередко используется при классификации текста, где признаки соответствуют количеству слов или частотам их употребления в классифицируемых документах. Мы уже обсуждали вопрос извлечения подобных признаков из текста в главе 40. Здесь же, чтобы продемонстрировать классификацию коротких документов по категориям, мы воспользуемся разреженными признаками количеств слов из корпуса текста 20 Newsgroups («20 дискуссионных групп»).

Скачаем данные и изучим целевые названия:

```
In [7]: from sklearn.datasets import fetch_20newsgroups

        data = fetch_20newsgroups()
        data.target_names
Out[7]: ['alt.atheism',
         'comp.graphics',
         'comp.os.ms-windows.misc',
         'comp.sys.ibm.pc.hardware',
         'comp.sys.mac.hardware',
         'comp.windows.x',
         'misc.forsale',
         'rec.autos',
         'rec.motorcycles',
         'rec.sport.baseball',
         'rec.sport.hockey',
         'sci.crypt',
         'sci.electronics',
         'sci.med',
         'sci.space',
         'soc.religion.christian',
         'talk.politics.guns',
         'talk.politics.mideast',
         'talk.politics.misc',
         'talk.religion.misc']
```

Для простоты выберем лишь несколько из этих категорий, после чего скачаем обучающую и контрольную последовательности:

```
In [8]: categories = ['talk.religion.misc', 'soc.religion.christian',
                      'sci.space', 'comp.graphics']
        train = fetch_20newsgroups(subset='train', categories=categories)
        test = fetch_20newsgroups(subset='test', categories=categories)
```

Вот типичный образец записи из этого набора данных:

```
In [9]: print(train.data[5][48:])
Out[9]: Subject: Federal Hearing
        Originator: dmcgee@uluhe
        Organization: School of Ocean and Earth Science and Technology
        Distribution: usa
        Lines: 10

        Fact or rumor....?  Madalyn Murray O'Hare an atheist who eliminated the
        use of the bible reading and prayer in public schools 15 years ago is now
        going to appear before the FCC with a petition to stop the reading of the
        Gospel on the airways of America.  And she is also campaigning to remove
        Christmas programs, songs, etc from the public schools.  If it is true
        then mail to Federal Communications Commission 1919 H Street Washington DC
        20054 expressing your opposition to her request.  Reference Petition number

        2493.
```

Чтобы использовать эти данные для машинного обучения, необходимо преобразовать содержимое каждой строки в числовой вектор. Для этого воспользуемся векторизатором TF-IDF (который обсуждали в главе 40) и создадим конвейер, подключающий его последовательно к полиномиальному наивному байесовскому классификатору:

```
In [10]: from sklearn.feature_extraction.text import TfidfVectorizer
         from sklearn.naive_bayes import MultinomialNB
         from sklearn.pipeline import make_pipeline

         model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

Используя этот конвейер, можно применить модель к обучающей последовательности и предсказать метки для контрольных данных:

```
In [11]: model.fit(train.data, train.target)
         labels = model.predict(test.data)
```

Теперь, предсказав метки, изучим их и выясним эффективность модели. Например, вот матрица различий между настоящими и предсказанными метками для контрольных данных (рис. 41.4):

```
In [12]: from sklearn.metrics import confusion_matrix
         mat = confusion_matrix(test.target, labels)
         sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                     xticklabels=train.target_names, yticklabels=train.target_names,
                     cmap='Blues')
         plt.xlabel('true label')       # истинная метка
         plt.ylabel('predicted label'); # прогнозируемая метка
```

Рис. 41.4. Матрица различий для полиномиального наивного байесовского классификатора текста

Даже этот очень простой классификатор смог отделить обсуждения космоса от дискуссий о компьютерах, но он путает обсуждения религии вообще и обсуждения христианства. Вероятно, этого следовало ожидать!

Самое замечательное, что у нас теперь есть инструмент определения категории для любой строки с помощью метода `predict` нашего конвейера. Следующий фрагмент кода описывает простую вспомогательную функцию, возвращающую предсказание для отдельной строки:

```
In [13]: def predict_category(s, train=train, model=model):
             pred = model.predict([s])
             return train.target_names[pred[0]]
```

Давайте опробуем ее:

```
In [14]: predict_category('sending a payload to the ISS')
Out[14]: 'sci.space'
```

```
In [15]: predict_category('discussing the existence of God')
Out[15]: 'soc.religion.christian'

In [16]: predict_category('determining the screen resolution')
Out[16]: 'comp.graphics'
```

Это лишь простая вероятностная модель (взвешенной) частоты каждого из слов в строке, тем не менее результат поразителен. Даже очень наивный алгоритм может оказаться удивительно эффективным при разумном использовании и обучении на большом наборе многомерных данных.

Когда имеет смысл использовать наивный байесовский классификатор

В силу столь смелых допущений относительно данных наивные байесовские классификаторы обычно работают хуже, чем более сложные модели. Тем не менее они имеют несколько достоинств:

- выполняют обучение и предсказание исключительно быстро;
- обеспечивают простое вероятностное предсказание;
- их результаты часто очень легко интерпретировать;
- у них очень мало (если вообще есть) настраиваемых параметров.

Эти достоинства означают, что наивный байесовский классификатор зачастую оказывается удачным кандидатом на роль первоначальной грубой классификации. Если он демонстрирует удовлетворительные результаты, то поздравляем: вы нашли для своей задачи очень быстрый классификатор, возвращающий простые для интерпретации результаты. Если же нет, то вы всегда можете попробовать применить более сложные модели, уже имея представление о том, насколько хорошо они должны работать.

Наивные байесовские классификаторы склонны демонстрировать особенно хорошие результаты в следующих случаях:

- когда данные действительно соответствуют наивным допущениям (на практике бывает очень редко);
- для сильно отличающихся категорий, когда сложность модели не столь важна;
- для данных с очень большим числом измерений, когда сложность модели не столь важна.

Два последних случая кажутся не связанными друг с другом, но на самом деле это не так: с увеличением количества измерений в наборе данных вероятность

обнаружить близость любых двух точек уменьшается (в конце концов, чтобы находиться рядом, они должны находиться рядом по всем измерениям). Это значит, что кластеры в многомерных случаях имеют склонность к более выраженной изоляции, чем кластеры в случаях с меньшим количеством измерений (конечно, если новые измерения действительно вносят дополнительную информацию). Поэтому упрощенные классификаторы, такие как наивный байесовский классификатор, с ростом количества измерений начинают работать не хуже, а то и лучше более сложных: когда данных достаточно много, даже простая модель может оказаться весьма эффективной.

ГЛАВА 42

Заглянем глубже: линейная регрессия

Подобно тому как наивный байесовский классификатор (обсуждавшийся в главе 41) может служить отличной отправной точкой для задач классификации, так и модели линейной регрессии могут служить хорошей отправной точкой для задач регрессии. Подобные модели очень популярны, потому что быстро обучаются и возвращают простые для интерпретации результаты. Вероятно, вы уже знакомы с простейшей формой модели линейной регрессии (прямой линией, разделяющей данные), но такие модели можно распространить на данные с более сложной организацией.

В этой главе мы начнем с краткого и интуитивно понятного математического описания известной задачи, а потом посмотрим, как можно обобщить линейные модели, чтобы они аппроксимировали более сложные закономерности в данных.

Начнем с импортирования необходимых модулей:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Простая линейная регрессия

Начнем с линейной регрессии, аппроксимирующей данные прямой линией. Модель прямолинейной аппроксимации имеет вид:

$$y = ax + b,$$

где a — это *угловой коэффициент*, а b — *точка пересечения* с осью координат Y.

Рассмотрим следующие данные, разбросанные вдоль прямой с угловым коэффициентом 2 и точкой пересечения –5 (рис. 42.1):

```
In [2]: rng = np.random.RandomState(1)
        x = 10 * rng.rand(50)
        y = 2 * x - 5 + rng.randn(50)
        plt.scatter(x, y);
```

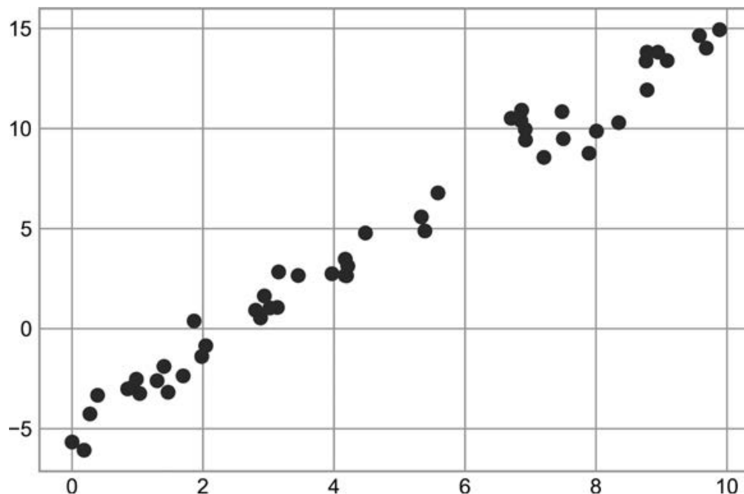


Рис. 42.1. Данные для линейной регрессии

Используем класс `LinearRegression` из библиотеки Scikit-Learn для обучения на этих данных и поиска оптимальной прямой (рис. 42.2):

```
In [3]: from sklearn.linear_model import LinearRegression
        model = LinearRegression(fit_intercept=True)

        model.fit(x[:, np.newaxis], y)

        xfit = np.linspace(0, 10, 1000)
        yfit = model.predict(xfit[:, np.newaxis])

        plt.scatter(x, y)
        plt.plot(xfit, yfit);
```

Подбираемые параметры модели (их имена в библиотеке Scikit-Learn всегда завершаются знаком подчеркивания) включают угловой коэффициент и точку пересечения с осью координат. В данном случае это параметры `coef_` и `intercept_`:

```
In [4]: print("Model slope:    ", model.coef_[0])
        print("Model intercept:", model.intercept_)
Out[4]: Model slope:     2.0272088103606953
        Model intercept: -4.998577085553204
```

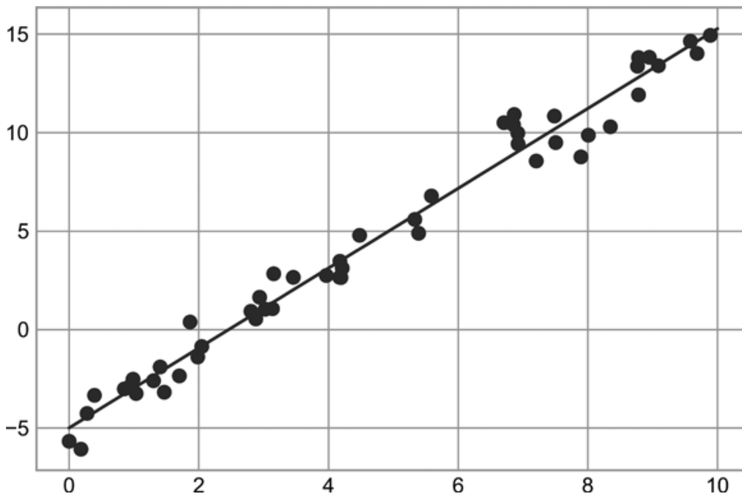


Рис. 42.2. Модель простой линейной регрессии

Как видите, результаты очень близки к параметрам, использовавшимся для генерирования данных, как мы и надеялись.

Однако возможности класса `LinearRegression` намного шире: помимо аппроксимации прямыми линиями, он может также работать с многомерными линейными моделями вида:

$$y = a_0 + a_1x_1 + a_2x_2 + ...$$

с несколькими величинами x. Геометрически это подобно подбору плоскости для точек в трех измерениях или гиперплоскости для точек в пространстве с бóльшим числом измерений.

Многомерная сущность подобных регрессий усложняет их визуализацию, но мы можем посмотреть на одну из этих аппроксимаций в действии, создав данные для нашего примера с помощью оператора матричного умножения из библиотеки NumPy:

```
In [5]: rng = np.random.RandomState(1)
        X = 10 * rng.rand(100, 3)
        y = 0.5 + np.dot(X, [1.5, -2., 1.])
        model.fit(X, y)
        print(model.intercept_)
        print(model.coef_)
Out[5]: 0.50000000000001
        [ 1.5 -2.   1. ]
```

Здесь значение y формируется как линейная комбинация из трех случайных значений x, а линейная регрессия восстанавливает использовавшиеся для формирования коэффициенты.

Аналогичным образом можно использовать класс `LinearRegression` для аппроксимации данных прямыми, плоскостями и гиперплоскостями. По-прежнему складывается впечатление, что этот подход ограничивается лишь строго линейными отношениями между переменными, но оказывается, что это требование можно ослабить.

Регрессия по комбинации базисных функций

Один из трюков, позволяющих адаптировать линейную регрессию к нелинейным отношениям между переменными, — преобразование данных в соответствии с новыми *базисными функциями*. Один из вариантов этого трюка мы уже встречали в конвейере `PolynomialRegression`, который использовался в главах 39 и 40. Идея состоит в том, чтобы взять многомерную линейную модель:

$$y = a_0 + a_1x_1 + a_2x_2 + a_3x_3 + \ldots$$

и построить x_1, x_2, x_3 и т. д. на основе имеющегося одномерного входного значения x. То есть у нас $x_n = f_n(x)$, где $f_n(x)$ — некая функция, выполняющая преобразование данных.

Например, если $f_n(x) = x^n$, то наша модель превращается в полиномиальную регрессию:

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$$

Обратите внимание, что модель *по-прежнему остается линейной* — линейность относится к тому факту, что коэффициенты a_n никогда не умножаются и не делятся друг на друга. Фактически мы взяли одномерные значения x и спроецировали их в пространство с большим числом измерений, и теперь с помощью линейной модели можем аппроксимировать более сложные зависимости между x и y.

Полиномиальные базисные функции

Данная полиномиальная проекция настолько удобна, что была встроена в библиотеку Scikit-Learn в виде класса преобразователя `PolynomialFeatures`:

```
In [6]: from sklearn.preprocessing import PolynomialFeatures
        x = np.array([2, 3, 4])
        poly = PolynomialFeatures(3, include_bias=False)
        poly.fit_transform(x[:, None])
Out[6]: array([[ 2.,  4.,  8.],
               [ 3.,  9., 27.],
               [ 4., 16., 64.]])
```

Как видите, преобразователь превратил наш одномерный массив в трехмерный, в котором каждый столбец представляет определенную степень исходного значения. Это новое многомерное представление данных можно далее использовать для линейной регрессии.

Как мы уже видели в главе 40, самый изящный способ добиться желаемого — воспользоваться конвейером. Давайте создадим с его помощью полиномиальную модель седьмого порядка:

```
In [7]: from sklearn.pipeline import make_pipeline
        poly_model = make_pipeline(PolynomialFeatures(7),
                                   LinearRegression())
```

После завершения всех преобразований можно воспользоваться линейной моделью для подбора намного более сложных зависимостей между величинами x и y. Например, рассмотрим зашумленную синусоиду (рис. 42.3):

```
In [8]: rng = np.random.RandomState(1)
        x = 10 * rng.rand(50)
        y = np.sin(x) + 0.1 * rng.randn(50)

        poly_model.fit(x[:, np.newaxis], y)
        yfit = poly_model.predict(xfit[:, np.newaxis])

        plt.scatter(x, y)
        plt.plot(xfit, yfit);
```

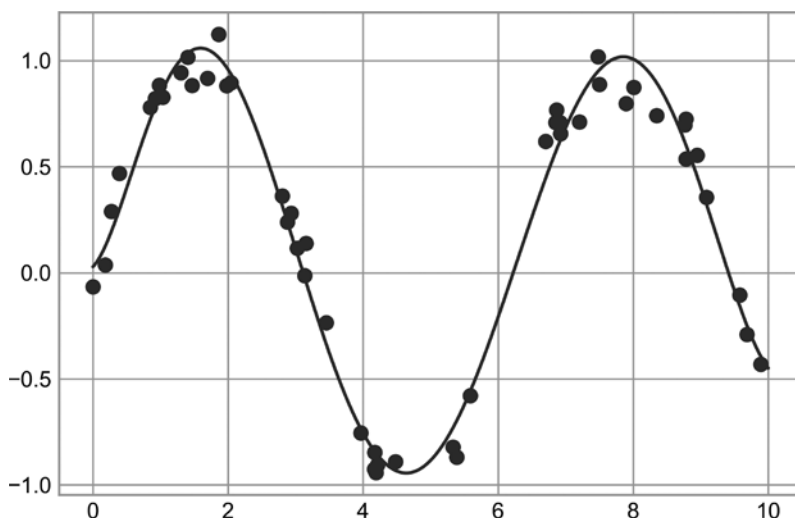


Рис. 42.3. Полиномиальная аппроксимация нелинейной обучающей последовательности

Наша линейная модель благодаря использованию полиномиальных базисных функций седьмого порядка смогла обеспечить великолепную аппроксимацию этих нелинейных данных!

Гауссовы базисные функции

Конечно, другие базисные функции тоже можно использовать. Например, один из полезных паттернов — обучение модели, являющейся суммой не полиномиальных, а гауссовых базисных функций. Результат будет выглядеть следующим образом (рис. 42.4):

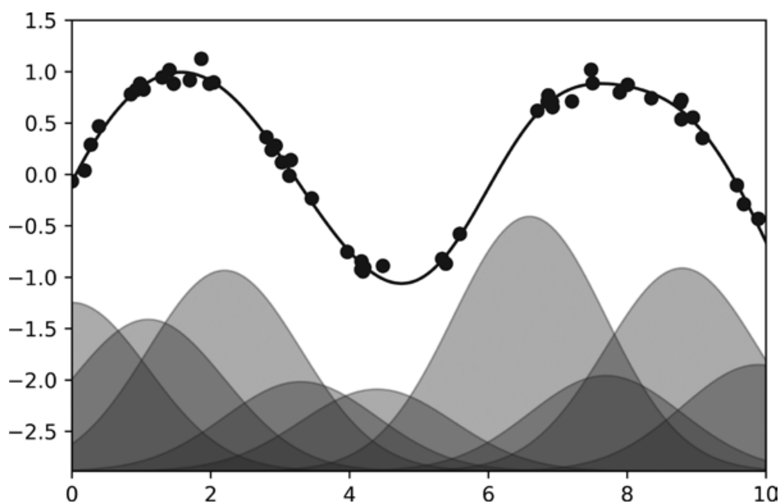


Рис. 42.4. Аппроксимация нелинейных данных с помощью гауссовых базисных функций

Затененные области на рис. 42.4 — нормированные базисные функции, дающие при сложении гладкую кривую, аппроксимирующую данные. Эти гауссовы базисные функции не встроены в библиотеку Scikit-Learn, но мы можем написать для их создания свой преобразователь, как показано ниже и проиллюстрировано на рис. 42.5 (преобразователи в библиотеке Scikit-Learn реализованы как классы; чтение исходного кода библиотеки Scikit-Learn — отличный способ разобраться с их созданием):

```
In [9]: from sklearn.base import BaseEstimator, TransformerMixin

        class GaussianFeatures(BaseEstimator, TransformerMixin):
            """Равномерно распределенные гауссовы признаки
               для одномерных входных данных"""
```

```
    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # Создает N центров, распределенных по всему диапазону данных
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor*(self.centers_[1]-self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                 self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```

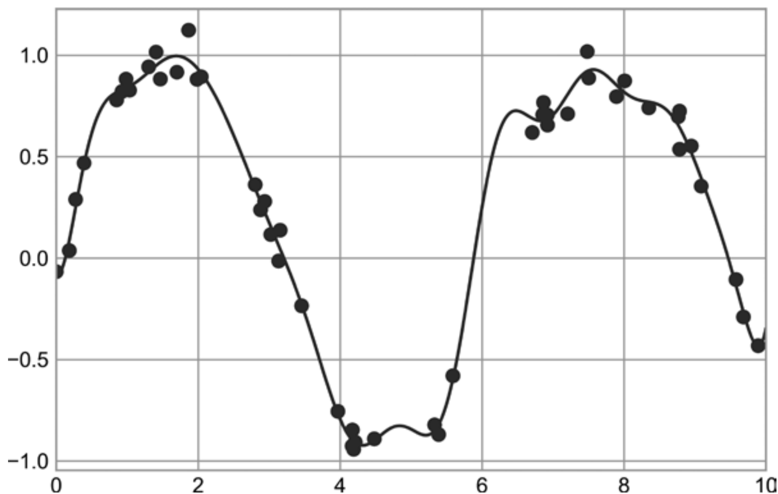


Рис. 42.5. Аппроксимация гауссовыми базисными функциями, вычисленными с помощью нестандартного преобразователя

Я привел этот пример лишь для того, чтобы подчеркнуть, что в полиномиальных базисных функциях нет никакого колдовства. Если у вас есть какие-то дополнительные сведения о процессе генерирования данных, исходя из которых вы полагаете, что наиболее подходящим будет тот или иной базис, то его тоже можно использовать.

Регуляризация

Применение базисных функций в линейной модели делает ее намного гибче, но также быстро приводит к переобучению (эта проблема обсуждалась в главе 39). Например, выбрав слишком много гауссовых базисных функций, мы в итоге получим не слишком хорошие результаты (рис. 42.6):

```
In [10]: model = make_pipeline(GaussianFeatures(30),
                               LinearRegression())
         model.fit(x[:, np.newaxis], y)

         plt.scatter(x, y)
         plt.plot(xfit, model.predict(xfit[:, np.newaxis]))

         plt.xlim(0, 10)
         plt.ylim(-1.5, 1.5);
```

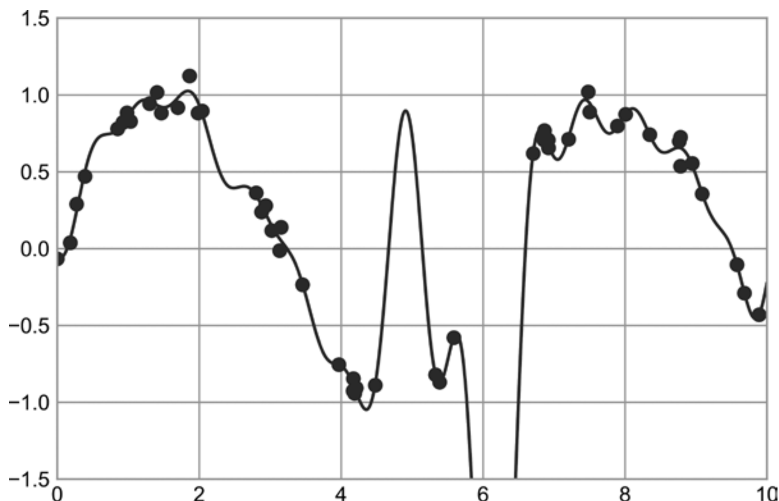


Рис. 42.6. Пример переобучения на данных: слишком сложная модель с базисными функциями

В результате проекции данных на 30-мерный базис модель оказалась слишком уж гибкой и стремится к экстремальным значениям в промежутках между точками,

в которых она ограничена данными. Причину этого можно понять, построив график коэффициентов гауссовых базисных функций в соответствии с координатой x (рис. 42.7):

```
In [11]: def basis_plot(model, title=None):
             fig, ax = plt.subplots(2, sharex=True)
             model.fit(x[:, np.newaxis], y)
             ax[0].scatter(x, y)
             ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
             ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

             if title:
                 ax[0].set_title(title)

             ax[1].plot(model.steps[0][1].centers_,
                        model.steps[1][1].coef_)
             ax[1].set(xlabel='basis location',
                       ylabel='coefficient',
                       xlim=(0, 10))
         model = make_pipeline(GaussianFeatures(30), LinearRegression())
         basis_plot(model)
```

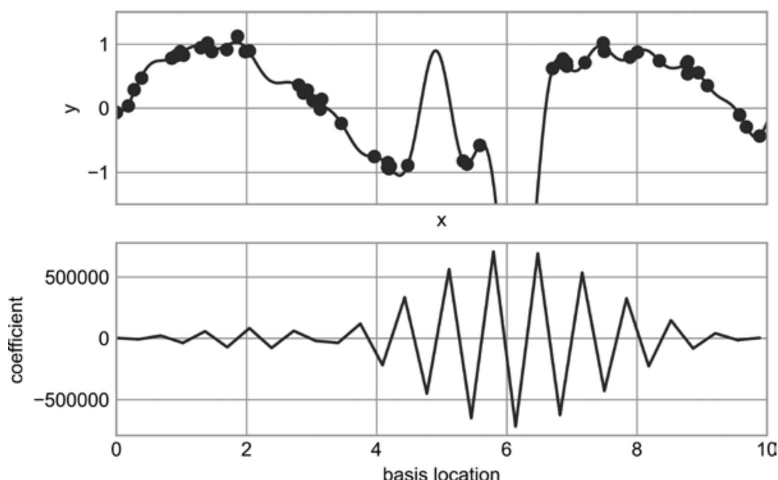


Рис. 42.7. Коэффициенты при гауссовых базисных функциях в чрезмерно сложной модели

Нижний график на рис. 42.7 демонстрирует амплитуду базисной функции в каждой из точек. Это типичное поведение при переобучении с перекрытием областей определения базисных функций: коэффициенты соседних базисных функций усиливают и подавляют друг друга. Мы знаем, что подобное поведение приводит к проблемам и было бы неплохо явно ограничивать подобные пики в модели, «накладывая штраф» на большие значения параметров. Подобное «штрафование» известно как *регуляризация* и существует в нескольких вариантах.

Гребневая регрессия (L_2-регуляризация)

Вероятно, самый часто встречающийся вид регуляризации — *гребневая регрессия* (ridge regression), или *L2-регуляризация* (L_2-regularization), также иногда называемая *регуляризацией Тихонова* (Tikhonov regularization). Она заключается в наложении штрафа на сумму квадратов (евклидовой нормы) коэффициентов модели θ_n. В данном случае штраф для модели будет равен:

$$P = \alpha \sum_{n=1}^{N} \theta_n^2,$$

где α — свободный параметр, управляющий величиной штрафа. Этот тип модели со штрафом встроен в библиотеку Scikit-Learn в виде класса `Ridge` (рис. 42.8):

```
In [12]: from sklearn.linear_model import Ridge
         model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
         basis_plot(model, title='Ridge Regression')
```

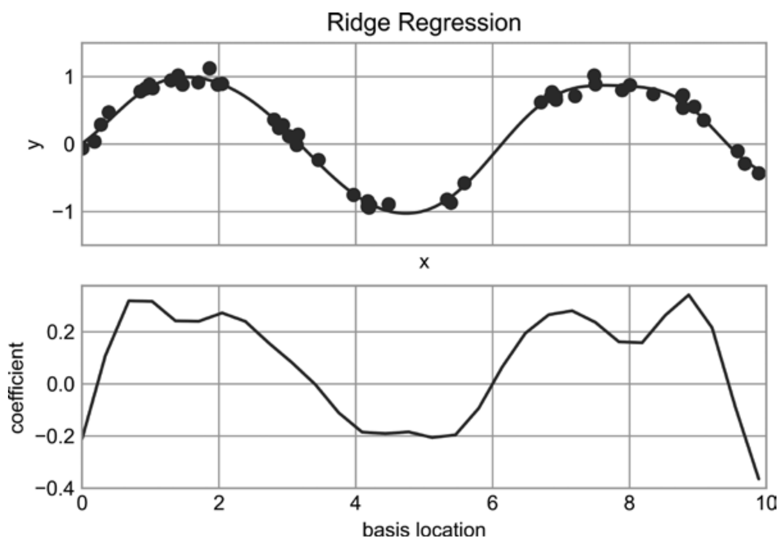


Рис. 42.8. Применение гребневой (L_2) регуляризации к слишком сложной модели (ср. с рис. 42.7)

Параметр α управляет сложностью получаемой в итоге модели. В предельном случае $\alpha \to 0$ мы получаем результат, соответствующий стандартной линейной регрессии; в предельном случае $\alpha \to \infty$ будет происходить подавление любого отклика модели. Достоинства гребневой регрессии включают, помимо прочего, возможность ее эффективного вычисления — вычислительные затраты практически не превышают затрат на вычисление исходной линейной регрессионной модели.

Лассо-регрессия (L_1-регуляризация)

Еще один распространенный тип регуляризации — так называемая *лассо-регрессия*, или *L_1-регуляризация*, включающая штраф за сумму абсолютных значений (L_1-норма) коэффициентов регрессии:

$$P = \alpha\sum_{n=1}^{N}\left|\theta_n\right|.$$

Хотя концептуально эта регрессия очень близка к гребневой, результаты их могут очень сильно различаться. Например, по геометрическим причинам лассо-регрессия отдает предпочтение *разреженным моделям*, то есть по возможности делает коэффициенты модели равными нулю.

Чтобы посмотреть на поведение этой регрессии, повторим предыдущий пример, но на этот раз используем коэффициенты, нормализованные с помощью нормы L_1 (рис. 42.9):

```
In [13]: from sklearn.linear_model import Lasso
         model = make_pipeline(GaussianFeatures(30),
                               Lasso(alpha=0.001, max_iter=2000))
         basis_plot(model, title='Lasso Regression')
```

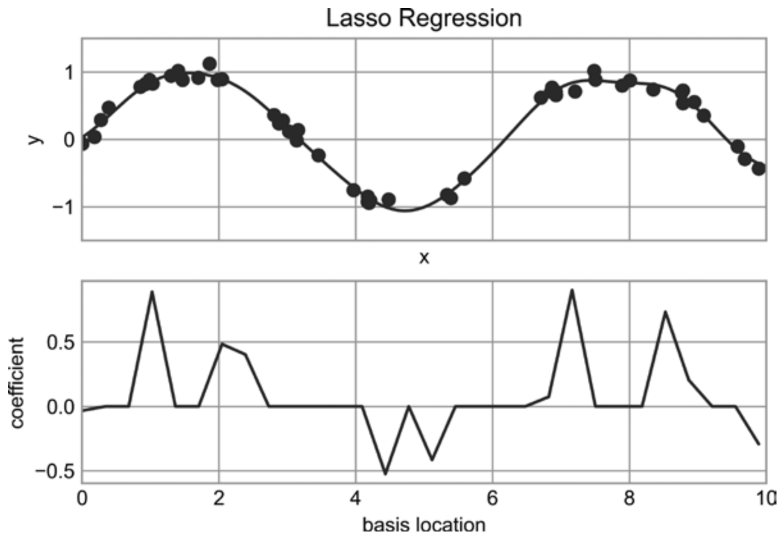


Рис. 42.9. Применение лассо-регуляризации (L_1) к слишком сложной модели (ср. с рис. 42.8)

При использовании лассо-регрессии большинство коэффициентов получаются точно равными нулю, а функциональное поведение моделируется небольшим

подмножеством из числа имеющихся базисных функций. Как и в случае гребневой регуляризации, параметр α управляет величиной штрафа и его следует определять путем перекрестной проверки (см. главу 39).

Пример: предсказание велосипедного трафика

В качестве примера посмотрим, сможем ли мы предсказать количество велосипедистов, пересекающих Фримонтский мост в Сиэтле, основываясь на данных о погоде, времени года и других факторах. Мы уже работали с этими данными в главе 23. В этом разделе мы соединим данные, описывающие велосипедный трафик, с другим набором данных и попробуем установить, насколько погода и сезонные факторы — температура, осадки и продолжительность светового дня — влияют на велосипедный трафик по этому коридору. К счастью, Национальное управление по исследованию океанов и атмосферы (National Oceanic and Atmospheric Administration, NOAA) открыло доступ к ежедневным данным с их метеорологических станций (https://oreil.ly/sE5zO) — я воспользовался станцией с ID USW00024233, — а библиотека Pandas дает возможность с легкостью соединить эти два источника данных. Мы попробуем выявить зависимость велосипедного трафика от погоды, выполнив простую линейную регрессию, чтобы оценить, как изменения этих параметров повлияют на число велосипедистов в заданный день.

В частности, этот пример демонстрирует, как можно использовать инструменты Scikit-Learn в статистическом моделировании, предполагающем осмысленность параметров модели. Это отнюдь не стандартный подход для машинного обучения, но для некоторых моделей подобная трактовка возможна.

Начнем с загрузки двух наборов данных, индексированных по дате:

```
In [14]: # url = 'https://raw.githubusercontent.com/jakevdp/bicycle-data/main'
         # !curl -O {url}/FremontBridge.csv
         # !curl -O {url}/SeattleWeather.csv

In [15]: import pandas as pd
         counts = pd.read_csv('FremontBridge.csv',
                              index_col='Date', parse_dates=True)
         weather = pd.read_csv('SeattleWeather.csv',
                               index_col='DATE', parse_dates=True)
```

Возьмем данные до 2020 года, чтобы исключить влияние на результаты последствий пандемии COVID-19, которая существенно изменила маршруты поездок в Сиэтле:

```
In [16]: counts = counts[counts.index < "2020-01-01"]
         weather = weather[weather.index < "2020-01-01"]
```

Далее вычислим общий ежедневный поток велосипедистов и поместим эти данные в отдельный объект DataFrame:

```
In [17]: daily = counts.resample('d').sum()
         daily['Total'] = daily.sum(axis=1)
         daily = daily[['Total']] # удаляем остальные столбцы
```

Мы уже знаем, что трафик варьируется день ото дня. Учтем это в наших данных, добавив двоичные столбцы-индикаторы дня недели:

```
In [18]: days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
         for i in range(7):
             daily[days[i]] = (daily.index.dayofweek == i).astype(float)
```

Следует ожидать, что велосипедисты будут вести себя иначе по выходным. Добавим индикаторы и для этого случая:

```
In [19]: from pandas.tseries.holiday import USFederalHolidayCalendar
         cal = USFederalHolidayCalendar()
         holidays = cal.holidays('2012', '2020')
         daily = daily.join(pd.Series(1, index=holidays, name='holiday'))
         daily['holiday'].fillna(0, inplace=True)
```

Логично предположить, что на трафик велосипедистов будет также влиять продолжительность светового дня. Воспользуемся стандартными астрономическими расчетами, чтобы добавить эту информацию (рис. 42.10):

```
In [20]: def hours_of_daylight(date, axis=23.44, latitude=47.61):
             """Вычисляет длительность светового дня для заданной даты"""
             days = (date - pd.datetime(2000, 12, 21)).days
             m = (1. - np.tan(np.radians(latitude))
                  * np.tan(np.radians(axis) * np.cos(days * 2 * np.pi / 365.25)))
             return 24. * np.degrees(np.arccos(1 - np.clip(m, 0, 2))) / 180.

         daily['daylight_hrs'] = list(map(hours_of_daylight, daily.index))
         daily[['daylight_hrs']].plot()
         plt.ylim(8, 17)
Out[20]: (8.0, 17.0)
```

Также добавим к данным среднюю температуру и общее количество осадков. Помимо количества дюймов осадков, добавим еще и флаг, обозначающий засушливые дни (с нулевым количеством осадков):

```
In [21]: weather['Temp (F)'] = 0.5 * (weather['TMIN'] + weather['TMAX'])
         weather['Rainfall (in)'] = weather['PRCP']
         weather['dry day'] = (weather['PRCP'] == 0).astype(int)

         daily = daily.join(weather[['Rainfall (in)', 'Temp (F)', 'dry day']])
```

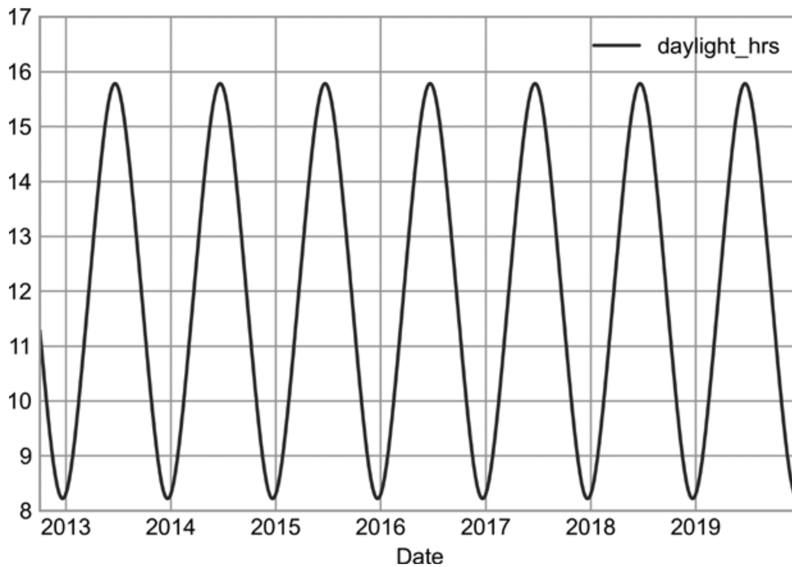


Рис. 42.10. Визуализация длительности светового дня в Сиэтле

Наконец, добавим счетчик, который будет увеличиваться начиная с первого дня и отмерять количество прошедших лет. Он позволит нам отслеживать ежегодные увеличения или уменьшения ежедневного количества проезжающих велосипедистов:

```
In [22]: daily['annual'] = (daily.index - daily.index[0]).days / 365.
```

На этом подготовка данных завершена, и мы можем посмотреть на них:

```
In [23]: daily.head()
Out[23]:               Total  Mon  Tue  Wed  Thu  Fri  Sat  Sun  holiday \
         Date
         2012-10-03  14084.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0      0.0
         2012-10-04  13900.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0      0.0
         2012-10-05  12592.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0      0.0
         2012-10-06   8024.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0      0.0
         2012-10-07   8568.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0      0.0

                     daylight_hrs Rainfall (in)  Temp (F)  dry day    annual
         Date
         2012-10-03     11.277359           0.0      56.0        1  0.000000
         2012-10-04     11.219142           0.0      56.5        1  0.002740
         2012-10-05     11.161038           0.0      59.5        1  0.005479
         2012-10-06     11.103056           0.0      60.5        1  0.008219
         2012-10-07     11.045208           0.0      60.5        1  0.010959
```

Теперь можно выбрать нужные столбцы и обучить линейную регрессионную модель. Зададим параметр `fit_intercept = False`, поскольку флаги для дней, по сути, выполняют подбор точек пересечения с осями координат по дням:

```
In [24]: # Отбросим все строки с пустыми значениями
         daily.dropna(axis=0, how='any', inplace=True)

         column_names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun',
                         'holiday', 'daylight_hrs', 'Rainfall (in)',
                         'dry day', 'Temp (F)', 'annual']
         X = daily[column_names]
         y = daily['Total']

         model = LinearRegression(fit_intercept=False)
         model.fit(X, y)
         daily['predicted'] = model.predict(X)
```

По завершении сравним фактический и предсказанный моделью велосипедный трафик визуально (рис. 42.11):

```
In [25]: daily[['Total', 'predicted']].plot(alpha=0.5);
```

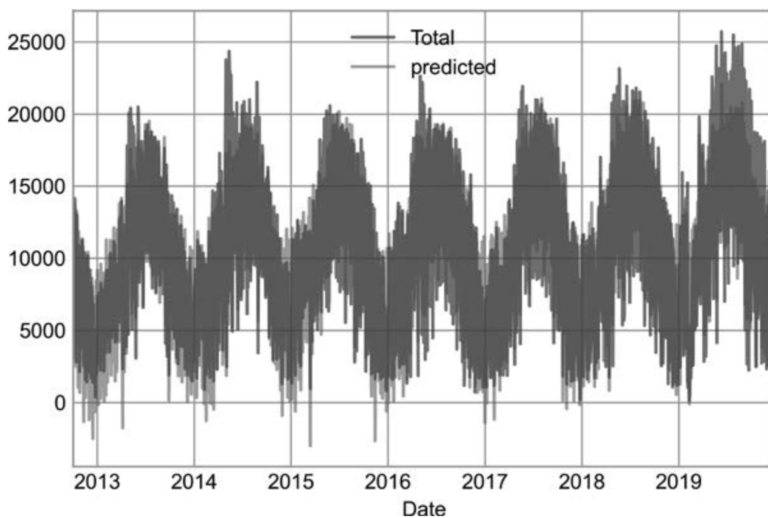


Рис. 42.11. Предсказанные моделью значения велосипедного трафика

Из того факта, что трафик, предсказанный моделью, недостаточно точно совпадает с фактическим, можно сделать вывод, что мы упустили какие-то ключевые признаки, особенно характерные для летнего времени. Или список наших признаков неполон (то есть люди принимают решение о том, ехать ли на работу на велосипеде,

основываясь не только на них), или имеются какие-то нелинейные зависимости, которые нам не удалось учесть (например, возможно, что люди ездят реже как в жару, так и в холод). Тем не менее нашей грубой аппроксимации достаточно, чтобы понять основные характеристики данных, и мы можем посмотреть на коэффициенты нашей линейной модели, чтобы оценить, какой вклад вносит в ежедневное количество поездок на велосипедах каждый из признаков:

```
In [26]: params = pd.Series(model.coef_, index=X.columns)
         params
Out[26]: Mon             -3309.953439
         Tue             -2860.625060
         Wed             -2962.889892
         Thu             -3480.656444
         Fri             -4836.064503
         Sat            -10436.802843
         Sun            -10795.195718
         holiday         -5006.995232
         daylight_hrs      409.146368
         Rainfall (in)   -2789.860745
         dry day          2111.069565
         Temp (F)          179.026296
         annual            324.437749
         dtype: float64
```

Эти числа нелегко интерпретировать в отсутствие какой-либо меры их неопределенности. Быстро вычислить погрешности можно путем бутстрэппинга — повторных выборок данных:

```
In [27]: from sklearn.utils import resample
         np.random.seed(1)
         err = np.std([model.fit(*resample(X, y)).coef_
                       for i in range(1000)], 0)
```

Оценив эти ошибки, взглянем на результаты еще раз:

```
In [28]: print(pd.DataFrame({'effect': params.round(0),
                             'uncertainty': err.round(0)}))
Out[28]:                  effect  uncertainty
         Mon             -3310.0        265.0
         Tue             -2861.0        274.0
         Wed             -2963.0        268.0
         Thu             -3481.0        268.0
         Fri             -4836.0        261.0
         Sat            -10437.0        259.0
         Sun            -10795.0        267.0
         holiday         -5007.0        401.0
         daylight_hrs      409.0         26.0
         Rainfall (in)   -2790.0        186.0
         dry day          2111.0        101.0
         Temp (F)          179.0          7.0
         annual            324.0         22.0
```

Столбец `effect` показывает, как изменение рассматриваемой функции влияет на количество велосипедистов. Например, существует четкое разделение по дням недели: по выходным дням число велосипедистов меньше на тысячи, чем по будням. Можно также заметить, что с каждым дополнительным часом светлого времени суток велосипедистов становится больше на 409 ± 26; рост температуры на 1 градус Фаренгейта стимулирует 179 ± 7 человек сесть на велосипед; отсутствие осадков дает прирост в среднем на 2111 ± 101 велосипедист; каждый дюйм осадков ведет к тому, что 2790 ± 186 человек выбирают другой способ транспортировки. После учета всех влияний мы получаем умеренный рост ежедневного количества велосипедистов на 324 ± 22 человека в год.

Нашей модели почти наверняка недостает информации, относящейся к делу. Например, нелинейные влияния (такие как совместное влияние осадков *и* низкой температуры) и нелинейные тренды в пределах каждой из переменных (такие как нежелание ездить на велосипедах в очень холодную и очень жаркую погоду) не могут быть учтены в этой модели. Кроме того, мы отбросили некоторые нюансы (такие как различие между дождливым утром и дождливым полуднем), а также проигнорировали корреляции между днями (такие как возможное влияние дождливого вторника на показатели среды или влияние внезапного солнечного дня после полосы дождливых). Это все очень интересные влияния, и у вас, если захотите, теперь есть инструменты для их исследования!

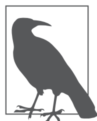
ГЛАВА 43

Заглянем глубже: метод опорных векторов

Метод опорных векторов (support vector machines, SVMs) — очень мощный и гибкий класс алгоритмов обучения с учителем как для классификации, так и для регрессии. В этой главе мы узнаем, как использовать метод опорных векторов в задачах классификации.

Начнем с импортирования необходимых модулей:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        from scipy import stats
```



Полноразмерные и полноцветные рисунки доступны в онлайн-приложении на GitHub (https://oreil.ly/PDSH_GitHub).

Основания для использования метода опорных векторов

В ходе обсуждения байесовской классификации (см. главу 41) мы познакомились с простой моделью, описывающей распределение всех базовых классов, и попробовали использовать ее для вероятностного определения меток для новых точек. Это был пример *генеративной классификации* (generative classification), здесь же мы рассмотрим *разделяющую классификацию* (discriminative classification). Вместо моделирования каждого из классов мы найдем прямую или кривую (в двумерном

пространстве) или многообразие (в многомерном пространстве), отделяющее классы друг от друга.

В качестве примера рассмотрим простую задачу классификации разделения двух классов точек (рис. 43.1):

```
In [2]: from sklearn.datasets import make_blobs
        X, y = make_blobs(n_samples=50, centers=2,
                          random_state=0, cluster_std=0.60)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

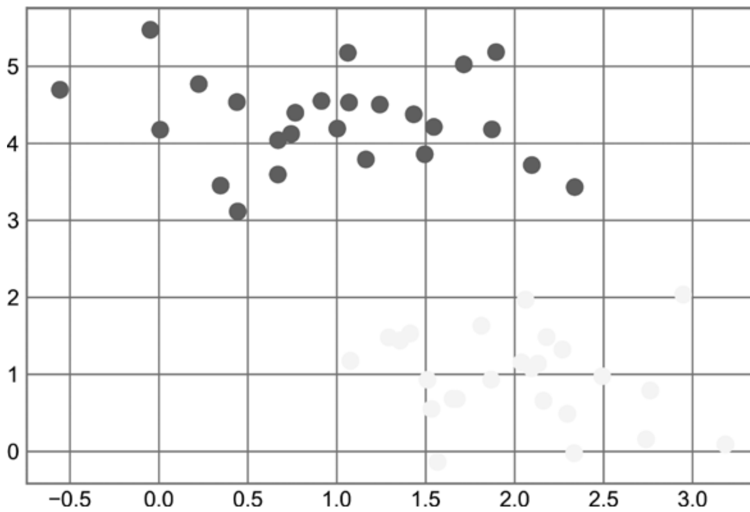


Рис. 43.1. Простые данные для классификации

Линейный разделяющий классификатор попытается провести прямую линию, разделяющую два набора данных, создав тем самым модель классификации. Для таких двумерных данных эту задачу можно решить вручную. Однако сразу же возникает проблема: существует более одной прямой, идеально разделяющей два класса!

Нарисуем их (рис. 43.2):

```
In [3]: xfit = np.linspace(-1, 3.5)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

        for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
            plt.plot(xfit, m * xfit + b, '-k')

        plt.xlim(-1, 3.5);
```

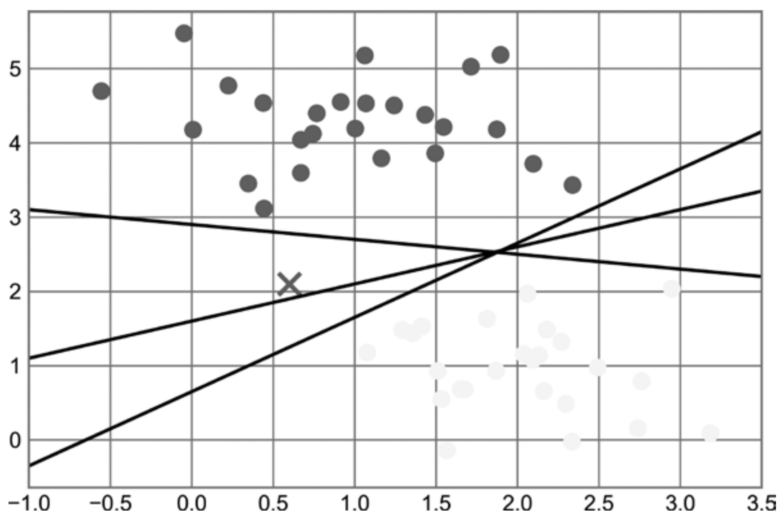


Рис. 43.2. Три идеальных линейных разделяющих классификатора для наших данных

Эти три разделителя очень разные, и все они прекрасно разделяют наши образцы. Однако в зависимости от выбора разделителя новой точке данных (например, отмеченной знаком «X» на рис. 43.2) могут быть присвоены разные метки! Очевидно, что интуитивный подход с «проведением прямой между классами» работает недостаточно хорошо и нужно подойти к вопросу более основательно.

Метод опорных векторов: максимизация отступа

Метод опорных векторов предлагает один из способов решения этой проблемы. Идея заключается в следующем: вместо рисования между классами прямой нулевой толщины можно около каждой из прямых нарисовать *отступ* (margin) некоторой ширины, простирающийся до ближайшей точки. Вот пример, как подобный подход мог бы выглядеть (рис. 43.3):

```
In [4]: xfit = np.linspace(-1, 3.5)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

        for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
            yfit = m * xfit + b
            plt.plot(xfit, yfit, '-k')
            plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
                             color='lightgray', alpha=0.5)

        plt.xlim(-1, 3.5);
```

Линия с максимальным отступом и будет той, которую мы выберем в качестве оптимальной модели.

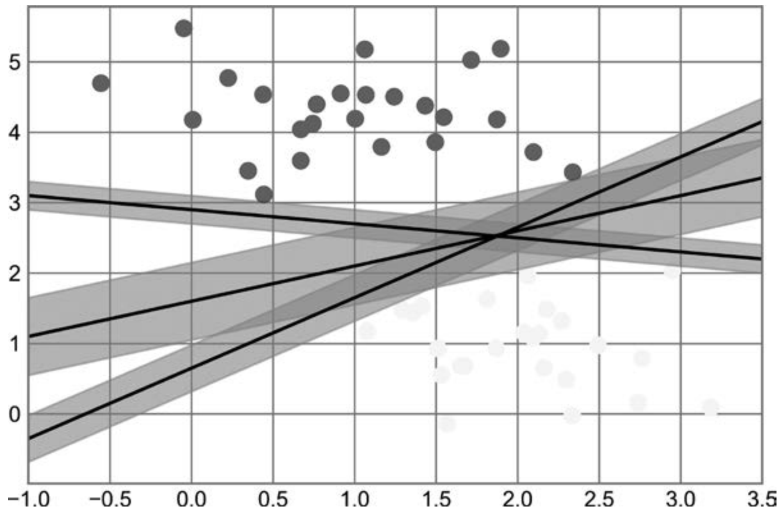


Рис. 43.3. Визуализация «отступов» в разделяющих классификаторах

Аппроксимация методом опорных векторов

Взглянем на реальную аппроксимацию этих данных: воспользуемся классификатором на основе метода опорных векторов (`SVC`) из библиотеки Scikit-Learn и обучим SVM-модель. Пока мы будем использовать линейное ядро и зададим очень большое значение параметра `C` (что это значит, вы узнаете далее):

```
In [5]: from sklearn.svm import SVC # "Support Vector Classifier"
        model = SVC(kernel='linear', C=1E10)
        model.fit(X, y)
Out[5]: SVC(C=10000000000.0, kernel='linear')
```

Для визуализации происходящего создадим простую и удобную функцию, строящую график границ решений, полученных методом SVM (рис. 43.4):

```
In [6]: def plot_svc_decision_function(model, ax=None, plot_support=True):
            """Строит график решающей функции для двумерного SVC"""
            if ax is None:
                ax = plt.gca()
            xlim = ax.get_xlim()
            ylim = ax.get_ylim()

            # Создаем координатную сетку для оценки модели
            x = np.linspace(xlim[0], xlim[1], 30)
            y = np.linspace(ylim[0], ylim[1], 30)
            Y, X = np.meshgrid(y, x)
```

```
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    # Рисуем границы принятия решений и отступы
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])

    # Рисуем опорные векторы
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=300, linewidth=1, edgecolors='black',
                   facecolors='none');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)

In [7]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(model);
```

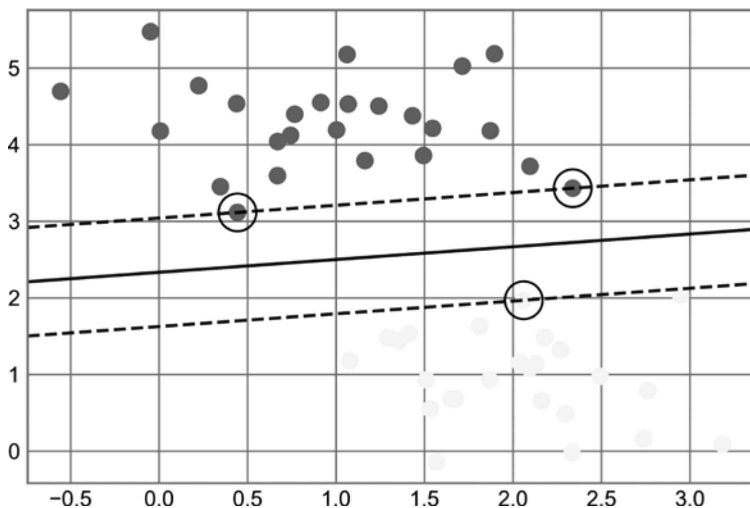


Рис. 43.4. Обучение классификатора на основе метода опорных векторов. На рисунке показаны границы отступов (пунктирные линии) и опорные векторы (окружности)

Эта разделяющая линия имеет максимальный отступ между двумя наборами точек. Обратите внимание, что некоторые из обучающих точек лишь касаются отступа. Они отмечены на рис. 43.5 окружностями. Эти точки — ключевые элементы аппроксимации, их называют *опорными векторами* (support vectors). В их честь алгоритм и получил свое название. В библиотеке Scikit-Learn данные об этих точках хранятся в атрибуте `support_vectors_` классификатора:

```
In [8]: model.support_vectors_
Out[8]: array([[0.44359863, 3.11530945],
               [2.33812285, 3.43116792],
               [2.06156753, 1.96918596]])
```

Ключ к успеху классификатора в том, что значение имеет только расположение опорных векторов. Все точки, находящиеся на правильной стороне, но дальше от отступа, не меняют аппроксимацию! Формально эти точки не вносят вклада в функцию потерь, используемую для обучения модели, поэтому их расположение и количество не имеют значения, если они не пересекают отступов.

Это можно увидеть, например, если построить график модели, обученной на первых 60 и первых 120 точках набора данных (рис. 43.5):

```
In [9]: def plot_svm(N=10, ax=None):
            X, y = make_blobs(n_samples=200, centers=2,
                              random_state=0, cluster_std=0.60)
            X = X[:N]
            y = y[:N]
            model = SVC(kernel='linear', C=1E10)
            model.fit(X, y)

            ax = ax or plt.gca()
            ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
            ax.set_xlim(-1, 4)
            ax.set_ylim(-1, 6)
            plot_svc_decision_function(model, ax)

        fig, ax = plt.subplots(1, 2, figsize=(16, 6))
        fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
        for axi, N in zip(ax, [60, 120]):
            plot_svm(N, axi)
            axi.set_title('N = {0}'.format(N))
```

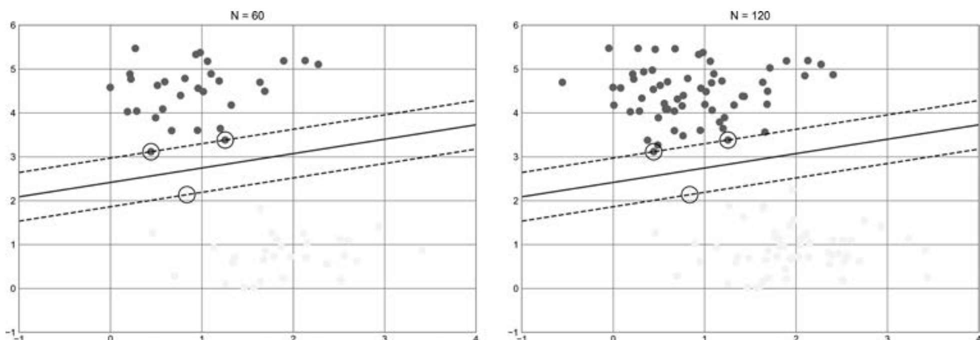


Рис. 43.5. Влияние новых обучающих точек на модель SVM

На графике слева показаны модель и опорные векторы для выборки с 60 обучающими точками. На графике справа количество обучающих точек вдвое больше, но модель не изменилась: три опорных вектора на графике слева остались опорными векторами и на графике справа. Подобная невосприимчивость к расположению отдаленных точек — одна из сильных сторон модели SVM.

Если вы работаете с этим блокнотом в интерактивном режиме, то воспользуйтесь интерактивными виджетами IPython для интерактивного просмотра этой особенности модели SVM:

```
In [10]: from ipywidgets import interact, fixed
         interact(plot_svm, N=(10, 200), ax=fixed(None));
Out[10]: interactive(children=(IntSlider(value=10, description='N', max=200, min=10),
          > Output()), _dom_classes=('widget-...
```

За границами линейности: SVM-ядро

Возможности метода SVM расширяются еще больше, если объединить его с *ядрами* (kernels). Мы уже сталкивались с ними ранее, в регрессии по комбинации базисных функций в главе 42. Там мы спроецировали данные в пространство большей размерности, определяемое полиномиальными и гауссовыми базисными функциями, и благодаря этому получили возможность аппроксимировать нелинейные зависимости с помощью линейного классификатора.

В SVM-моделях можно использовать один из вариантов той же идеи. Чтобы понять, зачем нужны ядра, рассмотрим следующие данные, которые нельзя разделить прямой линией (рис. 43.6):

```
In [11]: from sklearn.datasets import make_circles
         X, y = make_circles(100, factor=.1, noise=.1)

         clf = SVC(kernel='linear').fit(X, y)

         plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
         plot_svc_decision_function(clf, plot_support=False);
```

Очевидно, что эти данные *не разделимы* линейно. Но мы можем извлечь урок из регрессии по комбинации базисных функций, которую рассмотрели в главе 42, и попытаться спроецировать эти данные в пространство с большим числом измерений, где линейного разделителя *будет* достаточно. Например, одна из подходящих простых проекций — вычисление *радиальной базисной функции* (Radial Basis Function, RBF), центрированной посередине совокупности данных:

```
In [12]: r = np.exp(-(X ** 2).sum(1))
```

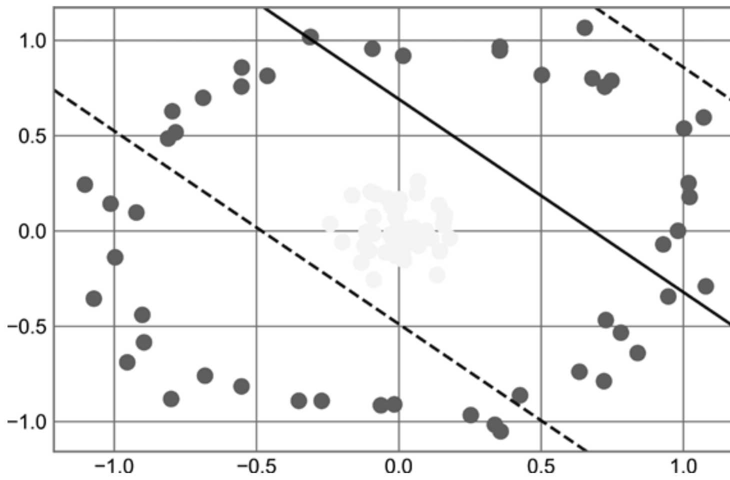


Рис. 43.6. В случае нелинейных границ линейный классификатор неэффективен

Визуализировать это дополнительное измерение можно с помощью трехмерного графика (рис. 43.7):

```
In [13]: from mpl_toolkits import mplot3d

         ax = plt.subplot(projection='3d')
         ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50, cmap='autumn')
         ax.view_init(elev=20, azim=30)
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('r');
```

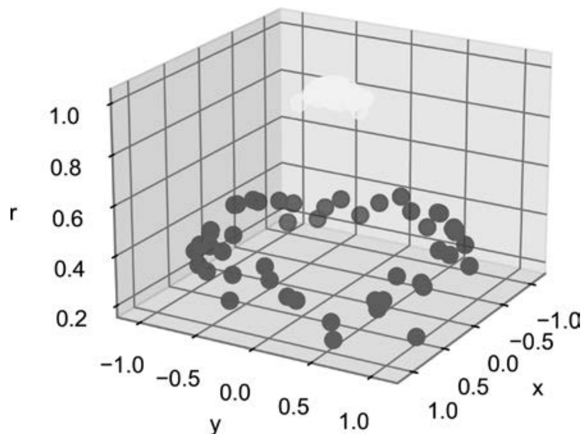


Рис. 43.7. Добавление третьего измерения дает возможность линейного разделения

Как видите, после добавления третьего измерения данные можно элементарно разделить линейно, проведя разделяющую плоскость на высоте, скажем $r = 0{,}7$.

Нам пришлось с особым вниманием подойти к выбору и настройке нашей проекции: если бы мы не центрировали радиальную базисную функцию должным образом, то не получили бы столь «чистых», линейно разделимых результатов. Необходимость подобного выбора — еще одна задача, требующая решения: хотелось бы иметь какой-то механизм, способный автоматически находить оптимальные базисные функции.

Одна из применяемых с этой целью стратегий состоит в вычислении базисных функций, центрированных по *каждой* из точек набора данных, с тем чтобы далее алгоритм SVM проанализировал полученные результаты. Эта разновидность преобразования базисных функций, известная под названием *преобразования ядра* (kernel transformation), основана на отношении подобия (или ядре) между каждой парой точек.

Потенциальная проблема этой методики — проецирование *N* точек в *N* измерений — состоит в том, что с ростом *N* может потребоваться колоссальный объем вычислений. Однако благодаря изящной процедуре, известной под названием *kernel trick* (https://oreil.ly/h7PBj), обучение на преобразованных с помощью ядра данных можно произвести неявно, то есть даже без построения полного *N*-мерного представления ядерной проекции! Этот kernel trick (ядерный трюк) является частью SVM и одной из причин мощи этого метода.

В библиотеке Scikit-Learn, чтобы применить алгоритм SVM с ядерным преобразованием, достаточно заменить линейное ядро на ядро RBF с помощью гиперпараметра `kernel` модели:

```
In [14]: clf = SVC(kernel='rbf', C=1E6)
         clf.fit(X, y)
Out[14]: SVC(C=1000000.0)
```

Воспользуемся функцией визуализации результатов обучения, которую мы определили ранее, и зададим опорные векторы (рис. 43.8):

```
In [15]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
         plot_svc_decision_function(clf)
         plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
                     s=300, lw=1, facecolors='none');
```

С помощью этого ядерного метода опорных векторов мы можем определить подходящую нелинейную границу решений. Такая методика ядерного преобразования часто используется в машинном обучении для превращения быстрых линейных методов в быстрые нелинейные, особенно для моделей, в которых можно использовать ядерный трюк (kernel trick).

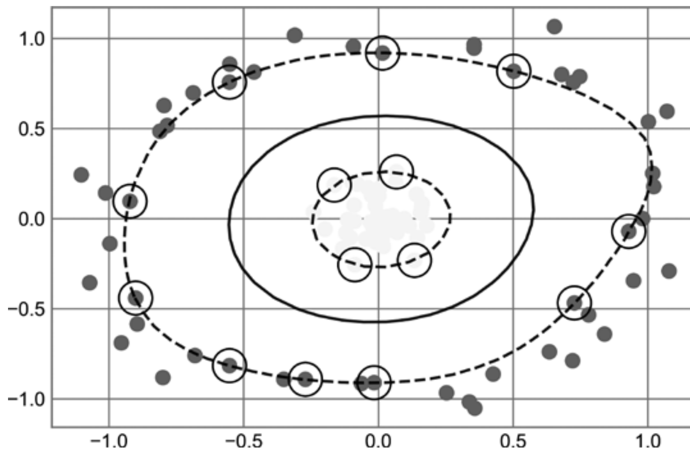


Рис. 43.8. Обучение ядерного SVM на наших данных

Настройка SVM: размытие отступов

До сих пор наше обсуждение касалось хорошо подобранных наборов данных, в которых существует идеальная граница решений. Но что, если данные в некоторой степени перекрываются? Например, допустим, мы имеем дело со следующими данными (рис. 43.9):

```
In [16]: X, y = make_blobs(n_samples=100, centers=2,
                           random_state=0, cluster_std=1.2)
         plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

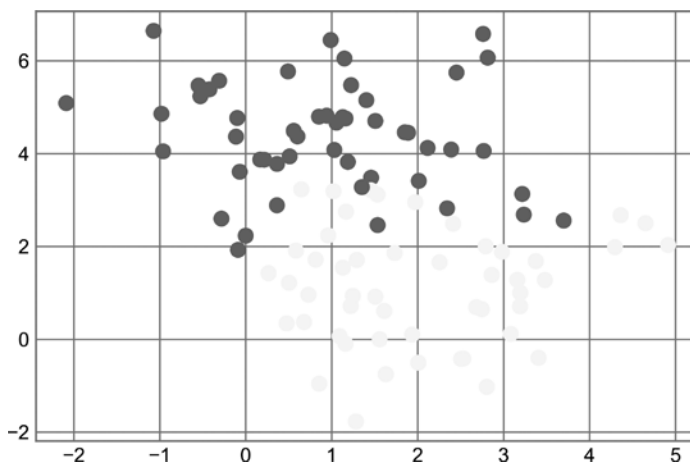


Рис. 43.9. Данные с некоторым перекрытием

На этот случай в реализации метода SVM есть небольшой поправочный параметр для «размытия» отступа. Данный параметр разрешает некоторым точкам «заходить» в область отступа, когда это приводит к лучшей аппроксимации. Степень размытости отступа управляется настроечным параметром, известным как `C`. При очень большом значении параметра `C` отступ задается «жестко» и точки не могут находиться на нем. При меньшем значении параметра `C` отступ становится более размытым и может включать некоторые точки.

На рис. 43.10 показано, как влияет изменение параметра `C` на итоговую аппроксимацию, размывая отступ:

```
In [17]: X, y = make_blobs(n_samples=100, centers=2,
                           random_state=0, cluster_std=0.8)

         fig, ax = plt.subplots(1, 2, figsize=(16, 6))
         fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

         for axi, C in zip(ax, [10.0, 0.1]):
             model = SVC(kernel='linear', C=C).fit(X, y)
             axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
             plot_svc_decision_function(model, axi)
             axi.scatter(model.support_vectors_[:, 0],
                         model.support_vectors_[:, 1],
                         s=300, lw=1, facecolors='none');
             axi.set_title('C = {0:.1f}'.format(C), size=14)
```

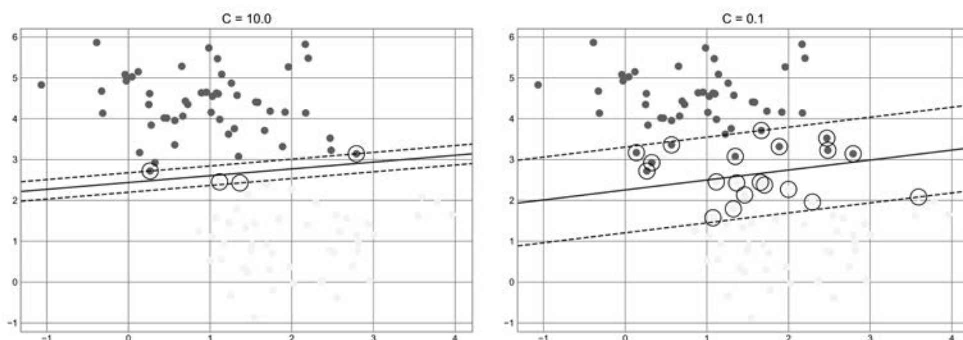


Рис. 43.10. Влияние параметра C на аппроксимацию методом опорных векторов

Оптимальное значение параметра `C` зависит от конкретного набора данных. Его следует настраивать с помощью перекрестной проверки или какой-либо аналогичной процедуры (как описывалось в главе 39).

Пример: распознавание лиц

В качестве примера работы метода опорных векторов рассмотрим задачу распознавания лиц. Воспользуемся набором данных Labeled Faces in the Wild (LFW), включающим несколько тысяч фотографий различных общественных деятелей. В библиотеку Scikit-Learn встроена утилита для загрузки этого набора данных:

```
In [18]: from sklearn.datasets import fetch_lfw_people
         faces = fetch_lfw_people(min_faces_per_person=60)
         print(faces.target_names)
         print(faces.images.shape)
Out[18]: ['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
          'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
         (1348, 62, 47)
```

Выведем несколько фотографий, чтобы увидеть, с чем мы будем иметь дело (рис. 43.11):

```
In [19]: fig, ax = plt.subplots(3, 5, figsize=(8, 6))
         for i, axi in enumerate(ax.flat):
             axi.imshow(faces.images[i], cmap='bone')
             axi.set(xticks=[], yticks=[],
                     xlabel=faces.target_names[faces.target[i]])
```

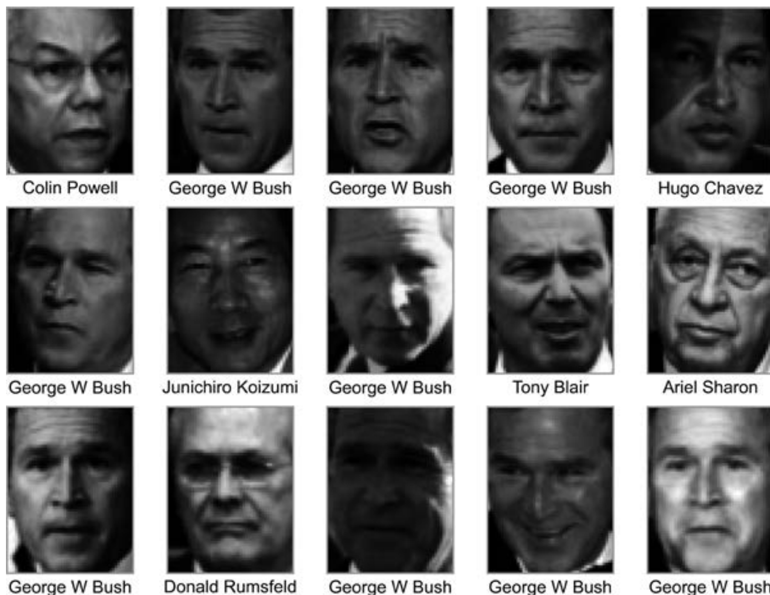


Рис. 43.11. Примеры фотографий из набора данных Labeled Faces in the Wild

Каждое изображение содержит 62 × 47, то есть примерно 3000 пикселов. Мы можем рассматривать каждый пиксел как признак, но эффективнее использовать какой-либо препроцессор для извлечения более осмысленных признаков. В данном случае мы воспользуемся методом главных компонент (см. главу 45) для извлечения 150 базовых компонент, которые передадим нашему классификатору на основе метода опорных векторов. Упростим эту задачу, объединив препроцессор и классификатор в единый конвейер:

```
In [20]: from sklearn.svm import SVC
         from sklearn.decomposition import PCA
         from sklearn.pipeline import make_pipeline

         pca = PCA(n_components=150, whiten=True,
                   svd_solver='randomized', random_state=42)
         svc = SVC(kernel='rbf', class_weight='balanced')
         model = make_pipeline(pca, svc)
```

Для проверки результатов работы нашего классификатора разобьем данные на обучающую и контрольную последовательности:

```
In [21]: from sklearn.model_selection import train_test_split
         Xtrain, Xtest, ytrain, ytest = train_test_split(faces.data, faces.target,
                                                         random_state=42)
```

Наконец, воспользуемся поиском по сетке с перекрестной проверкой для анализа сочетаний параметров. Подберем значения параметров `C` (управляющего размытием отступов) и `gamma` (управляющего размером ядра радиальной базисной функции) и определим оптимальную модель:

```
In [22]: from sklearn.model_selection import GridSearchCV
         param_grid = {'svc__C': [1, 5, 10, 50],
                       'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}
         grid = GridSearchCV(model, param_grid)

         %time grid.fit(Xtrain, ytrain)
         print(grid.best_params_)
Out[22]: CPU times: user 1min 19s, sys: 8.56 s, total: 1min 27s
         Wall time: 36.2 s
         {'svc__C': 10, 'svc__gamma': 0.001}
```

Оптимальные значения приходятся на середину сетки. Если бы они приходились на края сетки, то желательно было бы расширить сетку, чтобы гарантировать нахождение истинного оптимума.

Теперь с помощью этой модели, подвергнутой перекрестной проверке, можно предсказать метки для контрольных данных, которые модель еще не видела:

```
In [23]: model = grid.best_estimator_
         yfit = model.predict(Xtest)
```

Рассмотрим некоторые из контрольных изображений и предсказанных для них значений (рис. 43.12):

```
In [24]: fig, ax = plt.subplots(4, 6)
         for i, axi in enumerate(ax.flat):
             axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
             axi.set(xticks=[], yticks=[])
             axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                            color='black' if yfit[i] == ytest[i] else 'red')
         fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);
```

Рис. 43.12. Имена, спрогнозированные моделью

В этой небольшой выборке наша оптимальная модель ошиблась только для одного лица (лицо Дж. Буша в нижнем ряду было ошибочно отмечено как лицо Тони Блэра). Чтобы лучше прочувствовать эффективность работы модели, воспользуемся отчетом о классификации, в котором приведена статистика восстановления значений по каждой метке:

```
In [25]: from sklearn.metrics import classification_report
         print(classification_report(ytest, yfit,
                                     target_names=faces.target_names))
Out[25]:                    precision    recall  f1-score   support

              Ariel Sharon       0.65      0.73      0.69        15
              Colin Powell       0.80      0.87      0.83        68
           Donald Rumsfeld       0.74      0.84      0.79        31
             George W Bush       0.92      0.83      0.88       126
         Gerhard Schroeder       0.86      0.83      0.84        23
```

```
         Hugo Chavez       0.93      0.70      0.80        20
   Junichiro Koizumi       0.92      1.00      0.96        12
          Tony Blair       0.85      0.95      0.90        42

            accuracy                           0.85       337
           macro avg       0.83      0.84      0.84       337
        weighted avg       0.86      0.85      0.85       337
```

Можно также вывести на экран матрицу различий между этими классами (рис. 43.13):

```
In [26]: from sklearn.metrics import confusion_matrix
         import seaborn as sns
         mat = confusion_matrix(ytest, yfit)
         sns.heatmap(mat.T, square=True, annot=True, fmt='d',
                     cbar=False, cmap='Blues',
                     xticklabels=faces.target_names,
                     yticklabels=faces.target_names)
         plt.xlabel('true label')
         plt.ylabel('predicted label');
```

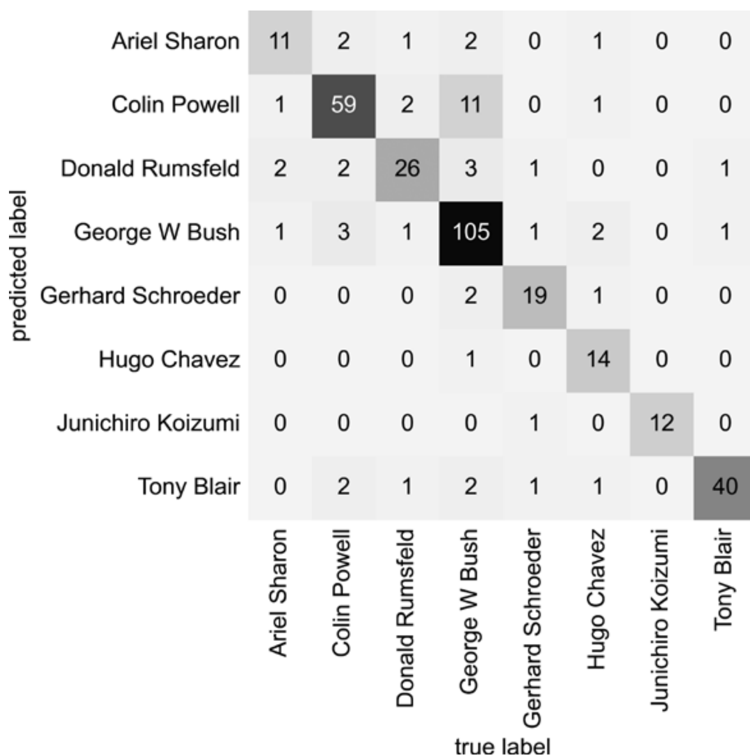


Рис. 43.13. Матрица различий для данных по лицам

Эти сведения помогают понять, какие имена может перепутать модель.

В реальных задачах распознавания лиц, когда фотографии не обрезаются заранее до одинаковых размеров, единственное отличие в схеме классификации лиц будет состоять в выборе признаков: вам потребуется использовать более сложный алгоритм для поиска лиц и извлечения признаков, не зависящий от пикселизации. Для таких случаев хорошим вариантом является использование OpenCV (http://opencv.org/) — библиотеки алгоритмов компьютерного зрения, обработки изображений и численных алгоритмов общего назначения, — которая, кроме всего прочего, включает предварительно обученные реализации современных инструментов извлечения признаков изображений в целом и лиц в частности.

Резюме

В этой главе мы привели краткое и понятное введение в основы методов опорных векторов. Эти методы являются мощными методами классификации по ряду причин.

- Зависимость от относительно небольшого количества опорных векторов означает компактность модели и небольшой объем используемой оперативной памяти.
- Этап предсказания после обучения модели занимает очень мало времени.
- Поскольку на работу этих методов влияют только точки, находящиеся возле отступа, они хорошо подходят для многомерных данных — даже данных с количеством измерений, превышающим количество образцов, — непростые условия работы для других алгоритмов.
- Интеграция с ядерными методами делает их универсальными, обеспечивает приспособляемость к множеству типов данных.

Однако у методов опорных векторов есть свои недостатки.

- Они масштабируются при количестве образцов N в наихудшем случае как $O[N^3]$ ($O[N^2]$ для более эффективных реализаций). При значительном количестве обучающих образцов вычислительные затраты могут оказаться непомерно высокими.
- Результаты зависят от удачности выбора параметра размытия `C`. Его необходимо тщательно выбирать с помощью перекрестной проверки, которая тоже может потребовать значительных вычислительных затрат при большом объеме данных.

- У результатов отсутствует непосредственная вероятностная интерпретация. Ее можно получить путем внутренней перекрестной проверки (см. параметр `probability` в классе `SVC`), но эта дополнительная оценка обходится недешево в смысле вычислительных затрат.

Учитывая эти особенности, я обращаюсь к SVM только тогда, когда более простые, быстрые и требующие меньшего количества настроек методы не удовлетворяют моим потребностям. Тем не менее если у вас достаточно мощный процессор, чтобы выполнять обучение и перекрестную проверку SVM на ваших данных, то этот метод может показать превосходные результаты.

ГЛАВА 44

Заглянем глубже: деревья решений и случайные леса

Мы подробно рассмотрели простой генеративный классификатор (наивный байесовский, см. главу 41) и более мощный разделяющий классификатор (метод опорных векторов, см. главу 43). В этой главе мы рассмотрим еще один мощный непараметрический алгоритм — *случайные леса* (random forests). Случайные леса — пример одного из *ансамблевых методов*, основанных на агрегировании результатов множества более простых моделей. Несколько неожиданный результат использования подобных ансамблевых методов — целое может оказаться больше суммы составных частей: точность прогнозирования большинством голосов среди нескольких моделей может оказаться выше точности любой отдельно взятой модели, участвующей в голосовании! Как такое происходит, вы увидите в следующих разделах.

Начнем с импортирования необходимых модулей:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
```

Движущая сила случайных лесов: деревья принятия решений

Случайные леса — пример обучаемого ансамбля на основе деревьев принятия решений. Поэтому начнем с обсуждения самих деревьев решений.

Деревья решений — интуитивно понятные способы классификации объектов: вы просто задаете серию уточняющих вопросов, ответы на которые помогут в классификации. Например, дерево принятия решений для классификации животных, встретившихся вам в походе, могло бы выглядеть, как показано на рис. 44.1.

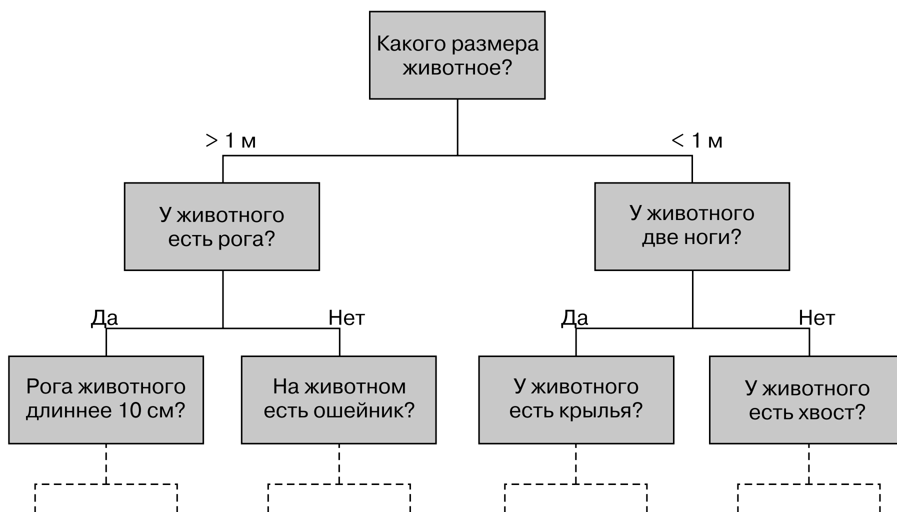


Рис. 44.1. Пример бинарного дерева принятия решений

Бинарное разбиение чрезвычайно эффективно: в хорошо спроектированном дереве каждый вопрос будет уменьшать количество вариантов приблизительно вдвое, очень быстро сужая круг возможных вариантов даже при большом количестве классов. Фокус состоит в том, какие вопросы задавать на каждом шаге. В реализациях деревьев принятия решений, связанных с машинным обучением, вопросы обычно имеют форму разделения данных по осям, то есть каждый узел дерева разбивает данные на две группы по одному из признаков.

Создание дерева принятия решений

Рассмотрим следующие двумерные данные с четырьмя возможными метками классов (рис. 44.2):

```
In [2]: from sklearn.datasets import make_blobs

        X, y = make_blobs(n_samples=300, centers=4,
                          random_state=0, cluster_std=1.0)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```

Простое дерево принятия решений для этих данных будет многократно делить данные по одной или нескольким осям в соответствии с определенным количественным критерием и на каждом уровне маркировать новую область согласно большинству лежащих в ней точек. На рис. 44.3 показаны первые четыре шага классификатора на этих данных, созданного на основе дерева принятия решений.

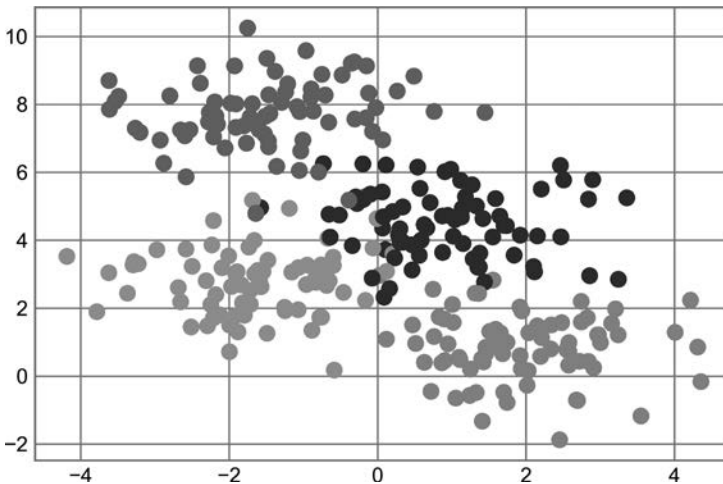


Рис. 44.2. Данные для классификатора на основе дерева принятия решений

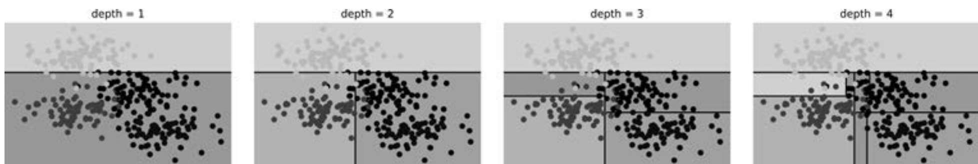


Рис. 44.3. Так дерево принятия решений делит данные[1]

Обратите внимание, что после первого разделения все точки в верхней ветви остаются неизменными, поэтому необходимости в дальнейшем ее разбиении нет. За исключением узлов, в которых присутствует только один цвет, на каждом из уровней *все* области снова делятся по одному из двух признаков.

Процесс обучения дерева принятия решений на наших данных можно выполнить в Scikit-Learn с помощью класса `DecisionTreeClassifier`:

```
In [3]: from sklearn.tree import DecisionTreeClassifier
        tree = DecisionTreeClassifier().fit(X, y)
```

Напишем небольшую вспомогательную функцию, чтобы упростить визуализацию результатов классификации:

```
In [4]: def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
            ax = ax or plt.gca()
```

[1] Код, генерирующий рисунки этой главы, можно найти в онлайн-приложении (https://oreil.ly/jv0wb).

```
    # Рисуем обучающие точки
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    ax.axis('off')
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # Обучаем модель
    model.fit(X, y)
    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                         np.linspace(*ylim, num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    # Создаем цветной график с результатами
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap=cmap, zorder=1)

    ax.set(xlim=xlim, ylim=ylim)
```

Теперь можно рассмотреть результат классификации с помощью дерева принятия решений (рис. 44.4):

```
In [5]: visualize_classifier(DecisionTreeClassifier(), X, y)
```

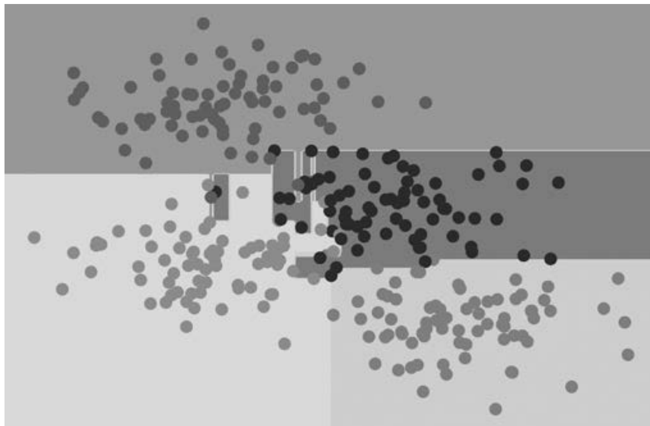


Рис. 44.4. Визуализация классификации с помощью дерева принятия решений

Если вы работаете с этим блокнотом в интерактивном режиме, то воспользуйтесь вспомогательным сценарием, включенным в онлайн-приложение (https://oreil.ly/etDrN), чтобы запустить интерактивную визуализацию процесса работы дерева принятия решений:

```
In [6]: # Модуль helpers_05_08 можно найти в онлайн-приложении
        import helpers_05_08
        helpers_05_08.plot_tree_interactive(X, y);
Out[6]: interactive(children=(Dropdown(description='depth', index=1, options=(1, 5),
         > value=5), Output()), _dom_classes...
```

Обратите внимание, что по мере продвижения процесс классификации создает области очень странной формы. Например, на шаге 5 между светлой и темной областями появляется узкая и вытянутая в высоту фиолетовая область. Очевидно, что это обусловлено не собственно распределением данных, а скорее особенностями их отбора или присутствием шума. То есть это дерево принятия решений уже на пятом шаге оказывается переобученным.

Деревья принятия решений и переобучение

Подобное переобучение присуще всем деревьям принятия решений: нет ничего проще, чем дойти до уровня в дереве, где начинают аппроксимироваться нюансы конкретных данных вместо общих характеристик распределений, из которых они получены. Другой способ увидеть это переобучение — обратиться к моделям, обученным на различных подмножествах набора данных, например, на рис. 44.5 показано обучение двух различных деревьев, каждое на половине исходного набора данных.

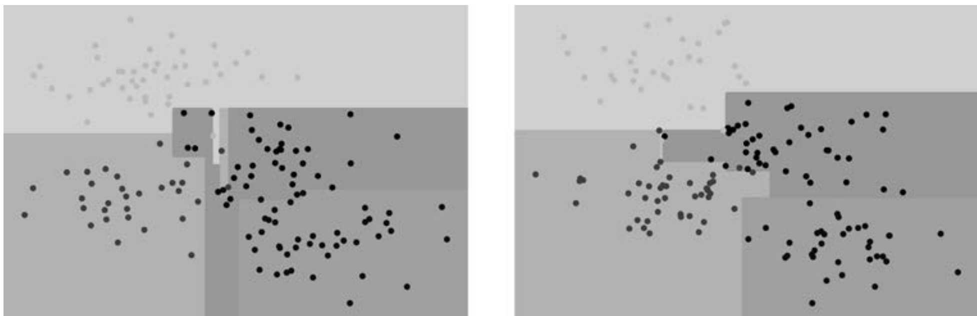


Рис. 44.5. Пример двух случайных деревьев принятия решений

Очевидно, что в некоторых местах результаты этих двух деревьев не противоречат друг другу (например, в углах), а в других местах их результаты классификации сильно различаются (например, в смежных областях любых двух кластеров). Важнейший вывод из этого рисунка: расхождения имеют тенденцию появляться там, где степень достоверности классификации ниже, а значит, мы можем добиться лучшего результата, используя информацию из обоих деревьев!

Если вы работаете с этим блокнотом в интерактивном режиме, то следующая функция позволит вам запустить интерактивную визуализацию процесса работы деревьев, обученных на случайных подмножествах набора данных:

```
In [7]: # Модуль helpers_05_08 можно найти в онлайн-приложении
        import helpers_05_08
        helpers_05_08.randomized_tree_interactive(X, y)
Out[7]: interactive(children=(Dropdown(description='random_state', options=(0, 100),
         > value=0), Output()), _dom_classes...
```

Подобно тому как использование информации из двух деревьев позволяет достичь лучшего результата, объединение информации из множества деревьев может дать наилучший результат.

Ансамбли моделей: случайные леса

Идея объединения нескольких переобученных моделей для уменьшения влияния эффекта переобучения лежит в основе ансамблевого метода, называемого *баггинг* (bagging). Баггинг использует ансамбль (набор) моделей, переобученных на исследуемых данных, и усредняет результаты для получения оптимальной классификации. Ансамбль случайных деревьев принятия решений называется *случайным лесом* (random forest).

Выполнить подобную баггинг-классификацию можно вручную с помощью метамодели `BaggingClassifier` из библиотеки Scikit-Learn (рис. 44.6):

```
In [8]: from sklearn.tree import DecisionTreeClassifier
        from sklearn.ensemble import BaggingClassifier

        tree = DecisionTreeClassifier()
        bag = BaggingClassifier(tree, n_estimators=100, max_samples=0.8,
                                random_state=1)

        bag.fit(X, y)
        visualize_classifier(bag, X, y)
```

В этом примере мы рандомизировали данные, обучив все модели на случайном подмножестве, включающем 80 % обучающих точек. На практике для более эффективной рандомизации деревьев принятия решений в процесс выбора разбиений добавляется некоторая стохастическая (случайная) составляющая. При этом всякий раз в обучении участвуют все данные, но результаты обучения все равно сохраняют требуемую случайность. Например, выбирая признак для разбиения, случайное дерево может выбирать из нескольких верхних признаков. Узнать больше о стратегиях рандомизации можно в документации библиотеки Scikit-Learn (https://oreil.ly/4jrv4) и упомянутых в ней справочных руководствах.

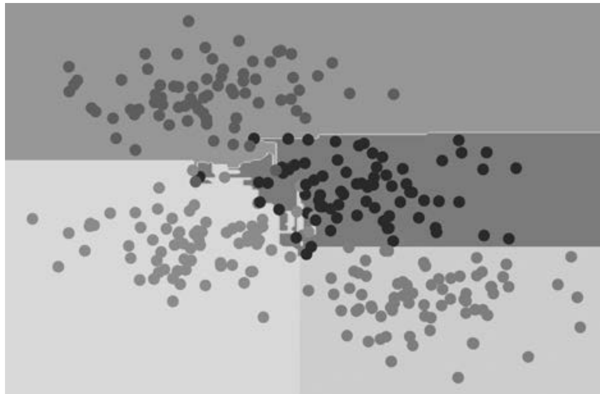


Рис. 44.6. Границы принятия решений для ансамбля случайных деревьев решений

В библиотеке Scikit-Learn подобный оптимизированный ансамбль случайных деревьев принятия решений, автоматически выполняющий всю рандомизацию, реализован в классе `RandomForestClassifier`. Вам нужно лишь выбрать количество деревьев, а класс очень быстро (при необходимости параллельно) обучит ансамбль деревьев (рис. 44.7):

```
In [9]: from sklearn.ensemble import RandomForestClassifier

        model = RandomForestClassifier(n_estimators=100, random_state=0)
        visualize_classifier(model, X, y);
```

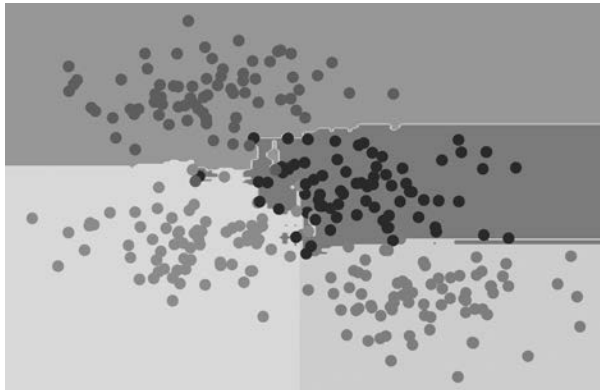


Рис. 44.7. Ансамбль случайных деревьев принятия решений

Как видите, путем усреднения более чем 100 случайно возмущенных моделей мы получаем общую модель, намного более близкую к нашим интуитивным представлениям о правильном разбиении параметрического пространства.

Регрессия с помощью случайных лесов

В предыдущем разделе мы рассмотрели случайные леса в контексте классификации. Случайные леса могут также оказаться полезными для регрессии (то есть непрерывных, а не категориальных величин). В этом случае используется класс `RandomForestRegressor`, синтаксис похож на тот, что мы видели выше.

Рассмотрим данные, полученные из сочетания быстрых и медленных колебаний (рис. 44.8):

```
In [10]: rng = np.random.RandomState(42)
         x = 10 * rng.rand(200)

         def model(x, sigma=0.3):
             fast_oscillation = np.sin(5 * x)
             slow_oscillation = np.sin(0.5 * x)
             noise = sigma * rng.randn(len(x))

             return slow_oscillation + fast_oscillation + noise

         y = model(x)
         plt.errorbar(x, y, 0.3, fmt='o');
```

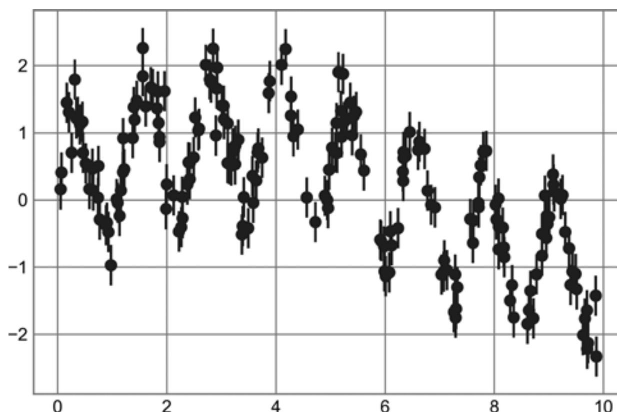


Рис. 44.8. Данные для регрессии с помощью случайного леса

Используя `RandomForestRegressor`, можно найти оптимальную аппроксимирующую кривую (рис. 44.9):

```
In [11]: from sklearn.ensemble import RandomForestRegressor
         forest = RandomForestRegressor(200)
         forest.fit(x[:, None], y)

         xfit = np.linspace(0, 10, 1000)
         yfit = forest.predict(xfit[:, None])
         ytrue = model(xfit, sigma=0)
```

```
plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)
plt.plot(xfit, yfit, '-r');
plt.plot(xfit, ytrue, '-k', alpha=0.5);
```

Рис. 44.9. Аппроксимация данных моделью на основе случайного леса

На этом рисунке истинная модель показана гладкой серой кривой, а модель на основе случайного леса — «рваной» красной. Как видите, непараметрическая модель на основе случайного леса достаточно гибкая для аппроксимации мультипериодических данных и не требует использовать мультипериодические модели!

Пример: использование случайного леса для классификации цифр

В главе 38 мы видели данные с изображениями рукописных цифр, включенные в состав библиотеки Scikit-Learn. Воспользуемся ими снова и посмотрим, как применить классификатор на основе случайных лесов для их распознавания.

```
In [12]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.keys()
Out[12]: dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names',
          > 'images', 'DESCR'])
```

Чтобы напомнить, с чем мы имеем дело, визуализируем несколько первых точек данных (рис. 44.10):

```
In [13]: # Настройки рисунка
         fig = plt.figure(figsize=(6, 6))  # figure size in inches
         fig.subplots_adjust(left=0, right=1, bottom=0, top=1,
                             hspace=0.05, wspace=0.05)
```

```
# Рисуем цифры: размер каждого изображения 8 x 8 пикселов
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

    # Маркируем изображение целевыми значениями
    ax.text(0, 7, str(digits.target[i]))
```

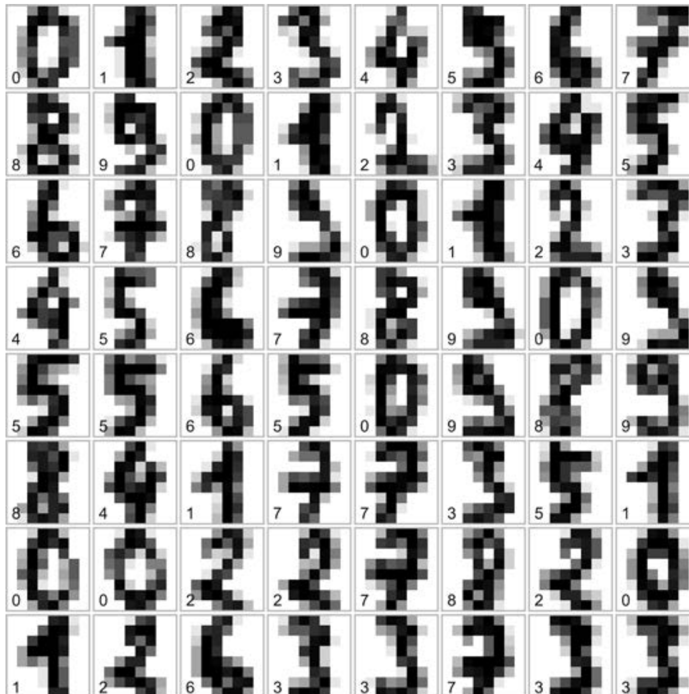


Рис. 44.10. Визуальное представление данных с изображениями рукописных цифр

Вот как можно классифицировать цифры с помощью случайного леса:

```
In [14]: from sklearn.model_selection import train_test_split

         Xtrain, Xtest, ytrain, ytest = train_test_split(digits.data, digits.target,
                                                         random_state=0)
         model = RandomForestClassifier(n_estimators=1000)
         model.fit(Xtrain, ytrain)
         ypred = model.predict(Xtest)
```

Взглянем на отчет о классификации для данного классификатора:

```
In [15]: from sklearn import metrics
         print(metrics.classification_report(ypred, ytest))
```

```
Out[15]:              precision    recall  f1-score   support

                  0       1.00      0.97      0.99        38
                  1       0.98      0.98      0.98        43
                  2       0.95      1.00      0.98        42
                  3       0.98      0.96      0.97        46
                  4       0.97      1.00      0.99        37
                  5       0.98      0.96      0.97        49
                  6       1.00      1.00      1.00        52
                  7       1.00      0.96      0.98        50
                  8       0.94      0.98      0.96        46
                  9       0.98      0.98      0.98        47

           accuracy                           0.98       450
          macro avg       0.98      0.98      0.98       450
       weighted avg       0.98      0.98      0.98       450
```

И на всякий случай нарисуем матрицу различий (рис. 44.11):

```
In [16]: from sklearn.metrics import confusion_matrix
         import seaborn as sns
         mat = confusion_matrix(ytest, ypred)
         sns.heatmap(mat.T, square=True, annot=True, fmt='d',
                     cbar=False, cmap='Blues')
         plt.xlabel('true label')
         plt.ylabel('predicted label');
```

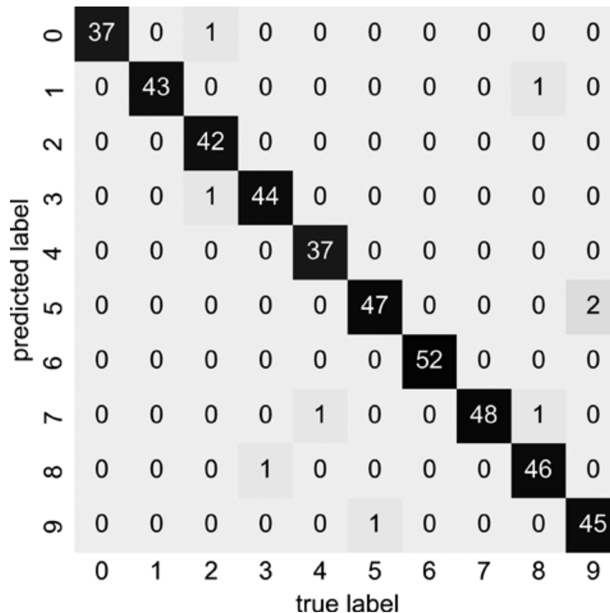


Рис. 44.11. Матрица различий для классификации цифр с помощью случайных лесов

Как оказывается, простой, не настроенный специальным образом случайный лес позволяет очень точно классифицировать рукописные цифры.

Резюме

В этой главе мы познакомились с понятием ансамблей моделей и, в частности, случайными лесами — ансамблями случайных деревьев принятия решений. Случайные леса — мощный метод, обладающий несколькими достоинствами.

- Как обучение, так и предсказание выполняются очень быстро в силу простоты моделей деревьев принятия решений, лежащих в основе. Кроме того, обе задачи допускают эффективное распараллеливание, так как отдельные деревья являются совершенно независимыми сущностями.
- Вариант с несколькими деревьями дает возможность использовать вероятностную классификацию: решение путем «голосования» моделей дает оценку вероятности (в библиотеке Scikit-Learn ее можно получить с помощью метода `predict_proba`).
- Непараметрическая модель исключительно гибкая и может эффективно работать с задачами, на которых другие модели оказываются недообученными.

Основной недостаток случайных лесов — их результаты сложно интерпретировать. Если требуется сделать осмысленные выводы относительно модели классификации, то случайные леса — не лучший вариант.

ГЛАВА 45

Заглянем глубже: метод главных компонент

До сих пор мы подробно изучали модели для машинного обучения с учителем, предсказывающие метки на основе маркированных обучающих данных. Теперь мы начнем знакомство с моделями обучения без учителя, позволяющими выявить интересные аспекты данных без всяких известных заранее меток.

В этой главе мы обсудим, возможно, один из наиболее широко используемых алгоритмов машинного обучения без учителя — метод главных компонент (principal component analysis, PCA). PCA — это алгоритм понижения размерности, но его также можно применять как инструмент визуализации, фильтрации шума, выделения и проектирования признаков и многого другого. После краткого концептуального обзора алгоритма PCA мы рассмотрим несколько примеров прикладных задач.

Начнем с импорта необходимых модулей:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
```

Знакомство с методом главных компонент

Метод главных компонент — быстрый и гибкий метод машинного обучения без учителя, предназначенный для понижения размерности данных. Мы познакомились с ним в главе 38. Его поведение проще всего визуализировать на примере двумерного набора данных. Рассмотрим следующие 200 точек (рис. 45.1):

```
In [2]: rng = np.random.RandomState(1)
        X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
        plt.scatter(X[:, 0], X[:, 1])
        plt.axis('equal');
```

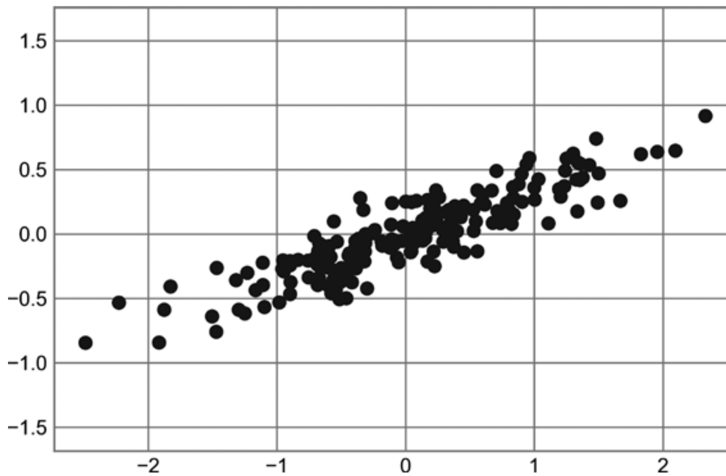


Рис. 45.1. Данные для демонстрации алгоритма PCA

Визуально очевидно, что зависимость между величинами x и y практически линейна. Это напоминает данные линейной регрессии, которые мы изучали в главе 42, но постановка задачи здесь несколько иная: задача машинного обучения без учителя состоит в выяснении *зависимости* между величинами x и y, а не в *предсказании* значений величины y по значениям величины x.

В методе главных компонент выполняется количественная оценка этой зависимости путем поиска списка *главных осей координат* (principal axes) данных и их использования для описания набора данных. Для этого можно использовать класс `PCA` из библиотеки Scikit-Learn:

```
In [3]: from sklearn.decomposition import PCA
        pca = PCA(n_components=2)
        pca.fit(X)
Out[3]: PCA(n_components=2)
```

При обучении алгоритм определяет некоторые относящиеся к данным величины, самые важные из них — компоненты и объясняемая ими дисперсия (explained variance):

```
In [4]: print(pca.components_)
Out[4]: [[-0.94446029 -0.32862557]
         [-0.32862557  0.94446029]]
In [5]: print(pca.explained_variance_)
Out[5]: [0.7625315 0.0184779]
```

Чтобы понять смысл этих чисел, визуализируем их поверх исходных данных в виде векторов, направление которых определяется значениями главных компонент, а длина — объясняемой ими дисперсией (рис. 45.2):

```
In [6]: def draw_vector(v0, v1, ax=None):
            ax = ax or plt.gca()
            arrowprops=dict(arrowstyle='->', linewidth=2,
                            shrinkA=0, shrinkB=0)
            ax.annotate('', v1, v0, arrowprops=arrowprops)

        # Визуализируем данные
        plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
        for length, vector in zip(pca.explained_variance_, pca.components_):
            v = vector * 3 * np.sqrt(length)
            draw_vector(pca.mean_, pca.mean_ + v)
        plt.axis('equal');
```

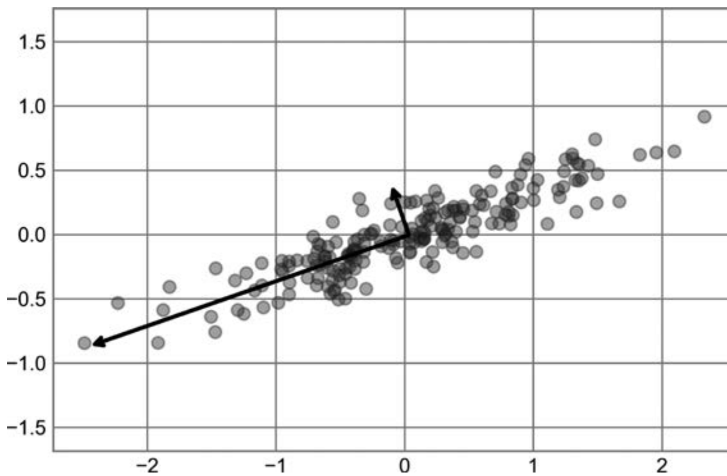


Рис. 45.2. Визуализация главных осей данных

Эти векторы представляют главные оси координат в пространстве данных, а их длины соответствуют «важности» осей для описания распределения данных, точнее, это мера дисперсии проекции данных на ось. Проекции точек данных на главные оси и есть главные компоненты.

Нарисовав главные компоненты поверх исходных данных, получаем графики, показанные на рис. 45.3.

Это преобразование, выполняющее переход от исходных осей координат к главным осям, называют *аффинным преобразованием* (affine transformation). В общем случае аффинное преобразование включает сдвиг (translation), поворот (rotation) и пропорциональное масштабирование (uniform scaling).

Хотя этот алгоритм поиска главных компонент может показаться всего лишь математической диковиной, оказывается, что у него есть весьма перспективные приложения в сфере машинного обучения и исследования данных.

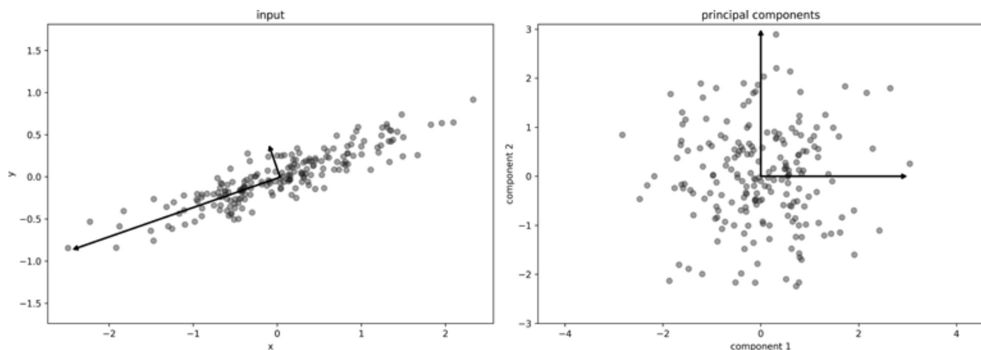


Рис. 45.3. Преобразованные главные оси данных[1]

PCA как метод понижения размерности

Использование метода PCA для понижения размерности включает обнуление одной или нескольких из минимальных главных компонент, в результате чего данные проецируются на пространство меньшей размерности с сохранением максимальной дисперсии данных.

Вот пример использования PCA в качестве понижающего размерность преобразования:

```
In [7]: pca = PCA(n_components=1)
        pca.fit(X)
        X_pca = pca.transform(X)
        print("original shape:   ", X.shape)
        print("transformed shape:", X_pca.shape)
Out[7]: original shape:    (200, 2)
        transformed shape: (200, 1)
```

В результате преобразования получились одномерные данные. Для лучшего понимания эффекта понижения размерности можно выполнить обратное преобразование данных и нарисовать их рядом с исходными (рис. 45.4):

```
In [8]: X_new = pca.inverse_transform(X_pca)
        plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
        plt.scatter(X_new[:, 0], X_new[:, 1], alpha=0.8)
        plt.axis('equal');
```

Светлые точки — это исходные данные, а темные — спроецированная версия. Этот рисунок наглядно показывает, как происходит понижение размерности с помощью

[1] Код, генерирующий рисунки этой главы, можно найти в онлайн-приложении (https://oreil.ly/jv0wb).

PCA: информация по наименее важным главным осям координат уничтожается, и остаются только компоненты данных с максимальной дисперсией. Отсекаемая часть дисперсии (пропорциональная разбросу точек рядом с линией, показанному на рис. 45.4) является приближенной мерой объема «информации», отбрасываемого при понижении размерности.

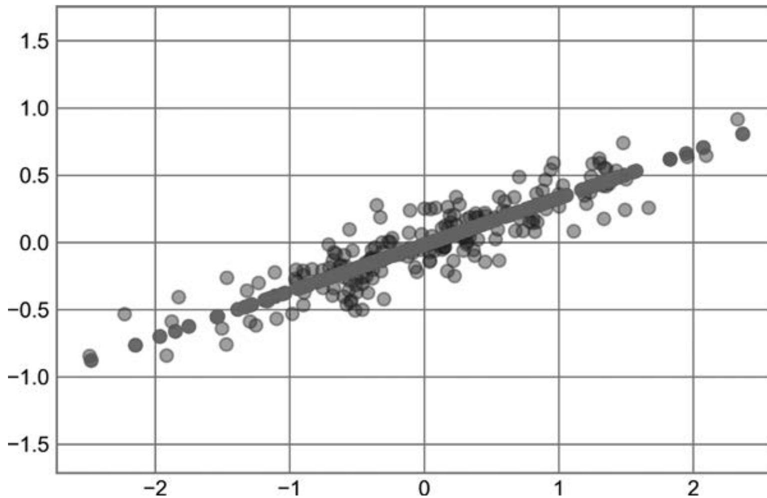


Рис. 45.4. Визуализация PCA как метода понижения размерности

Набор данных пониженной размерности в каком-то смысле «достаточно хорошо» представляет важнейшие зависимости между точками: несмотря на понижение размерности данных на 50 %, общая зависимость между точками данных по большей части сохранилась.

Использование метода PCA для визуализации: рукописные цифры

Полезность метода понижения размерности, возможно, не вполне ясна в случае двух измерений, но становится более очевидной при работе с многомерными данными. Рассмотрим приложение метода PCA к распознаванию рукописных цифр из набора данных, с которыми мы уже работали в главе 44.

Начнем с загрузки данных:

```
In [9]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.data.shape
Out[9]: (1797, 64)
```

Напомню, что данные состоят из изображений 8 × 8 пикселов, то есть 64-мерны. Чтобы понять зависимости между этими точками, воспользуемся методом PCA и спроецируем их в пространство меньшей размерности, например двумерное:

```
In [10]: pca = PCA(2)  # Проекция 64-мерного пространства в 2-мерное
         projected = pca.fit_transform(digits.data)
         print(digits.data.shape)
         print(projected.shape)
Out[10]: (1797, 64)
         (1797, 2)
```

Теперь построим график двух главных компонент каждой точки, чтобы получить больше информации о данных (рис. 45.5).

```
In [11]: plt.scatter(projected[:, 0], projected[:, 1],
                     c=digits.target, edgecolor='none', alpha=0.5,
                     cmap=plt.cm.get_cmap('rainbow', 10))
            plt.xlabel('component 1')
            plt.ylabel('component 2')
            plt.colorbar();
```

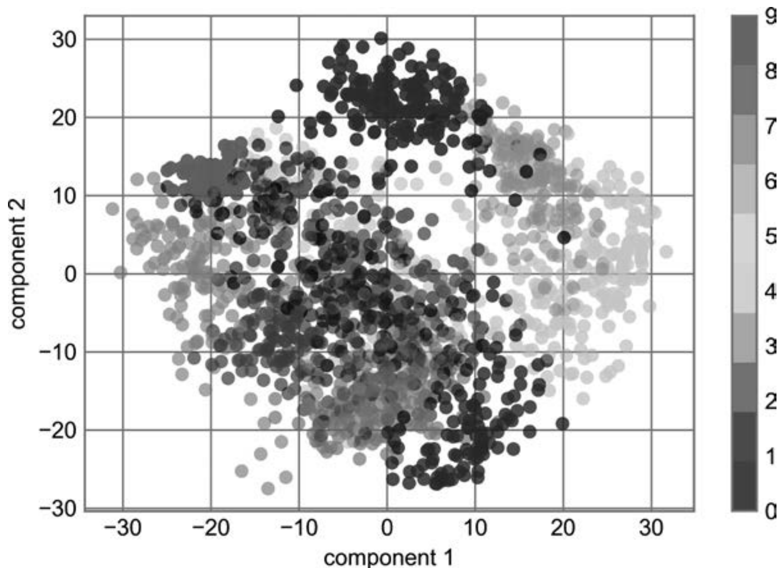


Рис. 45.5. Применение метода PCA к данным по рукописным цифрам

Вспомним, что означают эти компоненты: полный набор данных — это 64-мерное облако, а точки на рис. 45.5 — это проекции точек исходных данных на оси координат, соответствующие направлениям максимальной дисперсии. По существу, мы получили оптимальные растяжение и поворот в 64-мерном пространстве, по-

зволяющие увидеть, как цифры выглядят в двух измерениях, причем сделали это с помощью метода без учителя, то есть без использования истинных меток.

Что означают компоненты?

Заглянем еще глубже и спросим себя, что *означает* понижение размерности. Ответ на этот вопрос проще всего выразить в терминах сочетаний базисных векторов. Например, каждое изображение из обучающей последовательности описывается набором 64 значений пикселов, которые мы назовем вектором x:

$$x = [x_1, x_2, x_3 \ldots x_{64}].$$

Мы можем рассматривать это в терминах пиксельного базиса, то есть для формирования изображения необходимо умножить каждый элемент вышеприведенного вектора на значение описываемого им пиксела, после чего сложить результаты:

$$\text{image}(x) = x_1 \cdot (\text{пиксел } 1) + x_2 \cdot (\text{пиксел } 2) + x_3 \cdot (\text{пиксел } 3) \ldots x_{64} \cdot (\text{пиксел } 64).$$

Один из возможных способов понижения размерности этих данных — обнуление большей части базисных векторов. Например, если использовать только первые восемь пикселов, то получится восьмимерная проекция данных (рис. 45.6), но она будет плохо отражать изображения в целом, потому что отбрасывается почти 90 % пикселов!

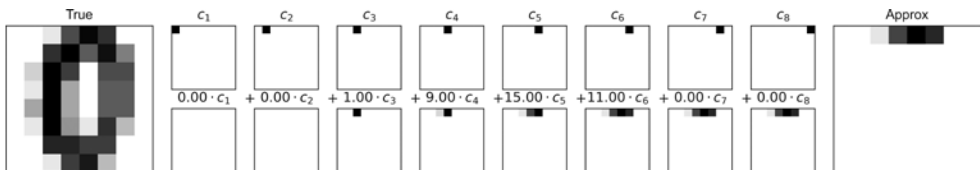


Рис. 45.6. Наивный метод понижения размерности путем отбрасывания пикселов

Верхний ряд показывает отдельные пикселы, а нижний — общий вклад этих пикселов в структуру изображения. С помощью только восьми компонент пиксельного базиса можно сконструировать лишь небольшую часть 64-пиксельного изображения. Продолжив эту последовательность действий и использовав все 64 пиксела, мы получили бы исходное изображение.

Однако попиксельное представление не единственный вариант базиса. Можно использовать и другие базисные функции, каждая из которых предполагает предопределенный вклад любого пиксела, и написать что-то вроде:

$$image(x) = \text{среднее} + x_1 \cdot (\text{базис } 1) + x_2 \cdot (\text{базис } 2) + x_3 \cdot (\text{базис } 3)\ldots$$

Метод PCA можно рассматривать как процесс выбора оптимальных базисных функций, чтобы комбинации лишь нескольких из них было достаточно для удовлетворительного воссоздания основной части элементов набора данных. Главные компоненты, служащие низкоразмерным представлением наших данных, будут в этом случае просто коэффициентами, умножаемыми на каждый из элементов ряда. На рис. 45.7 показано аналогичное восстановление цифры с помощью среднего значения и первых восьми базисных функций PCA.

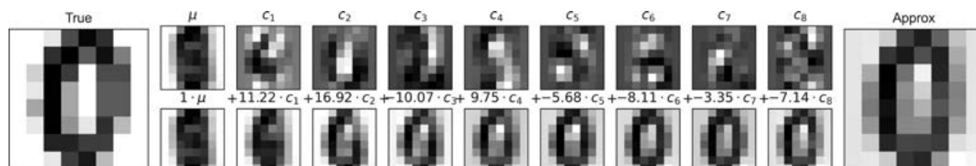


Рис. 45.7. Более продвинутый метод понижения размерности путем отбрасывания наименее важных главных компонент (ср. с рис. 45.6)

В отличие от пиксельного базис PCA позволяет восстановить наиболее заметные признаки входного изображения с помощью среднего значения и восьми компонент! Вклад каждого пиксела в каждый компонент в нашем двумерном примере зависит от направленности вектора. Именно в этом смысле PCA обеспечивает низкоразмерное представление данных: он находит более эффективный набор базисных функций, чем естественный пиксельный базис исходных данных.

Выбор количества компонент

Важнейшая составная часть применения метода PCA на практике — оценка количества компонент, необходимого для описания данных. Определить это количество можно с помощью представления интегральной *доли объясняемой дисперсии* (explained variance ratio) в виде функции от количества компонент (рис. 45.8):

```
In [12]: pca = PCA().fit(digits.data)
         plt.plot(np.cumsum(pca.explained_variance_ratio_))
         plt.xlabel('number of components')
         plt.ylabel('cumulative explained variance');
```

Эта кривая представляет количественную оценку содержания общей, 64-мерной, дисперсии в первых N компонентах. Например, на рис. 45.8 видно, что первые десять компонент набора данных с изображениями рукописных цифр описывают примерно 75 % дисперсии, а для описания доли дисперсии, близкой к 100 %, необходимо около 50 компонент.

В данном случае мы видим, что двумерная проекция теряет массу информации (согласно оценке объясняемой дисперсии) и что для сохранения 90 % дис-

персии необходимо около 20 компонент. Подобный график для многомерного набора данных помогает оценить уровень избыточности, присутствующей в признаках.

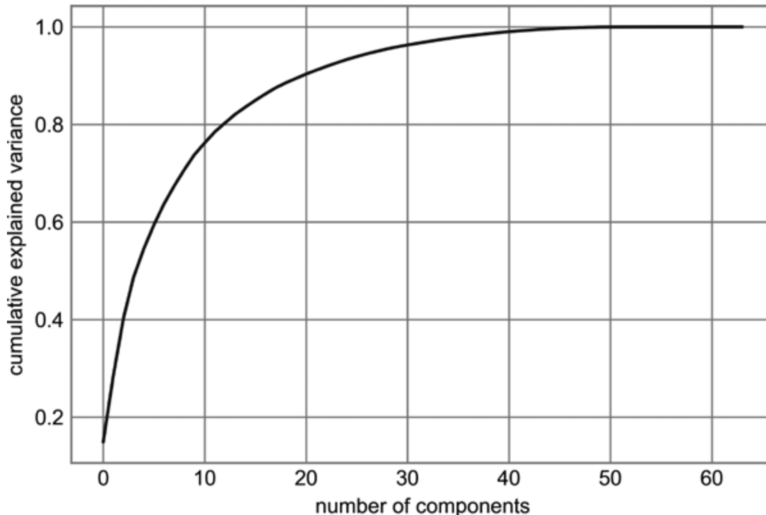


Рис. 45.8. Интегральная объяснимая дисперсия — мера сохранения методом PCA информационного наполнения данных

Использование метода PCA для фильтрации шума

Метод PCA можно применять для фильтрации зашумленных данных. Основная идея состоит в следующем: шум должен оказывать довольно слабое влияние на компоненты с дисперсией, значительно превышающей его уровень. Восстановление данных с помощью подмножества самых влиятельных главных компонент должно приводить к относительному сохранению сигнала и отбрасыванию шума.

Посмотрим, как это будет выглядеть в случае с изображениями цифр. Сначала нарисуем часть незашумленных исходных данных (рис. 45.9):

```
In [13]: def plot_digits(data):
             fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                                      subplot_kw={'xticks':[], 'yticks':[]},
                                      gridspec_kw=dict(hspace=0.1, wspace=0.1))
             for i, ax in enumerate(axes.flat):
                 ax.imshow(data[i].reshape(8, 8),
                           cmap='binary', interpolation='nearest',
                           clim=(0, 16))
         plot_digits(digits.data)
```

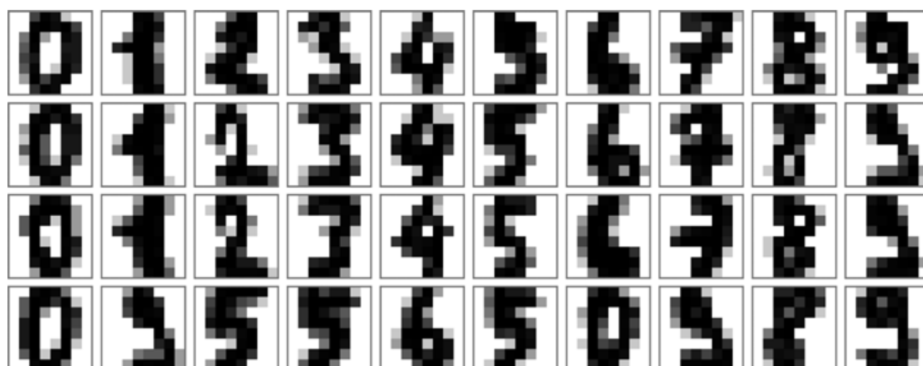


Рис. 45.9. Цифры без шума

Теперь добавим случайный шум, чтобы получить зашумленный набор данных, и нарисуем его (рис. 45.10):

```
In [14]: rng = np.random.default_rng(42)
         rng.normal(10, 2)
Out[14]: 10.609434159508863
In [15]: rng = np.random.default_rng(42)
         noisy = rng.normal(digits.data, 4)
         plot_digits(noisy)
```

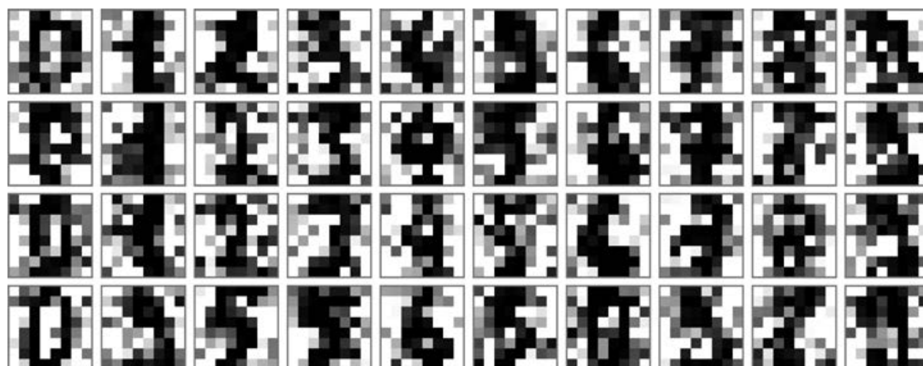


Рис. 45.10. Цифры с добавленным гауссовым случайным шумом

Визуально очевидно, что изображения зашумлены и содержат фиктивные пикселы. Обучим алгоритм PCA на этих зашумленных данных, указав, что проекция должна сохранять 50 % дисперсии:

```
In [16]: pca = PCA(0.50).fit(noisy)
         pca.n_components_
Out[16]: 12
```

В данном случае 50 % дисперсии соответствует 12 главным компонентам. Вычислим эти компоненты, после чего воспользуемся обратным преобразованием для восстановления отфильтрованных цифр (рис. 45.11):

```
In [17]: components = pca.transform(noisy)
         filtered = pca.inverse_transform(components)
         plot_digits(filtered)
```

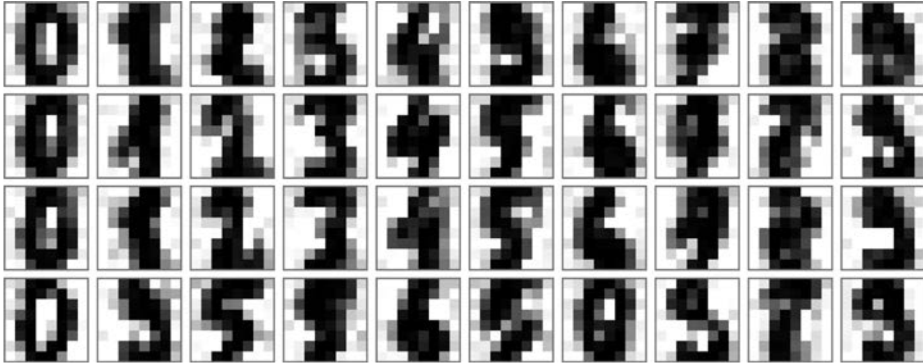


Рис. 45.11. Цифры после устранения шума с помощью метода PCA

Эти возможности по сохранению сигнала и фильтрации шума делают метод PCA очень удобной процедурой для выбора признаков, например, вместо обучения классификатора на чрезвычайно многомерных данных можно обучить его на низкоразмерном представлении, что автоматически приведет к фильтрации случайного шума во входных данных.

Пример: метод Eigenfaces

Ранее мы рассмотрели пример использования проекции PCA с целью выбора признаков для распознавания лиц с помощью метода опорных векторов (см. главу 43). Давайте вернемся к этому примеру и рассмотрим его подробнее. Напомню, что мы используем набор данных Labeled Faces in the Wild (LFW), доступный в библиотеке Scikit-Learn:

```
In [18]: from sklearn.datasets import fetch_lfw_people
         faces = fetch_lfw_people(min_faces_per_person=60)
         print(faces.target_names)
         print(faces.images.shape)
Out[18]: ['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
          'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
         (1348, 62, 47)
```

Выясним, какие главные оси координат охватывают этот набор данных. Поскольку набор данных велик, воспользуемся решателем `"random"` в классе модели `PCA`: он использует рандомизированный метод аппроксимации первых N главных компонент намного быстрее, чем стандартный подход, что очень удобно для многомерных данных (в данном случае размерность равна почти 3000). Рассмотрим первые 150 компонент:

```
In [19]: pca = PCA(150, svd_solver='randomized', random_state=42)
         pca.fit(faces.data)
Out[19]: PCA(n_components=150, random_state=42, svd_solver='randomized')
```

В нашем случае интересно визуализировать изображения, соответствующие первым нескольким главным компонентам (эти компоненты формально называют *собственными векторами* (eigenvectors), поэтому подобные изображения часто называют *«собственными лицами»* (eigenfaces); как показано на рис. 45.12, они такие же жуткие, как и их название):

```
In [20]: fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                                  subplot_kw={'xticks':[], 'yticks':[]},
                                  gridspec_kw=dict(hspace=0.1, wspace=0.1))
         for i, ax in enumerate(axes.flat):
             ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```

Рис. 45.12. Визуализация собственных лиц, полученных из набора данных LFW

Результат весьма интересен и позволяет увидеть разнообразие изображений: например, первые несколько собственных лиц (считая сверху слева), по всей вероятности, отражают угол падения света на лицо, а в дальнейшем главные векторы выделяют некоторые черты, например глаза, нос и губы. Посмотрим на интегральную дисперсию этих компонент, чтобы выяснить долю сохраненной информации (рис. 45.13).

```
In [21]: plt.plot(np.cumsum(pca.explained_variance_ratio_))
         plt.xlabel('number of components')
         plt.ylabel('cumulative explained variance');
```

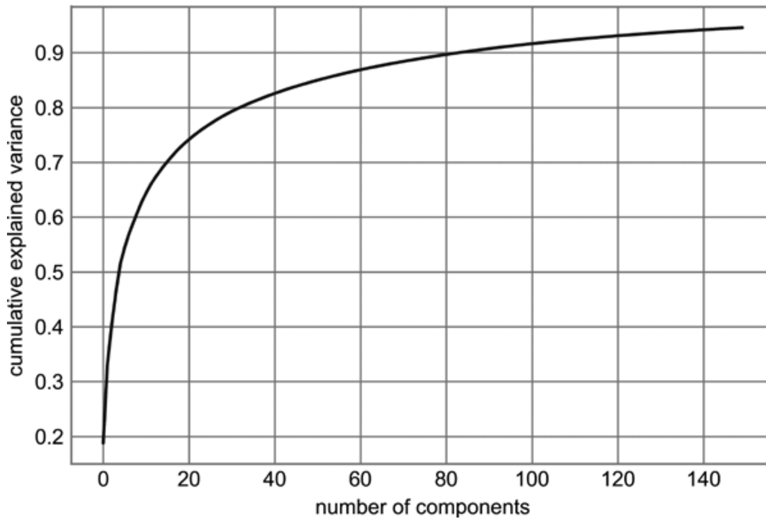


Рис. 45.13. Интегральная объяснимая дисперсия для набора данных LFW

Мы видим, что эти 150 компонент отвечают за более чем 90 % дисперсии. Это дает нам уверенность, что при использовании 150 компонент мы сможем восстановить большую часть существенных характеристик данных. Ради уточнения сравним входные изображения с восстановленными из этих 150 компонент (рис. 45.14):

```
In [22]: # Вычисляем компоненты и проекции лиц
         pca = pca.fit(faces.data)
         components = pca.transform(faces.data)
         projected = pca.inverse_transform(components)

In [23]: # Рисуем результаты
         fig, ax = plt.subplots(2, 10, figsize=(10, 2.5),
                                subplot_kw={'xticks':[], 'yticks':[]},
                                gridspec_kw=dict(hspace=0.1, wspace=0.1))
         for i in range(10):
             ax[0, i].imshow(faces.data[i].reshape(62, 47), cmap='binary_r')
             ax[1, i].imshow(projected[i].reshape(62, 47), cmap='binary_r')

         ax[0, 0].set_ylabel('full-dim\ninput')
         ax[1, 0].set_ylabel('150-dim\nreconstruction');
```

В верхнем ряду показаны исходные изображения, а в нижнем — восстановленные на основе лишь 150 из почти 3000 изначальных признаков. Теперь понятно, почему применявшийся в главе 43 выбор признаков с помощью PCA был настолько

успешен: хотя он понизил размерность данных почти в 20 раз, спроецированные изображения содержат всю необходимую информацию для визуального распознавания изображенных на фотографиях персон. А значит, наш алгоритм классификации достаточно обучить на 150-мерных, а не 3000-мерных данных, что в зависимости от конкретного алгоритма может оказаться намного более эффективным.

Рис. 45.14. 150-мерная реконструкция данных из LFW с помощью метода PCA

Резюме

В этой главе мы обсудили использование метода главных компонент для понижения размерности, визуализации многомерных данных, фильтрации шума и выбора признаков в многомерных данных. Метод PCA благодаря своей универсальности и простоте интерпретации результатов оказался эффективным в множестве контекстов и дисциплин. Я стараюсь при работе с любым многомерным набором данных начинать с использования метода PCA для визуализации зависимостей между точками (аналогично тому, как мы сделали с рукописными цифрами), выяснения дисперсии данных (аналогично тому, как мы поступили с собственными лицами) и внутренней размерности данных (путем построения графика доли объяснимой дисперсии). PCA не подходит для всех многомерных наборов данных, но с его помощью можно просто и эффективно почерпнуть полезную информацию о наборе многомерных данных.

Основной недостаток метода PCA состоит в том, что сильное влияние на него оказывают аномальные значения в данных. Поэтому было разработано немало вариантов PCA, устойчивых к ошибкам, многие из которых стремятся итеративно отбрасывать те точки данных, которые описываются исходными компонентами недостаточно хорошо. Библиотека Scikit-Learn содержит несколько реализаций интересных вариантов метода PCA в модуле `sklearn.decomposition`, включая класс `SparsePCA`, который вводит понятие регуляризации (см. главу 42), для обеспечения разреженности компонент.

В следующих главах мы рассмотрим другие методы машинного обучения без учителя, основывающиеся на некоторых идеях метода PCA.

ГЛАВА 46

Заглянем глубже: обучение на базе многообразий

В предыдущей главе мы познакомились с применением метода главных компонент для решения задачи понижения размерности — уменьшения количества признаков набора данных с сохранением существенных зависимостей между точками. Метод PCA — гибкий, быстрый и простой в интерпретации результатов, но при наличии *нелинейных* зависимостей в данных, примеры которых мы вскоре увидим, он работает не столь хорошо.

Чтобы справиться с этой проблемой, можно обратиться к *алгоритмам обучения на базе многообразий* (manifold learning) — классу моделей обучения без учителя, нацеленных на описание наборов данных как низкоразмерных многообразий, вложенных в пространство большей размерности. Чтобы понять, что такое многообразие, представьте себе лист бумаги: это двумерный объект в нашем привычном трехмерном мире, который можно изогнуть или свернуть в трех измерениях.

В терминах обучения на базе многообразий можно считать этот лист двумерным многообразием, вложенным в трехмерное пространство. Поворот, изгибание или растягивание листа бумаги в трехмерном пространстве не меняют его плоской геометрии: подобные операции эквивалентны линейному вложению. Если согнуть, скрутить или скомкать лист бумаги, он все равно остается двумерным многообразием, но вложение его в трехмерное пространство уже не будет линейным. Алгоритмы обучения на базе многообразий нацелены на изучение базовой двумерной природы листа бумаги, даже если он был смят ради размещения в трехмерном пространстве.

В этой главе мы рассмотрим несколько методов обучения на базе многообразий, причем наиболее глубоко изучим многомерное масштабирование (multidimensional

scaling, MDS), локально линейное вложение (locally linear embedding, LLE) и изометрическое отображение (isometric mapping, Isomap).

Начнем с импортирования необходимых модулей:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Обучение на базе многообразий: «HELLO»

Чтобы облегчить понимание вышеупомянутых понятий, для начала сгенерируем двумерные данные, подходящие для описания многообразия. Вот функция, создающая данные в форме слова «HELLO»:

```
In [2]: def make_hello(N=1000, rseed=42):
            # Создаем рисунок с текстом "HELLO"; сохраняем его в формате PNG
            fig, ax = plt.subplots(figsize=(4, 1))
            fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
            ax.axis('off')
            ax.text(0.5, 0.4, 'HELLO', va='center', ha='center',
                    weight='bold', size=85)
            fig.savefig('hello.png')
            plt.close(fig)

            # Открываем этот файл и извлекаем из него случайные точки
            from matplotlib.image import imread
            data = imread('hello.png')[::-1, :, 0].T
            rng = np.random.RandomState(rseed)
            X = rng.rand(4 * N, 2)
            i, j = (X * data.shape).astype(int).T
            mask = (data[i, j] < 1)
            X = X[mask]
            X[:, 0] *= (data.shape[0] / data.shape[1])
            X = X[:N]
            return X[np.argsort(X[:, 0])]
```

Вызываем эту функцию и отображаем полученные данные (рис. 46.1):

```
In [3]: X = make_hello(1000)
        colorize = dict(c=X[:, 0], cmap=plt.cm.get_cmap('rainbow', 5))
        plt.scatter(X[:, 0], X[:, 1], **colorize)
        plt.axis('equal');
```

Выходные данные двумерны и состоят из точек, формирующих слово «HELLO». Эта внешняя форма данных поможет нам визуально наблюдать за работой алгоритмов.

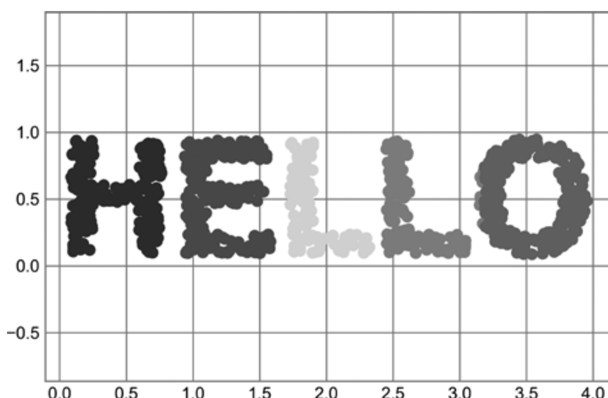


Рис. 46.1. Данные для обучения на базе многообразий

Многомерное масштабирование (MDS)

При взгляде на подобные данные становится ясно, что конкретные значения x и y — не самая существенная характеристика этого набора данных: мы можем пропорционально увеличить/сжать или повернуть данные, а надпись «HELLO» все равно останется четко различимой. Например, при использовании матрицы вращения для поворота данных значения x и y изменятся, но данные, по существу, останутся теми же (рис. 46.2):

```
In [4]: def rotate(X, angle):
            theta = np.deg2rad(angle)
            R = [[np.cos(theta), np.sin(theta)],
                 [-np.sin(theta), np.cos(theta)]]
            return np.dot(X, R)

        X2 = rotate(X, 20) + 5
        plt.scatter(X2[:, 0], X2[:, 1], **colorize)
        plt.axis('equal');
```

Это говорит о том, что значения x и y не всегда важны для внутренних зависимостей данных. Существенно в таком случае *расстояние* между каждой из точек и всеми остальными точками в наборе данных. Для представления его часто используют так называемую матрицу расстояний: для N точек создается такой массив $N \times N$, что его элемент (i, j) содержит расстояние между точками i и j. Воспользуемся эффективной функцией `pairwise_distances` из библиотеки Scikit-Learn, чтобы вычислить эти попарные расстояния для наших исходных данных:

```
In [5]: from sklearn.metrics import pairwise_distances
        D = pairwise_distances(X)
        D.shape
Out[5]: (1000, 1000)
```

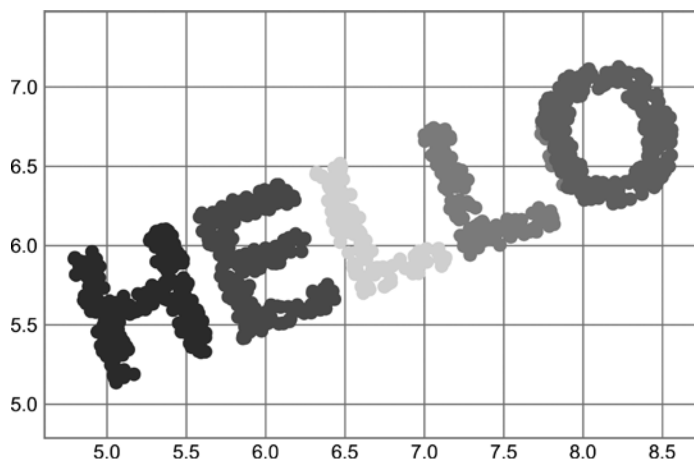


Рис. 46.2. Набор данных после поворота

Как я и обещал, для $N = 1000$ точек мы получили матрицу 1000×1000, которую можно визуализировать так, как показано на рис. 46.3:

```
In [6]: plt.imshow(D, zorder=2, cmap='viridis', interpolation='nearest')
        plt.colorbar();
```

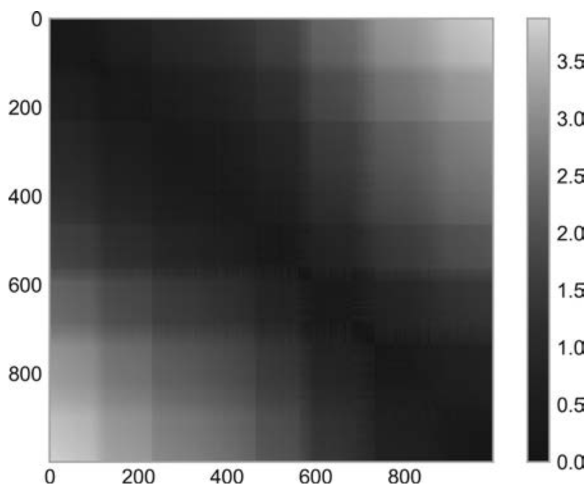


Рис. 46.3. Визуализация попарных расстояний между точками

Сформировав аналогичным образом матрицу расстояний между точками после поворота и сдвига, мы видим, что она не поменялась:

```
In [7]: D2 = pairwise_distances(X2)
        np.allclose(D, D2)
Out[7]: True
```

Благодаря подобной матрице расстояний мы получаем представление данных, инвариантное к поворотам и сдвигам, но визуализация матрицы интуитивно не слишком ясна. В представлении на рис. 46.3 потеряны всякие следы интересной структуры данных: виденного нами ранее слова «HELLO».

Хотя вычисление матрицы расстояний на основе координат (x, y) не представляет труда, обратное преобразование расстояний в координаты x и y — непростая задача. Именно для этого и служит алгоритм многомерного масштабирования: по заданной матрице расстояний между точками он восстанавливает D-мерное координатное представление данных. Посмотрим, как это будет выглядеть для нашей матрицы расстояний. Передадим значение `precomputed` в параметре `dissimilarity`, подсказав алгоритму, что мы передаем матрицу расстояний (рис. 46.4):

```
In [8]: from sklearn.manifold import MDS
        model = MDS(n_components=2, dissimilarity='precomputed', random_state=1701)
        out = model.fit_transform(D)
        plt.scatter(out[:, 0], out[:, 1], **colorize)
        plt.axis('equal');
```

Алгоритм MDS восстанавливает одно из возможных двумерных координатных представлений данных на основе одной лишь матрицы расстояний размера $N \times N$, описывающей зависимости между точками данных.

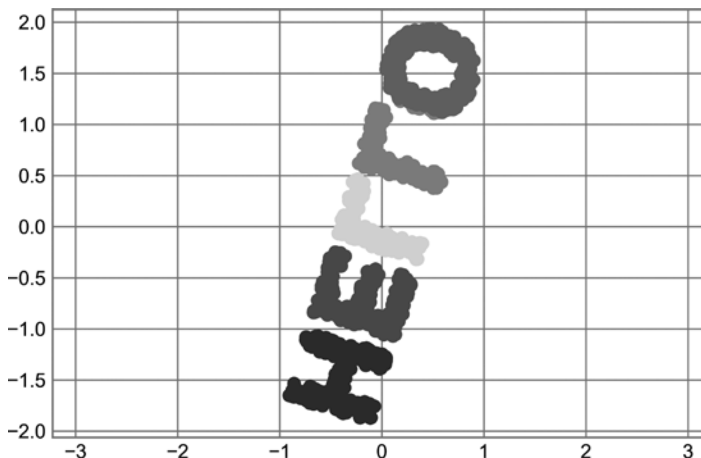


Рис. 46.4. MDS-вложение, вычисленное на основе попарных расстояний между точками

MDS как обучение на базе многообразий

Полезность этого метода становится еще очевиднее, если учесть, что матрицы расстояний можно вычислить для данных любой размерности. Так, например, вместо поворота данных в двумерном пространстве можно спроецировать их в трехмерное пространство с помощью следующей функции (по существу, это трехмерное обобщение матрицы вращения, с которой мы имели дело ранее):

```
In [9]: def random_projection(X, dimension=3, rseed=42):
            assert dimension >= X.shape[1]
            rng = np.random.RandomState(rseed)
            C = rng.randn(dimension, dimension)
            e, V = np.linalg.eigh(np.dot(C, C.T))
            return np.dot(X, V[:X.shape[1]])

        X3 = random_projection(X, 3)
        X3.shape
Out[9]: (1000, 3)
```

Построим график с этими точками, чтобы понять, с чем имеем дело (рис. 46.5):

```
In [10]: from mpl_toolkits import mplot3d
         ax = plt.axes(projection='3d')
         ax.scatter3D(X3[:, 0], X3[:, 1], X3[:, 2],
                      **colorize);
```

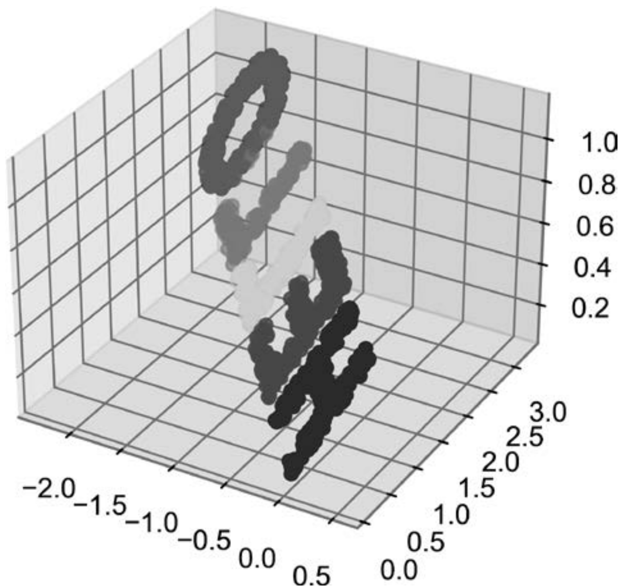


Рис. 46.5. Данные, линейно вложенные в трехмерное пространство

Эти данные можно передать классу MDS для вычисления матрицы расстояний и последующего определения оптимального двумерного вложения для нее. В результате мы получаем восстановленное представление исходных данных (рис. 46.6):

```
In [11]: model = MDS(n_components=2, random_state=1701)
         out3 = model.fit_transform(X3)
         plt.scatter(out3[:, 0], out3[:, 1], **colorize)
         plt.axis('equal');
```

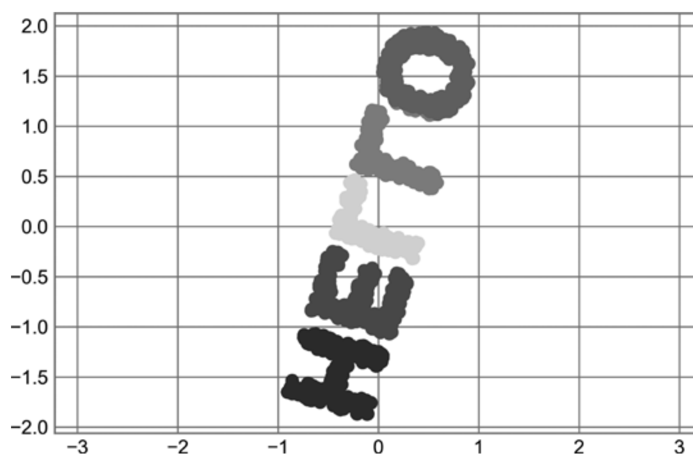


Рис. 46.6. MDS-вложение трехмерных данных позволяет восстановить исходные данные с точностью до поворота и отражения

В этом и состоит задача модели обучения на базе многообразий: при заданных многомерных вложенных данных она находит их представление в пространстве с меньшим числом измерений, не меняя определенные зависимости внутри данных. В частности, метод MDS сохраняет расстояния между всеми парами точек.

Нелинейные вложения: там, где MDS не работает

До сих пор мы говорили о *линейных* вложениях, состоящих из поворотов, сдвигов и масштабирования данных в пространствах более высокой размерности. Однако в случае нелинейного вложения, то есть при выходе за пределы этого простого набора операций, метод MDS терпит неудачу. Рассмотрим следующее вложение, при котором входные данные деформируются в форму трехмерной буквы «S»:

```
In [12]: def make_hello_s_curve(X):
             t = (X[:, 0] - 2) * 0.75 * np.pi
             x = np.sin(t)
             y = X[:, 1]
```

```
    z = np.sign(t) * (np.cos(t) - 1)
    return np.vstack((x, y, z)).T

XS = make_hello_s_curve(X)
```

Речь опять идет о трехмерных данных, но мы видим, что вложение получилось намного более сложным (рис. 46.7):

```
In [13]: from mpl_toolkits import mplot3d
         ax = plt.axes(projection='3d')
         ax.scatter3D(XS[:, 0], XS[:, 1], XS[:, 2],
                      **colorize);
```

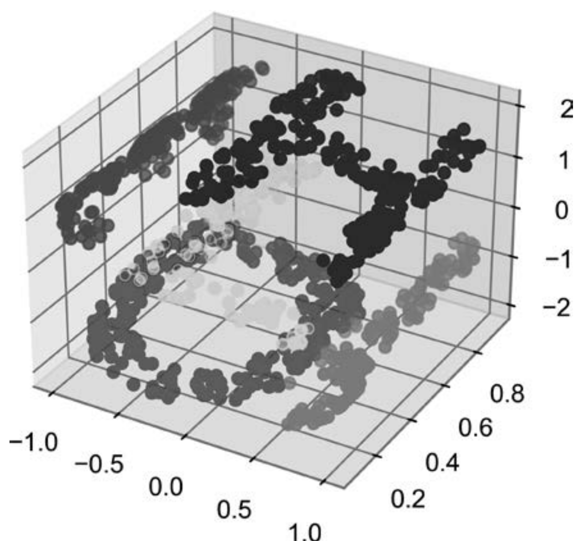


Рис. 46.7. Нелинейно вложенные в трехмерное пространство данные

Базовые взаимосвязи между точками данных сохранены, но на этот раз данные преобразованы нелинейным образом: они были свернуты в форму буквы «S».

Если попытаться применить к этим данным простой алгоритм MDS, он не сумеет «развернуть» это нелинейное вложение и мы потеряем из виду взаимосвязи во вложенном многообразии (рис. 46.8):

```
In [14]: from sklearn.manifold import MDS
         model = MDS(n_components=2, random_state=2)
         outS = model.fit_transform(XS)
         plt.scatter(outS[:, 0], outS[:, 1], **colorize)
         plt.axis('equal');
```

Даже самое лучшее двумерное линейное вложение не сможет развернуть обратно нашу S-образную кривую и просто отбросит исходную ось координат Y.

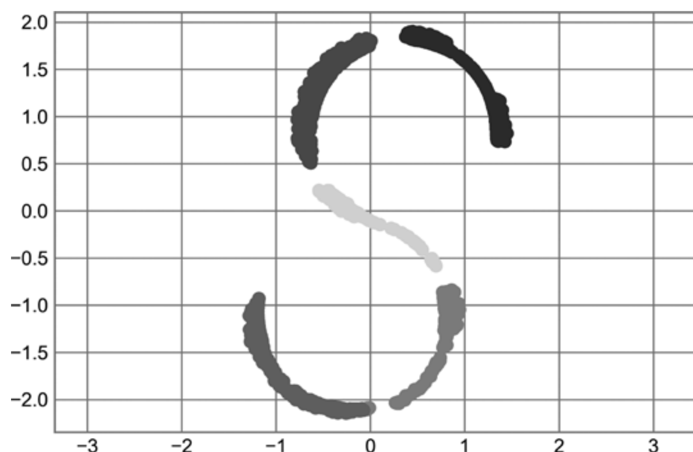


Рис. 46.8. Использование алгоритма MDS для нелинейных данных: попытка восстановления исходной структуры терпит неудачу

Нелинейные многообразия: локально линейное вложение

В чем проблема? Можно ли ее как-то решить? Корень проблемы в том, что MDS пытается сохранять расстояния между удаленными точками при формировании вложения. Но что, если изменить алгоритм так, чтобы он сохранял расстояния только между близлежащими точками? Полученное в результате вложение должно оказаться ближе к желаемому нами.

Наглядно можно представить этот метод так, как показано на рис. 46.9.

Тонкие линии отражают расстояния, которые необходимо сохранить при вложении. Слева представлена модель, используемая в методе MDS: она сохраняет расстояния между всеми парами точек в наборе данных. Справа — модель, используемая алгоритмом обучения на базе многообразий, который называется *локально линейным вложением* (locally linear embedding, LLE). Вместо *всех* расстояний сохраняются только расстояния между соседними точками: в данном случае ближайшими 100 соседями каждой точки.

При изучении рисунка слева становится понятно, почему метод MDS не работает: нет никакого способа «упростить» эти данные, сохранив в достаточной степени длину каждого отрезка между двумя точками. На рисунке справа ситуация выглядит несколько более оптимистично. Вполне можно представить себе такое развертывание данных, при котором длины отрезков сохранятся хотя бы приблизительно. Именно это и делает метод LLE, отыскивая глобальный экстремум отражающей эту логику функции стоимости.

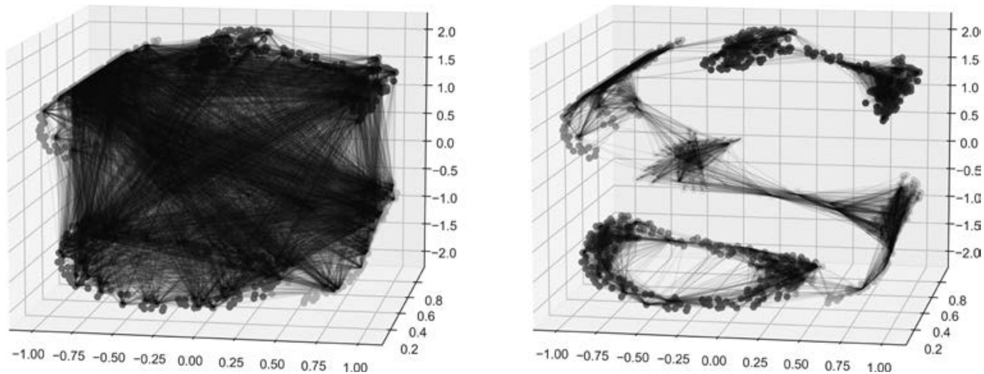


Рис. 46.9. Представление связей между точками в методах MDS и LLE

Существует несколько вариантов метода LLE, здесь для восстановления вложенного двумерного многообразия мы используем *модифицированный алгоритм LLE*. В целом он работает лучше других вариантов при восстановлении хорошо структурированных многообразий с очень небольшой дисторсией (рис. 46.10):

```
In [15]: from sklearn.manifold import LocallyLinearEmbedding
         model = LocallyLinearEmbedding(
             n_neighbors=100, n_components=2,
             method='modified', eigen_solver='dense')
         out = model.fit_transform(XS)

         fig, ax = plt.subplots()
         ax.scatter(out[:, 0], out[:, 1], **colorize)
         ax.set_ylim(0.15, -0.15);
```

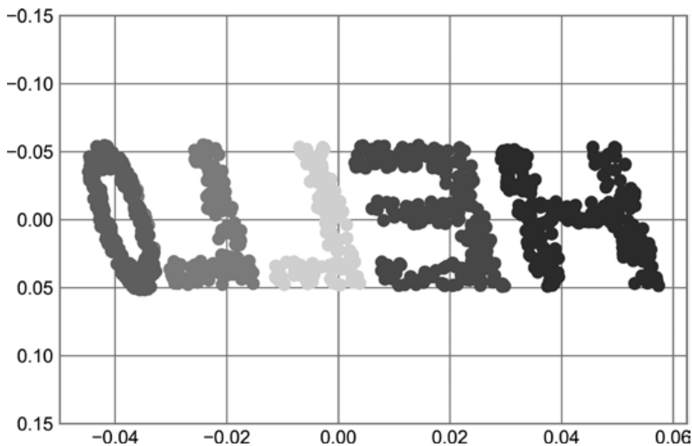


Рис. 46.10. Локально линейное вложение может восстанавливать изначальные данные из нелинейно вложенных входных данных

Результат остается несколько искаженным по сравнению с исходным многообразием, но существенные зависимости внутри данных метод сохранил!

Некоторые соображения относительно методов обучения на базе многообразий

Хотя это и была захватывающая история, на практике методы обучения на базе многообразий оказываются настолько привередливыми, что редко используются для чего-то большего, чем простая качественная визуализация многомерных данных.

Вот несколько конкретных примеров, когда обучение на базе многообразий проигрывает по сравнению с методом PCA.

- При обучении на базе многообразий не существует удачного решения для обработки отсутствующих данных. В отличие от него в методе PCA существуют простые итеративные подходы для работы с отсутствующими данными.
- При обучении на базе многообразий наличие шума в данных может «закоротить» многообразие и коренным образом изменить вложение. В отличие от него метод PCA естественным образом отделяет шум от наиболее важных компонент.
- Результат вложения многообразия обычно сильно зависит от количества выбранных соседей, и не существует надежного количественного способа выбора оптимального числа соседей. В отличие от него метод PCA не требует подобного выбора.
- При обучении на базе многообразий непросто определить оптимальное число измерений на выходе алгоритма. В отличие от него метод PCA позволяет определить выходную размерность, основываясь на доле объясняемой дисперсии.
- При обучении на базе многообразий смысл вложенных измерений не всегда понятен. В методе PCA смысл главных компонент совершенно ясен.
- При обучении на базе многообразий вычислительная сложность методов составляет $O[N^2]$ или даже $O[N^3]$. Некоторые рандомизированные варианты метода PCA работают гораздо быстрее (однако в пакете *megaman* (https://oreil.ly/VLBly) реализованы методы обучения на базе многообразий, масштабирующиеся гораздо лучше).

С учетом всего вышесказанного единственное безусловное преимущество методов обучения на базе многообразий перед PCA состоит в их способности сохранять нелинейные зависимости в данных. Именно поэтому я стараюсь сначала изучать данные с помощью PCA, а затем применять методы обучения на базе многообразий.

В библиотеке Scikit-Learn реализовано несколько распространенных вариантов обучения на базе многообразий и локально линейного вложения: в документации

Scikit-Learn имеются их обсуждение и сравнение (https://oreil.ly/tFzS5). Исходя из собственного опыта, могу дать вам следующие рекомендации.

- В небольших задачах, подобных задаче с S-образной кривой, локально линейное вложение (LLE) и его варианты (особенно модифицированный метод LLE) демонстрируют отличные результаты. Они реализованы в классе `sklearn.manifold.LocallyLinearEmbedding`.
- В случае многомерных данных, полученных из реальных источников, метод LLE часто работает плохо, и в отличие от него изометрическое отображение (Isomap) часто выдает более осмысленные вложения. Оно реализовано в классе `sklearn.manifold.Isomap`.
- Для сильно кластеризованных данных отличные результаты демонстрирует метод *стохастического вложения соседей на основе распределения Стьюдента* (t-distributed stochastic neighbor embedding), хотя и работает иногда очень медленно по сравнению с другими методами. Он реализован в классе `sklearn.manifold.TSNE`.

Если у вас появится желание узнать, как они работают, запустите каждый из них на данных из этого раздела.

Пример: использование Isomap для распознавания лиц

Обучение на базе многообразий часто применяется при исследовании зависимостей между многомерными точками данных. Один из распространенных случаев многомерных данных — изображения. Например, набор изображений, состоящих каждое из 1000 пикселов, можно рассматривать как набор точек в 1000-мерном пространстве — яркость каждого пиксела в каждом изображении соотносится с кординатой в соответствующем измерении.

Для иллюстрации применим алгоритм Isomap к данным из набора Labeled Faces in the Wild (LFW), с которым мы уже сталкивались в главах 43 и 45. Следующая команда скачает данные и сохранит их в вашем домашнем каталоге для дальнейшего использования:

```
In [16]: from sklearn.datasets import fetch_lfw_people
         faces = fetch_lfw_people(min_faces_per_person=30)
         faces.data.shape
Out[16]: (2370, 2914)
```

Итак, у нас имеется 2370 изображений, каждое размером 2914 пикселов. Другими словами, изображения можно считать точками данных в 2914-мерном пространстве!

Быстро визуализируем несколько изображений, чтобы посмотреть, с чем мы имеем дело (рис. 46.11):

```
In [17]: fig, ax = plt.subplots(4, 8, subplot_kw=dict(xticks=[], yticks=[]))
         for i, axi in enumerate(ax.flat):
             axi.imshow(faces.images[i], cmap='gray')
```

Рис. 46.11. Примеры исходных фотографий лиц

В главе 45, где мы работали с этими данными, нашей главной целью было сжатие и последующее использование главных компонент для восстановления исходных данных из представления с меньшим числом измерений.

Метод PCA достаточно универсален, чтобы использовать его в контексте построения низкоразмерного вложения 2914-мерных данных и изучить фундаментальные отношения между изображениями. Давайте снова посмотрим на долю объясняемой дисперсии. Это даст представление о том, сколько линейных признаков требуется для описания данных (рис. 46.12).

```
In [18]: from sklearn.decomposition import PCA
         model = PCA(100, svd_solver='randomized').fit(faces.data)
         plt.plot(np.cumsum(model.explained_variance_ratio_))
         plt.xlabel('n components')
         plt.ylabel('cumulative variance');
```

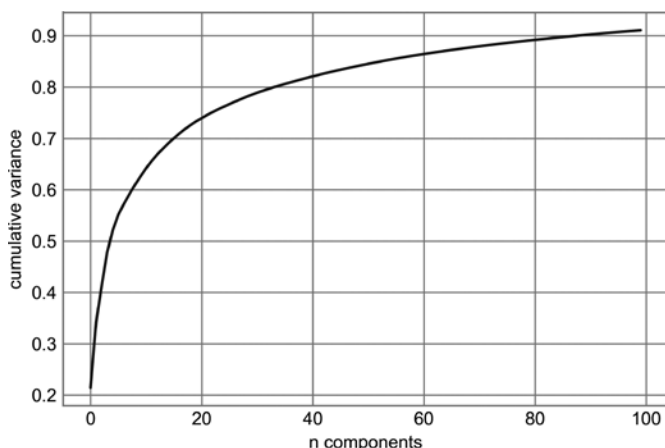


Рис. 46.12. Интегральная дисперсия, полученная из проекции методом PCA

Как видите, для сохранения 90 % дисперсии необходимо почти 100 компонент. Это значит, что данные, по сути, имеют чрезвычайно высокую размерность и их невозможно описать линейно с помощью всего нескольких компонент.

В подобных случаях могут оказаться полезны нелинейные вложения на базе многообразий, такие как LLE и Isomap. Рассчитать вложение Isomap для этих фотографий лиц можно аналогичным образом:

```
In [19]: from sklearn.manifold import Isomap
         model = Isomap(n_components=2)
         proj = model.fit_transform(faces.data)
         proj.shape
Out[19]: (2370, 2)
```

В результате получается двумерная проекция всех исходных изображений. Чтобы лучше представить, что говорит нам эта проекция, опишем функцию, выводящую миниатюры спроецированных изображений:

```
In [20]: from matplotlib import offsetbox

         def plot_components(data, model, images=None, ax=None,
                             thumb_frac=0.05, cmap='gray'):
             ax = ax or plt.gca()

             proj = model.fit_transform(data)
             ax.plot(proj[:, 0], proj[:, 1], '.k')

             if images is not None:
                 min_dist_2 = (thumb_frac * max(proj.max(0) - proj.min(0))) ** 2
                 shown_images = np.array([2 * proj.max(0)])
                 for i in range(data.shape[0]):
                     dist = np.sum((proj[i] - shown_images) ** 2, 1)
```

```
            if np.min(dist) < min_dist_2:
                # Не отображаем слишком близко расположенные точки
                continue
            shown_images = np.vstack([shown_images, proj[i]])
            imagebox = offsetbox.AnnotationBbox(
                offsetbox.OffsetImage(images[i], cmap=cmap),
                                      proj[i])
            ax.add_artist(imagebox)
```

Теперь вызовем эту функцию и рассмотрим результат (рис. 46.13):

```
In [21]: fig, ax = plt.subplots(figsize=(10, 10))
         plot_components(faces.data,
                         model=Isomap(n_components=2),
                         images=faces.images[:, ::2, ::2])
```

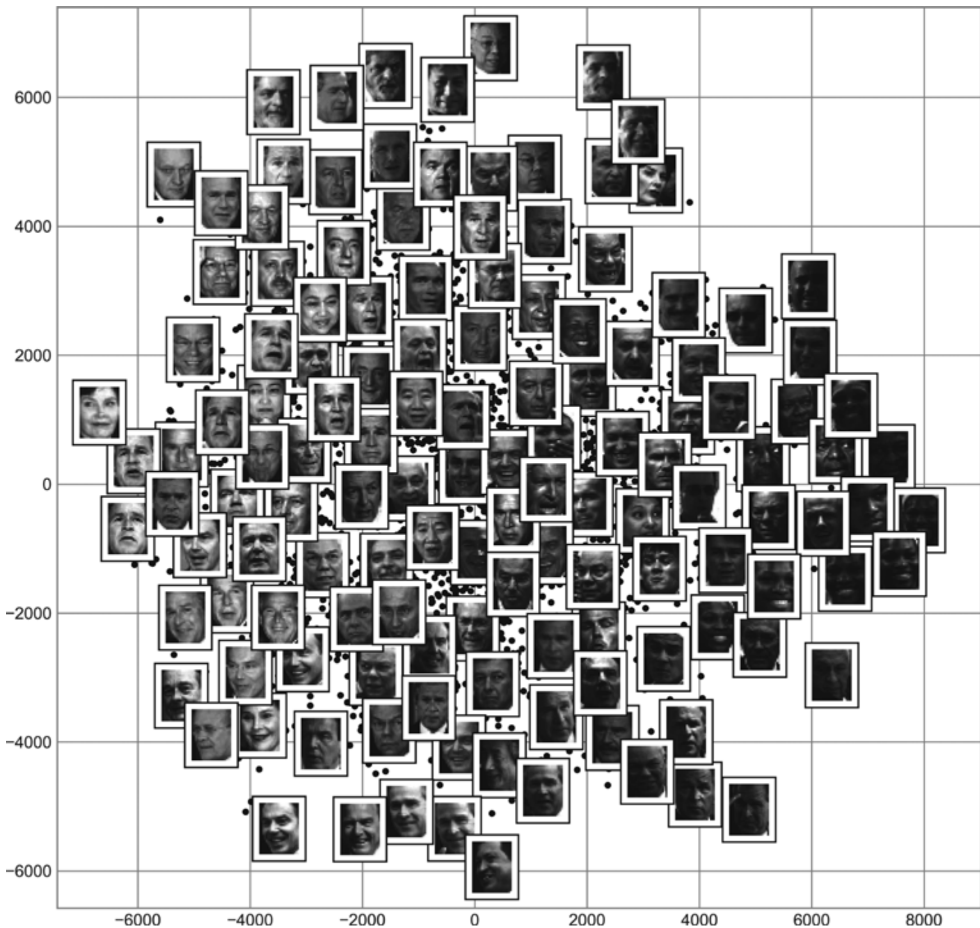


Рис. 46.13. Вложение данных с фотографиями лиц с помощью Isomap

Результат получился довольно интересный: первые два измерения Isomap, вероятно, описывают общие признаки изображений — низкую или высокую яркость изображения слева направо и общую ориентацию лица снизу вверх. Это дает нам общее представление о некоторых базовых признаках данных.

Далее можно перейти к классификации этих данных, возможно, с помощью признаков на базе многообразий в качестве входных данных для алгоритма классификации, аналогично тому, как мы поступили в главе 43.

Пример: визуализация структуры цифр

В качестве еще одного примера использования обучения на базе многообразий для визуализации рассмотрим набор MNIST рукописных цифр. Эти данные аналогичны цифрам, с которыми мы сталкивались в главе 44, но с намного большей детализацией изображений. Скачать их можно с сайта http://openml.org/ с помощью утилиты из библиотеки Scikit-Learn:

```
In [22]: from sklearn.datasets import fetch_openml
         mnist = fetch_openml('mnist_784')
         mnist.data.shape
Out[22]: (70000, 784)
```

Этот набор состоит из 70 000 изображений, каждое размером 784 пиксела (то есть изображение имеет размер 28 × 28 пикселов). Как и ранее, рассмотрим несколько первых изображений (рис. 46.14):

```
In [23]: mnist_data = np.asarray(mnist.data)
         mnist_target = np.asarray(mnist.target, dtype=int)

         fig, ax = plt.subplots(6, 8, subplot_kw=dict(xticks=[], yticks=[]))
         for i, axi in enumerate(ax.flat):
             axi.imshow(mnist_data[1250 * i].reshape(28, 28), cmap='gray_r')
```

Этот рисунок дает нам представление о разнообразии рукописных цифр в наборе данных.

Вычислим с помощью обучения на базе многообразий проекцию для этих данных. Для ускорения используем только 1/30 часть данных, то есть примерно 2000 точек данных (из-за относительно плохой масштабируемости методов обучения на базе многообразий я пришел к выводу, что несколько тысяч образцов — хорошее количество для начала, чтобы относительно быстро исследовать набор, прежде чем перейти к полномасштабным вычислениям). Результат показан на рис. 46.15.

```
In [24]: # используем только 1/30 часть данных:
         # вычисления для полного набора данных занимают слишком много времени!
         data = mnist_data[::30]
         target = mnist_target[::30]
```

```
model = Isomap(n_components=2)
proj = model.fit_transform(data)

plt.scatter(proj[:, 0], proj[:, 1], c=target,
                        cmap=plt.cm.get_cmap('jet', 10))
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5);
```

Рис. 46.14. Примеры цифр из набора MNIST

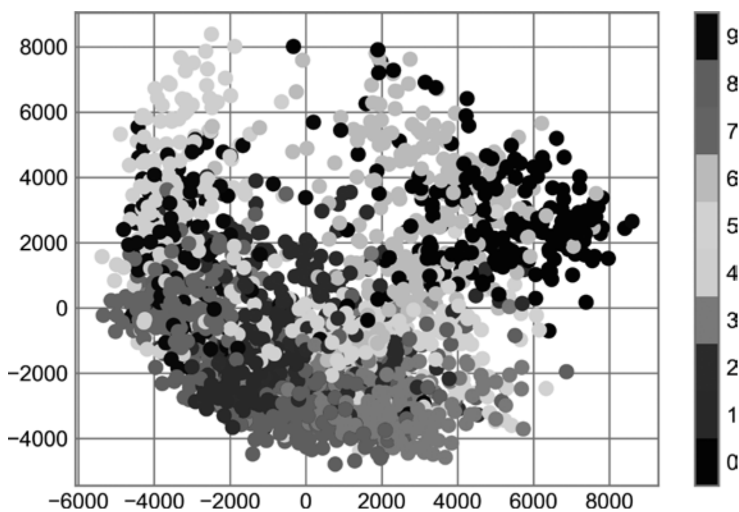


Рис. 46.15. Isomap-вложение для набора данных цифр MNIST

Полученная диаграмма рассеяния демонстрирует некоторые зависимости между точками данных, но точки на ней расположены слишком тесно. Мы можем получить больше информации, изучая за раз данные лишь об одной цифре (рис. 46.16):

```
In [25]: # Выбираем для проекции 1/4 цифр "1"
         data = mnist_data[mnist_target == 1][::4]

         fig, ax = plt.subplots(figsize=(10, 10))
         model = Isomap(n_neighbors=5, n_components=2, eigen_solver='dense')
         plot_components(data, model, images=data.reshape((-1, 28, 28)),
                         ax=ax, thumb_frac=0.05, cmap='gray_r')
```

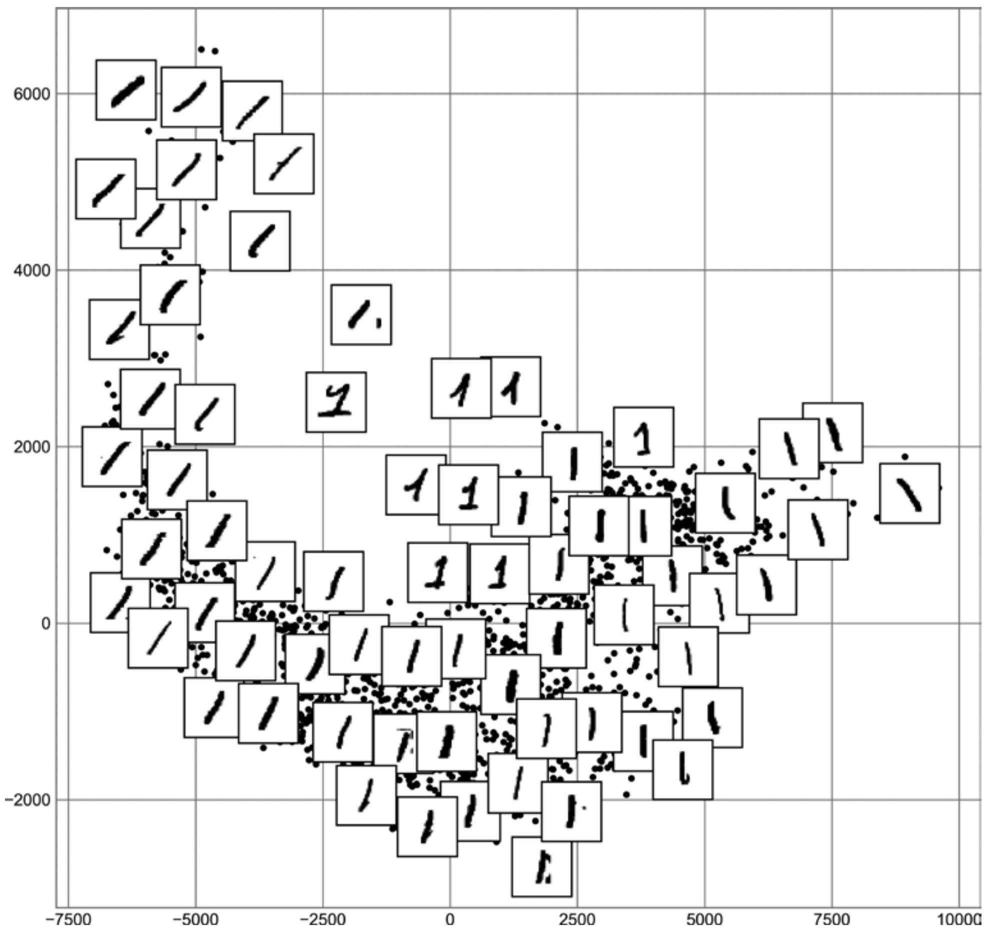


Рис. 46.16. Isomap-вложение только для единиц из набора данных о цифрах

Результат дает нам представление о разнообразии форм, которые может принимать цифра «1» в этом наборе данных. Данные располагаются вдоль широкой кривой в пространстве проекции, отражающей ориентацию цифры. При перемещении вверх по графику мы видим единицы со «шляпками» и/или «подошвами», хотя они в этом наборе данных редки. Проекция дает нам возможность обнаружить аномальные значения с проблемами в данных (например, части соседних цифр, попавших в извлеченные изображения).

Хотя само по себе для задачи классификации цифр это и не особо полезно, но может помочь нам получить представление о данных и подсказать, что делать дальше, например, какой предварительной обработке необходимо подвергнуть данные до создания конвейера классификации.

ГЛАВА 47

Заглянем глубже: кластеризация методом k средних

В предыдущих главах мы рассматривали только одну разновидность машинного обучения без учителя — понижение размерности. В этой главе мы перейдем к другому классу моделей машинного обучения без учителя — алгоритмам кластеризации. Алгоритмы кластеризации нацелены на поиск оптимального разбиения или дискретную маркировку групп точек, исходя из свойств данных.

В библиотеке Scikit-Learn и других местах имеется множество алгоритмов кластеризации, но, вероятно, наиболее простой для понимания — алгоритм *кластеризации методом k средних* (k-means clustering), реализованный в классе `sklearn.cluster.KMeans`.

Начнем с импортирования необходимых модулей:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Знакомство с методом k средних

Алгоритм *k* средних выполняет поиск заранее заданного количества кластеров в немаркированном многомерном наборе данных. Достигается это с помощью простого представления о том, что такое оптимальная кластеризация.

- *«Центр кластера»* — арифметическое среднее всех точек, относящихся к этому кластеру.
- Каждая точка ближе к центру своего кластера, чем к центрам других кластеров.

Эти два допущения составляют основу модели метода *k* средних. Далее мы рассмотрим детальнее, каким именно образом алгоритм находит это решение, а пока

возьмем простой набор данных и посмотрим на результаты работы метода *k* средних для него.

Во-первых, сгенерируем двумерный набор данных, содержащий четыре отдельных «облака». Чтобы подчеркнуть отсутствие учителя в этом алгоритме, мы не будем включать метки в визуализацию (рис. 47.1).

```
In [2]: from sklearn.datasets import make_blobs
          X, y_true = make_blobs(n_samples=300, centers=4,
                                             cluster_std=0.60, random_state=0)
          plt.scatter(X[:, 0], X[:, 1], s=50);
```

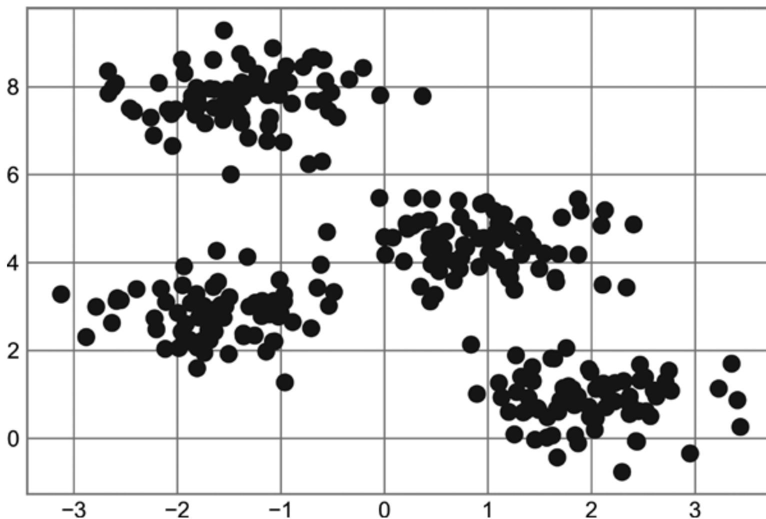


Рис. 47.1. Исходные данные для демонстрации кластеризации

Визуально выделить здесь четыре кластера не составляет труда. Алгоритм *k* средних делает это автоматически, используя API статистических оценок из библиотеки Scikit-Learn:

```
In [3]: from sklearn.cluster import KMeans
        kmeans = KMeans(n_clusters=4)
        kmeans.fit(X)
        y_kmeans = kmeans.predict(X)
```

Визуализируем результаты, выведя на график точки данных, окрашенные в соответствии с метками (рис. 47.2). Нарисуем центры кластеров, найденные методом *k* средних:

```
In [4]: plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

        centers = kmeans.cluster_centers_
        plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200);
```

Интересно отметить, что алгоритм *k* средних (по крайней мере в этом простом случае) включает точки в кластере практически так же, как мы сделали бы это вручную. Но у вас может возникнуть вопрос: как этому алгоритму удалось найти кластеры так быстро? В конце концов, количество возможных сочетаний кластеров зависит экспоненциально от числа точек данных, так что поиск методом полного перебора оказался бы чрезвычайно ресурсоемким и он здесь не требуется. Вместо него в типичном варианте метода *k* средних применяется интуитивно понятный подход, известный под названием *«максимизация математического ожидания»* (expectation-maximization, EM).

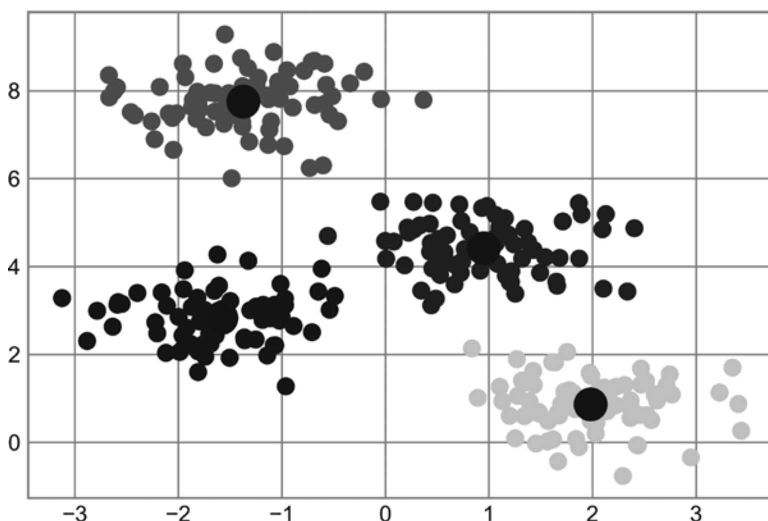


Рис. 47.2. Центры кластеров, найденных методом k средних, с окрашенными в разные цвета кластерами

Максимизация математического ожидания

Максимизация математического ожидания (EM) — мощный алгоритм, встречающийся во множестве контекстов науки о данных. Метод *k* средних — особенно простое и понятное приложение этого алгоритма, и мы рассмотрим его здесь вкратце. Подход максимизации математического ожидания состоит из следующей процедуры.

1. Выдвигаем гипотезу о центрах кластеров.
2. Повторяем до достижения сходимости:
   - *E-шаг*: приписываем точки к ближайшим центрам кластеров;
   - *M-шаг*: задаем новые центры кластеров в соответствии со средними значениями.

E-шаг, или *шаг ожидания* (expectation), назван так потому, что включает актуализацию математического ожидания того, к каким кластерам относятся точки. *M-шаг*, или *шаг максимизации* (maximization), назван так потому, что включает максимизацию некоторой целевой функции, описывающей местоположения центров кластеров. В таком случае максимизация достигается путем простого усреднения данных в кластере.

Этот алгоритм описывается во множестве изданий, но если коротко, то можно подвести его итоги следующим образом: при обычных обстоятельствах каждая итерация шагов E и M всегда будет приводить к улучшению оценки показателей кластера.

Мы можем визуализировать этот алгоритм, как показано на рис. 47.3. Для представленных здесь начальных данных алгоритм сходится всего за три итерации. (Интерактивную версию рисунка можно увидеть в онлайн-приложении (https://oreil.ly/wFnok).)

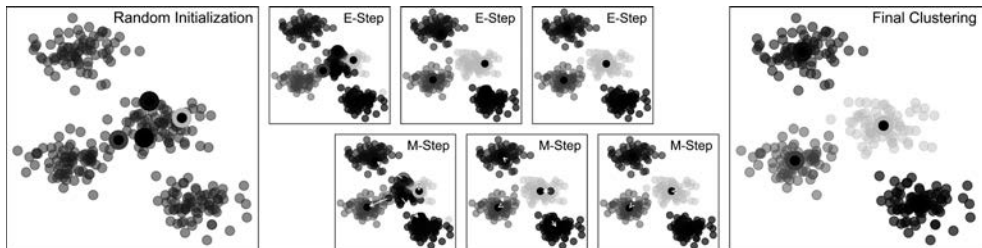


Рис. 47.3. Визуализация EM-алгоритма для метода k средних[1]

Алгоритм *k* средних достаточно прост, чтобы уместиться в нескольких строках кода. Вот простейшая его реализация (рис. 47.4):

```
In [5]: from sklearn.metrics import pairwise_distances_argmin

        def find_clusters(X, n_clusters, rseed=2):
            # 1. Выбираем случайные кластеры
            rng = np.random.RandomState(rseed)
            i = rng.permutation(X.shape[0])[:n_clusters]
            centers = X[i]

            while True:
                # 2a. Присваиваем метки в соответствии с ближайшим центром
                labels = pairwise_distances_argmin(X, centers)

                # 2b. Находим новые центры, исходя из средних значений точек
                new_centers = np.array([X[labels == i].mean(0)
                                        for i in range(n_clusters)])
```

[1] Код, генерирующий рисунки этой главы, можно найти в онлайн-приложении (https://oreil.ly/jv0wb).

```
        # 2c. Проверяем сходимость
        if np.all(centers == new_centers):
            break
        centers = new_centers

    return centers, labels

centers, labels = find_clusters(X, 4)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```

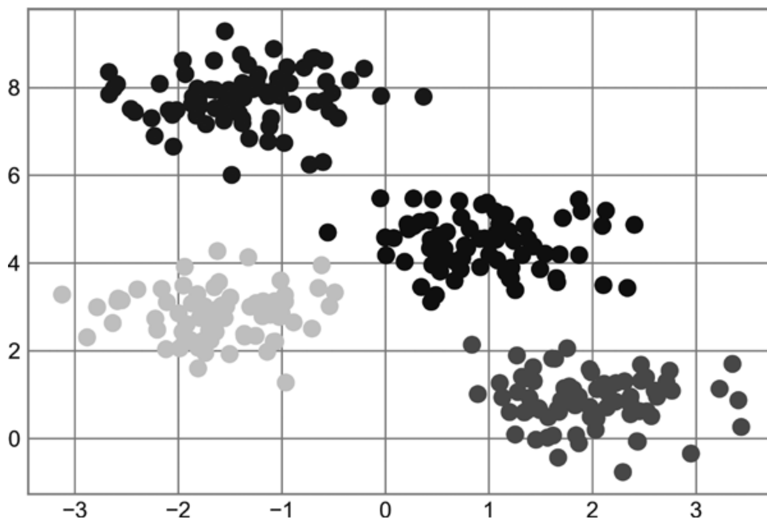


Рис. 47.4. Данные, маркированные с помощью метода k средних

Самые проверенные реализации выполняют «под капотом» немного больше действий, но основное представление о методе максимизации математического ожидания предыдущая функция дает. Однако, используя алгоритм максимизации математического ожидания, следует иметь в виду несколько нюансов.

- *Глобально оптимальный результат может оказаться недостижимым в принципе*. Во-первых, хотя процедура EM гарантированно улучшает результат на каждом шаге, уверенности в том, что она ведет к *глобально* наилучшему решению, нет. Например, если использовать нашу простую процедуру с другим начальным значением для генератора случайных чисел, то полученные начальные гипотезы приведут к неудачным результатам (рис. 47.5):

```
In [6]: centers, labels = find_clusters(X, 4, rseed=0)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```

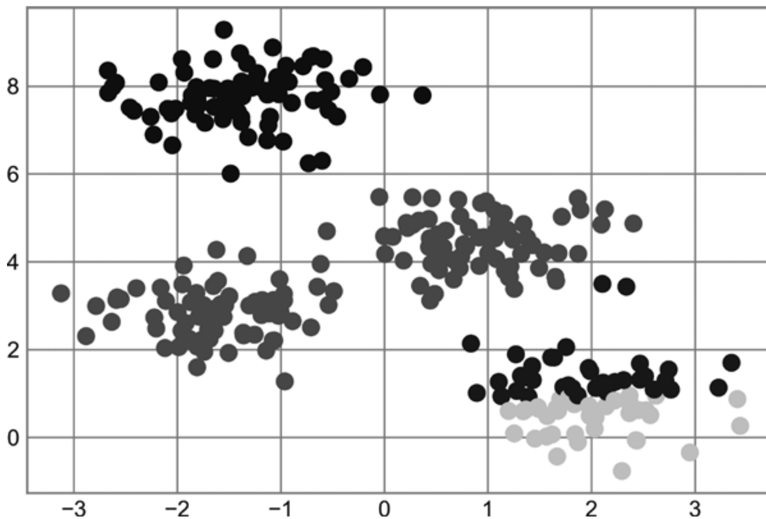


Рис. 47.5. Пример плохой сходимости в методе k средних

В этом случае EM-метод свелся к глобально неоптимальной конфигурации, поэтому его часто выполняют для нескольких начальных гипотез, что и делает по умолчанию библиотека Scikit-Learn (это задается с помощью параметра `n_init`, по умолчанию имеющего значение 10).

- *Количество кластеров должно выбираться заранее.* Еще одна часто встречающаяся проблема метода *k* средних — необходимость заранее определить ожидаемое количество кластеров: он не умеет вычислять количество кластеров на основе данных. Например, если предложить алгоритму выделить шесть кластеров, он с радостью это сделает и найдет шесть оптимальных кластеров, как показано на рис. 47.6:

```
In [7]: labels = KMeans(6, random_state=0).fit_predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```

Осмысленный ли результат получен — сказать трудно. Один из полезных в этом случае и интуитивно понятных подходов, который мы не станем обсуждать подробно, — силуэтный анализ (https://oreil.ly/xybmq).

В качестве альтернативы можно воспользоваться более сложным алгоритмом кластеризации с лучшим количественным показателем зависимости качества аппроксимации от количества кластеров (например, смесь гауссовых распределений, см. главу 48) или с возможностью выбора приемлемого количества кластеров (например, методы DBSCAN, сдвиг среднего или распространение аффинности (affinity propagation), реализованные в модуле `sklearn.cluster`).

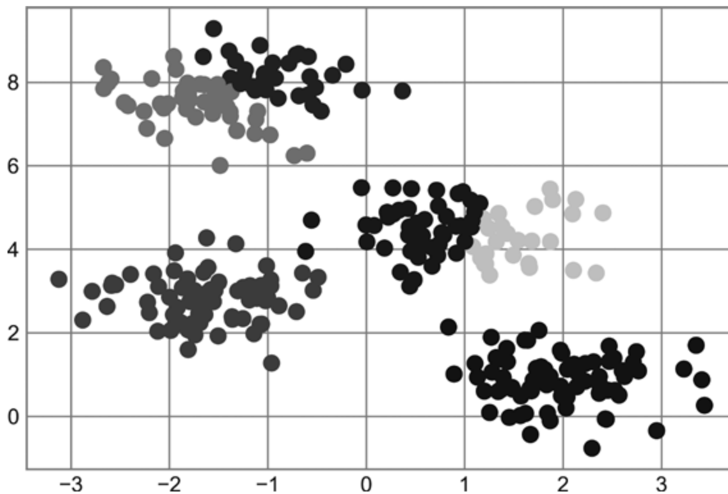


Рис. 47.6. Пример неудачного выбора количества кластеров

- *Применение метода k средних ограничивается случаем линейных границ кластеров*. Базовое допущение модели *k* средних (точки должны быть ближе к центру собственного кластера, чем других) означает, что этот алгоритм зачастую будет неэффективен в случае сложной геометрии кластеров.

  В частности, границы между кластерами в методе *k* средних всегда будут линейными, а значит, он будет плохо работать в случае более сложных границ. Рассмотрим следующие данные и метки кластеров, найденные для них обычным методом *k* средних (рис. 47.7):

```
In [8]: from sklearn.datasets import make_moons
        X, y = make_moons(200, noise=.05, random_state=0)

In [9]: labels = KMeans(2, random_state=0).fit_predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```

  Эта ситуация напоминает обсуждавшуюся в главе 43, где мы использовали ядерное преобразование для проецирования данных в пространство с большим числом измерений, в котором возможно линейное разделение. Можно попробовать воспользоваться той же уловкой, чтобы метод *k* средних распознал нелинейные границы.

  Одна из версий этого ядерного метода *k* средних реализована в библиотеке Scikit-Learn в классе `SpectralClustering`. Она использует граф ближайших соседей для вычисления представления данных в большем числе измерений, после чего задает соответствие меток с помощью алгоритма *k* средних (рис. 47.8):

```
In [10]: from sklearn.cluster import SpectralClustering
         model = SpectralClustering(n_clusters=2,
                                    affinity='nearest_neighbors',
                                    assign_labels='kmeans')
         labels = model.fit_predict(X)
         plt.scatter(X[:, 0], X[:, 1], c=labels,
                     s=50, cmap='viridis');
```

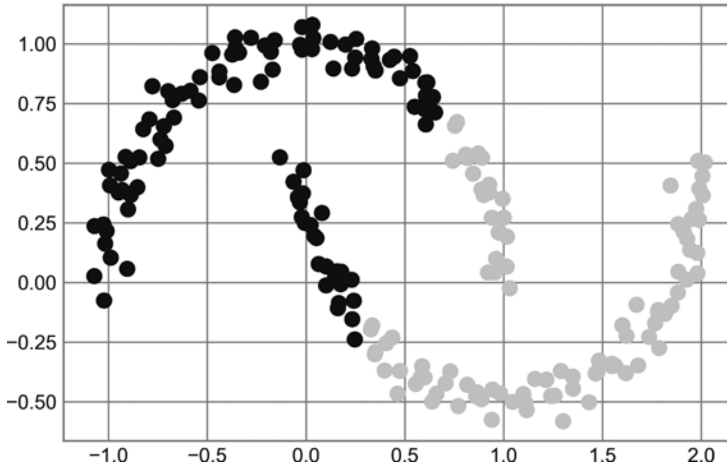


Рис. 47.7. Неудача метода k средних в случае нелинейных границ

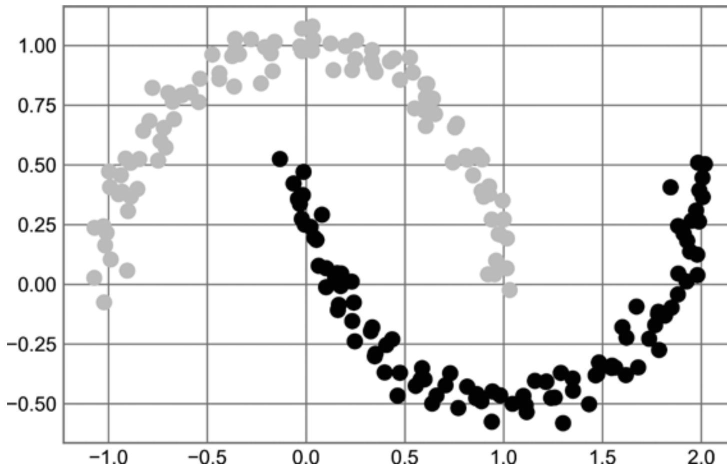


Рис. 47.8. Нелинейные границы, обнаруженные с помощью SpectralClustering

Как видите, с помощью этого ядерного преобразования метод *k* средних способен обнаруживать более сложные нелинейные границы между кластерами.

- *Метод k средних работает довольно медленно в случае большого количества образцов.* Алгоритм может замедляться с ростом количества образцов, потому что в каждой итерации методу *k* средних необходимо обращаться к каждой точке в наборе данных. Интересно, можно ли смягчить это требование относительно использования всех данных в каждой итерации, например, применить лишь подмножество данных для корректировки центров кластеров на каждом шаге? Эта идея лежит в основе пакетных алгоритмов *k* средних, один из которых реализован в классе `sklearn.cluster.MiniBatchKMeans`. Их интерфейс не отличается от обычного `KMeans`. В дальнейшем мы рассмотрим пример их использования.

Примеры

При соблюдении некоторой осторожности в отношении вышеупомянутых ограничений можно успешно использовать метод *k* средних во множестве ситуаций. Рассмотрим несколько примеров.

Пример 1: применение метода k средних для распознавания рукописных цифр

Для начала рассмотрим применение метода *k* средних для распознавания тех рукописных цифр, которые мы видели в главах 44 и 45. Попробуем воспользоваться методом *k* средних для распознавания схожих цифр *без использования информации об исходных метках*. Это напоминает первый шаг извлечения смысла из нового набора данных, для которого отсутствует какая-либо *априорная* информация о метках.

Начнем с загрузки цифр, а затем перейдем к поиску кластеров. Напомню, что набор данных с цифрами состоит из 1797 образцов с 64 признаками, где каждый из признаков определяет яркость одного пиксела в изображении размером 8 × 8:

```
In [11]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.data.shape
Out[11]: (1797, 64)
```

Кластеризация выполняется так же, как и ранее:

```
In [12]: kmeans = KMeans(n_clusters=10, random_state=0)
         clusters = kmeans.fit_predict(digits.data)
         kmeans.cluster_centers_.shape
Out[12]: (10, 64)
```

В результате мы получили десять кластеров в 64-мерном пространстве. Обратите внимание, что и центры кластеров представляют собой 64-мерные точки, а значит,

их можно интерпретировать как «типичные» цифры в кластере. Посмотрим, что представляют собой эти центры кластеров (рис. 47.9):

```
In [13]: fig, ax = plt.subplots(2, 5, figsize=(8, 3))
         centers = kmeans.cluster_centers_.reshape(10, 8, 8)
         for axi, center in zip(ax.flat, centers):
             axi.set(xticks=[], yticks=[])
             axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```

Рис. 47.9. Центры кластеров

Как видите, алгоритм `KMeans` *даже без меток* способен определить кластеры, чьи центры представляют легкоузнаваемые цифры, за исключением разве что цифр «1» и «8».

В силу того, что алгоритм *k* средних ничего не знает о сущности кластеров, метки 0–9 могут оказаться перепутаны местами. Исправить это можно, задав соответствие всех полученных меток кластеров имеющимся в них фактическим меткам:

```
In [14]: from scipy.stats import mode

         labels = np.zeros_like(clusters)
         for i in range(10):
             mask = (clusters == i)
             labels[mask] = mode(digits.target[mask])[0]
```

Теперь можно проверить, насколько точно кластеризация без учителя определила подобие цифр в наших данных:

```
In [15]: from sklearn.metrics import accuracy_score
         accuracy_score(digits.target, labels)
Out[15]: 0.7935447968836951
```

С помощью простого алгоритма *k* средних мы задали правильную группировку для почти 80 % исходных цифр! Посмотрим на матрицу различий (рис. 47.10):

```
In [16]: from sklearn.metrics import confusion_matrix
         import seaborn as sns
         mat = confusion_matrix(digits.target, labels)
```

```
sns.heatmap(mat.T, square=True, annot=True, fmt='d',
            cbar=False, cmap='Blues',
            xticklabels=digits.target_names,
            yticklabels=digits.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```

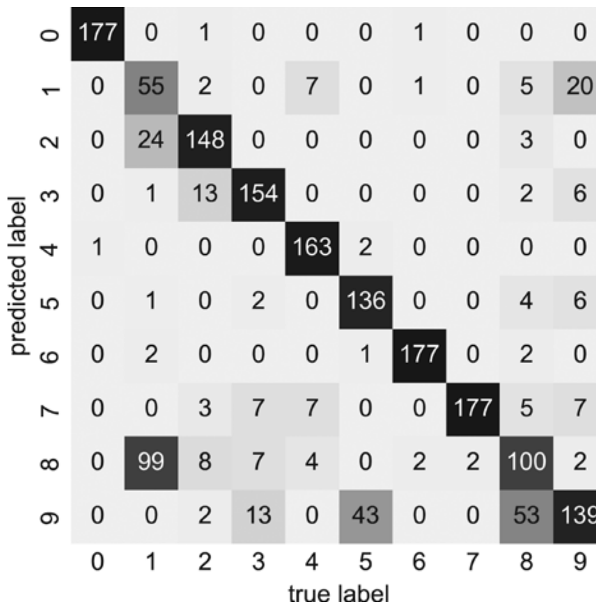


Рис. 47.10. Матрица различий для классификатора методом k средних

Как и следовало ожидать, судя по визуализированным ранее центрам кластеров, больше всего путаницы между единицами и восьмерками. Но все равно с помощью метода *k* средних мы фактически создали классификатор цифр *без каких-либо известных меток*!

Попробуем пойти еще дальше. Воспользуемся для предварительной обработки данных до выполнения *k* средних алгоритмом стохастического вложения соседей на основе распределения Стьюдента (t-SNE), упоминавшимся в главе 46. t-SNE — нелинейный алгоритм, особенно хорошо умеющий сохранять точки внутри кластеров. Посмотрим, как он работает:

```
In [17]: from sklearn.manifold import TSNE

         # Проекция данных: выполнение этого шага займет несколько секунд
         tsne = TSNE(n_components=2, init='random',
                     learning_rate='auto',random_state=0)
         digits_proj = tsne.fit_transform(digits.data)
```

```
        # Расчет кластеров
        kmeans = KMeans(n_clusters=10, random_state=0)
        clusters = kmeans.fit_predict(digits_proj)

        # Перестановка меток местами
        labels = np.zeros_like(clusters)
        for i in range(10):
            mask = (clusters == i)
            labels[mask] = mode(digits.target[mask])[0]

        # Оценка точности
        accuracy_score(digits.target, labels)
Out[17]: 0.9415692821368948
```

Точность, даже *без использования меток*, составляет почти 94 %. При разумном использовании алгоритмы машинного обучения без учителя демонстрируют отличные результаты: они могут извлекать информацию из набора данных, даже когда сделать это вручную или визуально очень непросто.

Пример 2: использование метода k средних для сжатия цветов

Одно из интересных приложений кластеризации — сжатие цветов в изображениях (этот пример заимствован из раздела «Color Quantization Using K-Means» (https://oreil.ly/TwsxU) в документации библиотеки Scikit-Learn). Допустим, что у нас есть изображение с миллионами цветов. Почти во всех изображениях большая часть цветов не используется и цвета многих пикселов изображения будут похожи или даже совпадать.

Например, рассмотрим изображение, показанное на рис. 47.11, взятом из модуля `datasets` библиотеки Scikit-Learn (для работы следующего кода у вас должен быть установлен пакет `PIL` языка Python):

```
In [18]: # Обратите внимание: для работы этого кода
         # должен быть установлен пакет PIL
         from sklearn.datasets import load_sample_image
         china = load_sample_image("china.jpg")
         ax = plt.axes(xticks=[], yticks=[])
         ax.imshow(china);
```

Само изображение хранится в трехмерном массиве размера (`высота`, `ширина`, `RGB`), содержащем значения по красному/синему/зеленому каналам в виде целочисленных значений от 0 до 255:

```
In [19]: china.shape
Out[19]: (427, 640, 3)
```

Рис. 47.11. Исходное изображение

Этот набор пикселов можно рассматривать как облако точек в трехмерном цветовом пространстве. Изменим форму данных на `[n_samples, n_features]` и масштабируем шкалу цветов так, чтобы они располагались между 0 и 1:

```
In [20]: data = china / 255.0  # используем шкалу 0...1
         data = data.reshape(-1, 3)
         data.shape
Out[20]: (273280, 3)
```

Визуализируем эти пикселы в данном цветовом пространстве, используя подмножество из 10 000 пикселов для ускорения работы (рис. 47.12):

```
In [21]: def plot_pixels(data, title, colors=None, N=10000):
             if colors is None:
                 colors = data

             # Выбираем случайное подмножество
             rng = np.random.default_rng(0)
             i = rng.permutation(data.shape[0])[:N]
             colors = colors[i]
             R, G, B = data[i].T

             fig, ax = plt.subplots(1, 2, figsize=(16, 6))
             ax[0].scatter(R, G, color=colors, marker='.')
             ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

             ax[1].scatter(R, B, color=colors, marker='.')
             ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

             fig.suptitle(title, size=20);

In [22]: plot_pixels(data, title='Input color space: 16 million possible colors')
```

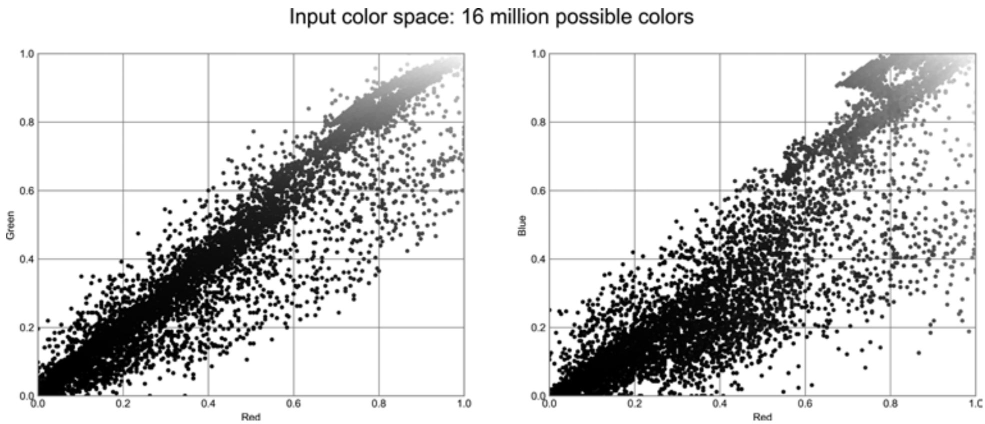


Рис. 47.12. Распределение пикселов в цветовом пространстве RGB

Теперь уменьшим количество цветов с 16 миллионов до 16 путем кластеризации методом *k* средних на пространстве пикселов. Так как наш набор данных очень велик, воспользуемся мини-пакетным методом *k* средних, который вычисляет результат гораздо быстрее, чем стандартный метод *k* средних, за счет работы с подмножествами из набора данных (рис. 47.13):

```
In [23]: from sklearn.cluster import MiniBatchKMeans
         kmeans = MiniBatchKMeans(16)
         kmeans.fit(data)
         new_colors = kmeans.cluster_centers_[kmeans.predict(data)]

         plot_pixels(data, colors=new_colors,
                     title="Reduced color space: 16 colors")
```

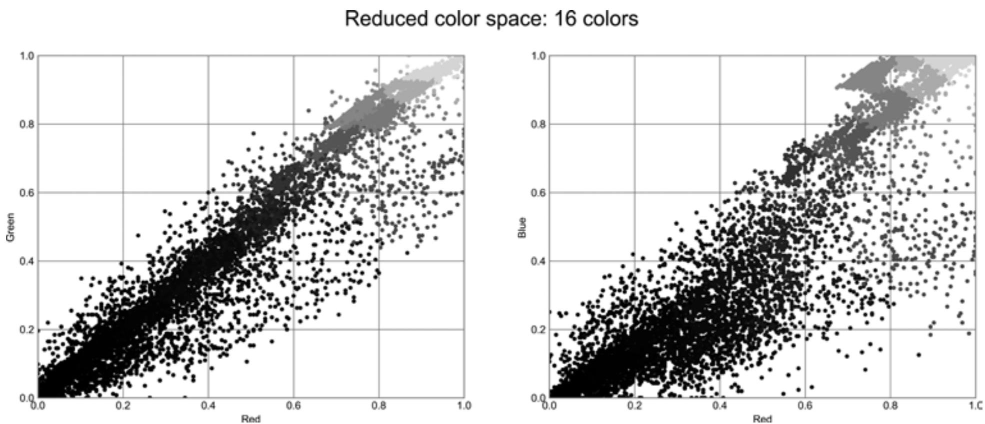


Рис. 47.13. Шестнадцать кластеров в цветовом пространстве RGB

В результате исходные пикселы перекрашиваются в другие цвета: каждый пиксел получает цвет ближайшего центра кластера. Рисуя эти новые цвета в пространстве изображения вместо пространства пикселов, видим эффект перекрашивания (рис. 47.14):

```
In [24]: china_recolored = new_colors.reshape(china.shape)

         fig, ax = plt.subplots(1, 2, figsize=(16, 6),
                                subplot_kw=dict(xticks=[], yticks=[]))
         fig.subplots_adjust(wspace=0.05)
         ax[0].imshow(china)
         ax[0].set_title('Original Image', size=16)
         ax[1].imshow(china_recolored)
         ax[1].set_title('16-color Image', size=16);
```

Рис. 47.14. Полноцветное изображение *(слева)* и 16-цветное *(справа)*

Некоторые детали в изображении справа утрачены, но изображение в целом остается вполне узнаваемым. Коэффициент сжатия этого изображения — почти 1 миллион! Существуют лучшие способы сжатия информации в изображениях, но этот пример демонстрирует возможности творческого подхода к методам машинного обучения без учителя, таким как метод *k* средних.

ГЛАВА 48

Заглянем глубже: смеси гауссовых распределений

Модель кластеризации методом *k* средних, которую мы обсуждали в предыдущей главе, проста и относительно легка для понимания, но ее простота приводит к сложностям в применении. В частности, не вероятностная природа метода *k* средних и использование им простого расстояния от центра кластера для определения принадлежности к кластеру приводит к низкой эффективности во многих встречающихся на практике ситуациях. В этой главе мы рассмотрим смеси гауссовых распределений (Gaussian Mixture Models, GMM), которые можно рассматривать в качестве развития метода *k* средних, но которые могут также стать мощным инструментом для статистических оценок, выходящих за пределы простой кластеризации.

Начнем с импорта необходимых модулей:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Причины появления GMM: недостатки метода k средних

Рассмотрим некоторые недостатки метода *k* средних и задумаемся о том, как можно усовершенствовать кластерную модель. Как мы уже видели в предыдущей главе, в случае простых, хорошо разделяемых данных метод *k* средних обеспечивает удовлетворительные результаты кластеризации.

Например, для случая простых «облаков» данных алгоритм *k* средних позволяет быстро маркировать кластеры достаточно близко к тому, как мы маркировали бы их на глаз (рис. 48.1):

```
In [2]: # Генерируем данные
        from sklearn.datasets import make_blobs
        X, y_true = make_blobs(n_samples=400, centers=4,
                               cluster_std=0.60, random_state=0)
        X = X[:, ::-1] # Транспонируем оси координат для удобства

In [3]: # Выводим данные на график с полученными методом k средних метками
        from sklearn.cluster import KMeans
        kmeans = KMeans(4, random_state=0)
        labels = kmeans.fit(X).predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

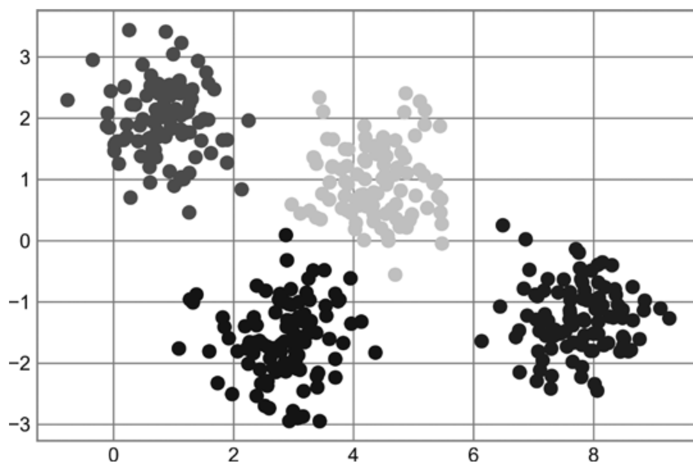


Рис. 48.1. Метки для простых данных, полученные методом k средних

Интуитивно можно ожидать, что при кластеризации одним точкам метки присваиваются с большей долей достоверности, чем другим. Например, два средних кластера чуть-чуть пересекаются, поэтому нельзя быть уверенными в том, к какому из них отнести точки, находящиеся посередине между ними. К сожалению, в модели *k* средних отсутствует внутренняя мера вероятности или достоверности отнесения точек к кластерам (хотя можно использовать бутстрэппинг для оценки этой вероятности). Для этого необходимо рассмотреть возможность обобщения модели.

Модель *k* средних можно рассматривать, в частности, как помещающую окружности (гиперсферы в пространствах большей размерности) с центрами в центрах кластеров и радиусами, соответствующими расстоянию до наиболее удаленной

точки кластера. Этот радиус задает жесткую границу соответствия точки кластеру в обучающей последовательности: все точки, находящиеся снаружи этой окружности, не считаются членами кластера. Визуализируем эту модель кластера с помощью следующей функции (рис. 48.2):

```
In [4]: from sklearn.cluster import KMeans
        from scipy.spatial.distance import cdist

        def plot_kmeans(kmeans, X, n_clusters=4, rseed=0, ax=None):
            labels = kmeans.fit_predict(X)

            # Выводим на график входные данные
            ax = ax or plt.gca()
            ax.axis('equal')
            ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)

            # Выводим на график представление модели k средних
            centers = kmeans.cluster_centers_
            radii = [cdist(X[labels == i], [center]).max()
                     for i, center in enumerate(centers)]
            for c, r in zip(centers, radii):
                ax.add_patch(plt.Circle(c, r, ec='black', fc='lightgray',
                                        lw=3, alpha=0.5, zorder=1))

In [5]: kmeans = KMeans(n_clusters=4, random_state=0)
        plot_kmeans(kmeans, X)
```

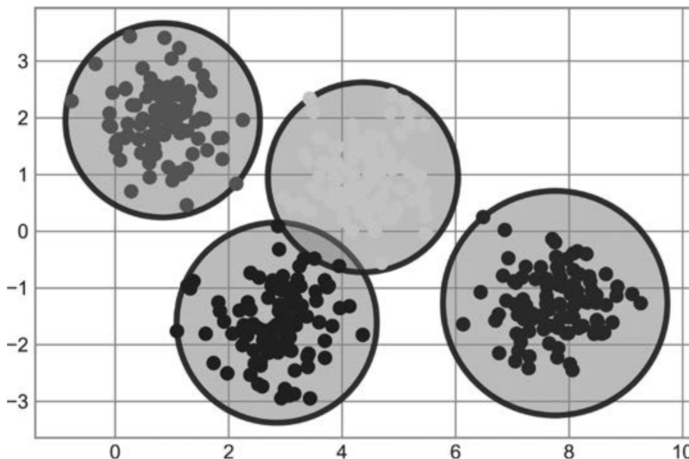


Рис. 48.2. Круглые кластеры, подразумеваемые моделью k средних

Немаловажный нюанс, касающийся метода *k* средних, — эти модели кластеров *обязательно должны иметь форму окружностей*: метод *k* средних не умеет работать

с овальными или эллипсовидными кластерами. Так, например, если преобразовать те же данные, то присвоенные метки окажутся перепутаны (рис. 48.3):

```
In [6]: rng = np.random.RandomState(13)
        X_stretched = np.dot(X, rng.randn(2, 2))

        kmeans = KMeans(n_clusters=4, random_state=0)
        plot_kmeans(kmeans, X_stretched)
```

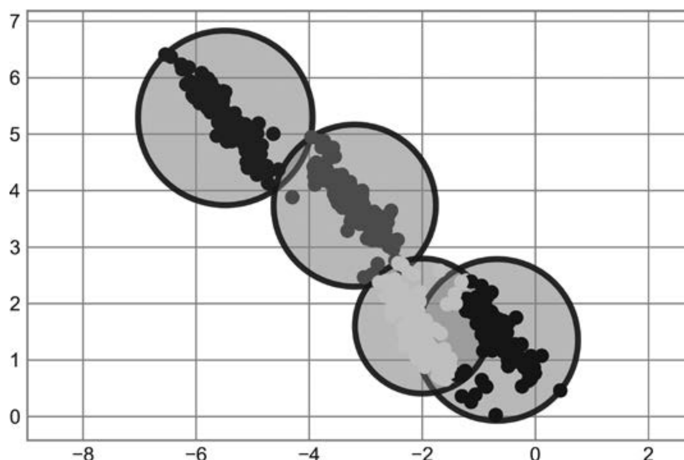


Рис. 48.3. Неудовлетворительная работа метода k средних в случае кластеров некруглой формы

Визуально заметно, что преобразованные кластеры имеют некруглую форму, а значит, круглые кластеры плохо подходят для их описания. Тем не менее метод *k* средних недостаточно гибок для учета этого нюанса и пытается втиснуть данные в четыре круглых кластера. Это приводит к неправильному присваиванию меток кластеров в местах перекрытия получившихся окружностей. Это особенно заметно справа внизу на рис. 48.3. Можно было бы попытаться решить эту проблему путем предварительной обработки данных с помощью PCA (см. главу 45), но на практике нет никаких гарантий, что подобная глобальная операция позволит вместить в окружности отдельные группы точек данных.

Отсутствие гибкости в вопросе формы кластеров и вероятностного назначения меток кластеров — два недостатка метода *k* средних, означающих, что для многих наборов данных (особенно с небольшим числом измерений) он будет работать не столь хорошо, как хотелось бы.

Можно попытаться избавиться от этих недостатков путем обобщения модели *k* средних. Например, можно оценивать степень достоверности назначения меток кластеров, сравнивая расстояния от каждой точки до всех центров кластеров, не сосредоточивая внимание лишь на ближайшем. Можно также разрешить эллип-

совидную форму границ кластеров, а не только круглую, чтобы учесть кластеры некруглой формы. Оказывается, что это базовые составляющие другой разновидности модели кластеризации — смеси гауссовых распределений.

Обобщение EM-модели: смеси гауссовых распределений

Смесь гауссовых распределений (gaussian mixture model, GMM) нацелена на поиск многомерных гауссовых распределений вероятностей, наилучшим образом моделирующих любой набор исходных данных. В простейшем случае смеси гауссовых распределений можно использовать для поиска кластеров аналогично методу *k* средних (рис. 48.4):

```
In [7]: from sklearn.mixture import GaussianMixture
        gmm = GaussianMixture(n_components=4).fit(X)
        labels = gmm.predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

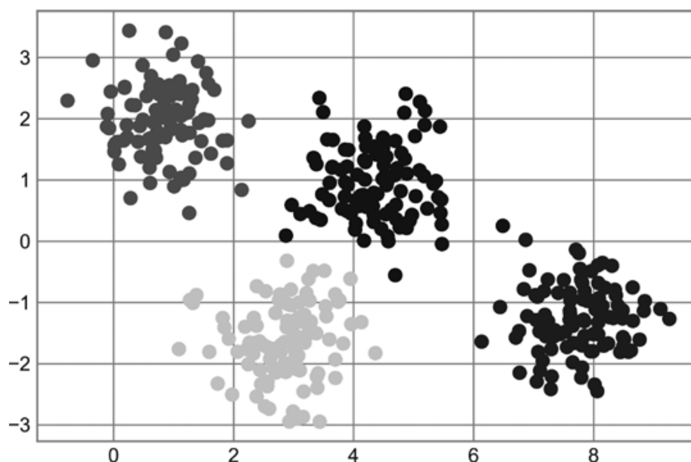


Рис. 48.4. Метки смеси гауссовых распределений для наших данных

В силу того, что GMM базируется на вероятностной модели, с ее помощью можно также присваивать метки кластеров на вероятностной основе — в библиотеке Scikit-Learn это можно сделать вызовом метода `predict_proba`. Он возвращает матрицу размера `[n_samples, n_clusters]`, содержащую оценки вероятностей принадлежности точки к конкретному кластеру:

```
In [8]: probs = gmm.predict_proba(X)
        print(probs[:5].round(3))
```

```
Out[8]: [[0.    0.531 0.469 0.   ]
         [0.    0.    0.    1.   ]
         [0.    0.    0.    1.   ]
         [0.    1.    0.    0.   ]
         [0.    0.    0.    1.   ]]
```

Для визуализации этих вероятностей можно, например, сделать размеры точек пропорциональными степени достоверности их предсказания. Глядя на рис. 48.5, можно увидеть, что как раз точки на границах между кластерами отражают эту неопределенность назначения точек кластерам:

```
In [9]: size = 50 * probs.max(1) ** 2  # Возведение в квадрат усиливает
                                        # влияние различий
        plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=size);
```

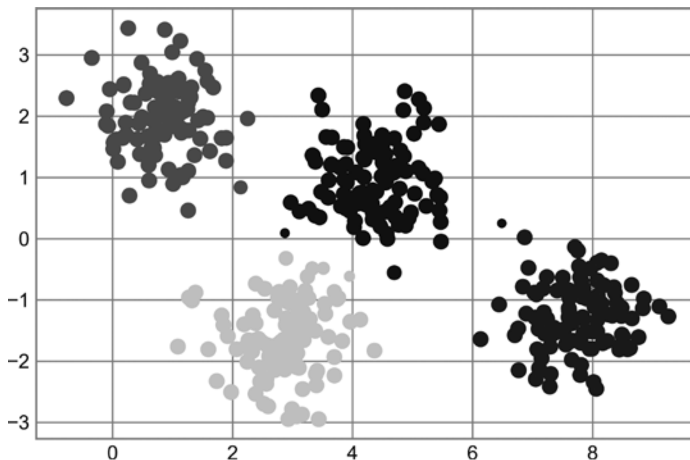


Рис. 48.5. Вероятностные метки GMM: вероятности пропорциональны размерам точек

За кулисами смесь гауссовых распределений очень напоминает метод *k* средних: она использует подход с максимизацией математического ожидания, который с качественной точки зрения делает следующее.

1. Выбирает первоначальные гипотезы относительно расположения и формы кластеров.
2. Повторяет до достижения сходимости:
   - *E-шаг* — для каждой точки находит веса, определяющие вероятность ее принадлежности к каждому кластеру;
   - *M-шаг* — для каждого кластера корректирует его местоположение, нормализацию и форму, опираясь на информацию обо *всех* точках данных с учетом их весов.

В результате каждый кластер оказывается связан не со сферой, имеющей четкую границу, а с гладкой гауссовой моделью. Аналогично подходу максимизации математического ожидания в методе *k* средних этот алгоритм иногда может промахиваться мимо наилучшего из возможных решений, поэтому на практике применяют несколько случайных наборов начальных значений.

Давайте создадим функцию, чтобы упростить визуализацию кластеров, полученных методом GMM, которая рисует эллипсы на основе информации, получаемой на выходе из GMM:

```
In [10]: from matplotlib.patches import Ellipse

         def draw_ellipse(position, covariance, ax=None, **kwargs):
             """Рисует эллипс с заданными местоположением и ковариацией"""
             ax = ax or plt.gca()

             # Преобразуем ковариацию в главные оси координат
             if covariance.shape == (2, 2):
                 U, s, Vt = np.linalg.svd(covariance)
                 angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
                 width, height = 2 * np.sqrt(s)
             else:
                 angle = 0
                 width, height = 2 * np.sqrt(covariance)

             # Рисуем эллипс
             for nsig in range(1, 4):
                 ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                                      angle, **kwargs))

         def plot_gmm(gmm, X, label=True, ax=None):
             ax = ax or plt.gca()
             labels = gmm.fit(X).predict(X)
             if label:
                 ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis',
                            zorder=2)
             else:
                 ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
             ax.axis('equal')

             w_factor = 0.2 / gmm.weights_.max()
             for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weights_):
                 draw_ellipse(pos, covar, alpha=w * w_factor)
```

Теперь посмотрим, какие результаты выдает четырехкомпонентный метод GMM для наших данных (рис. 48.6):

```
In [11]: gmm = GaussianMixture(n_components=4, random_state=42)
         plot_gmm(gmm, X)
```

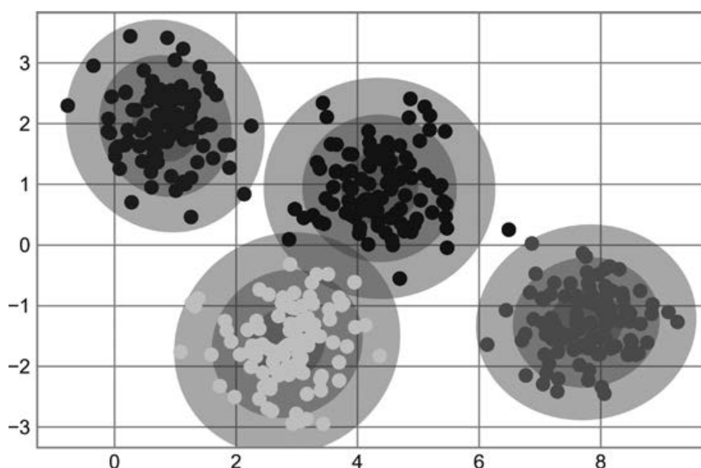


Рис. 48.6. Четырехкомпонентный метод GMM в случае круглых кластеров

Аналогично применим метод GMM к «растянутому» набору данных. С учетом полной ковариации модель будет хорошо аппроксимировать даже очень продолговатые, вытянутые в длину кластеры (рис. 48.7):

```
In [12]: gmm = GaussianMixture(n_components=4, covariance_type='full',
                               random_state=42)
         plot_gmm(gmm, X_stretched)
```

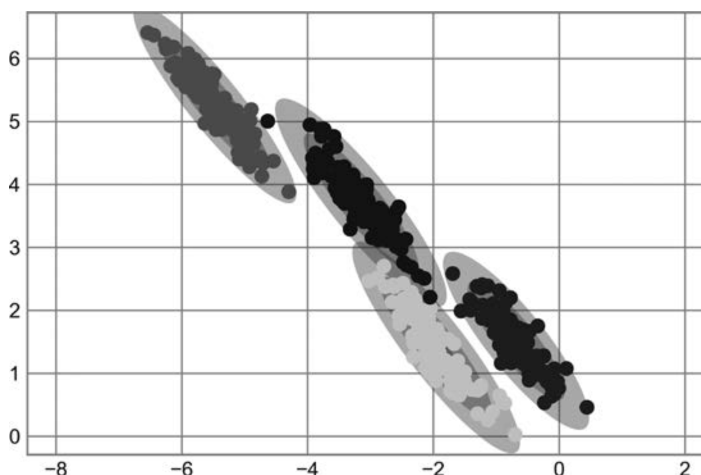


Рис. 48.7. Четырехкомпонентный метод GMM в случае некруглых кластеров

Как видите, метод GMM решает две известные нам основные практические проблемы метода *k* средних.

Выбор типа ковариации

Если внимательно посмотреть на предыдущие фрагменты кода, то можно заметить, что в каждом были заданы разные значения параметра `covariance_type`. Этот гиперпараметр управляет степенями свободы форм кластеров. Очень важно для любой задачи задавать его значения аккуратно. Значение по умолчанию — `covariance_type="diag"` — предполагает возможность независимо задать размеры кластера по всем измерениям с выравниванием полученного эллипса по осям координат. Несколько более простая и быстрая модель — `covariance_type="spherical"` — ограничивает форму кластера так, что все измерения равнозначны между собой. Получающаяся в этом случае кластеризация будет аналогична методу *k* средних, хотя и не полностью идентична. Вариант с `covariance_type="full"` представляет более сложную и требующую бо́льших вычислительных затрат модель (особенно с ростом числа измерений), в которой любой из кластеров может быть эллипсом с произвольной ориентацией. Графическое представление этих трех вариантов для одного кластера приведено на рис. 48.8.

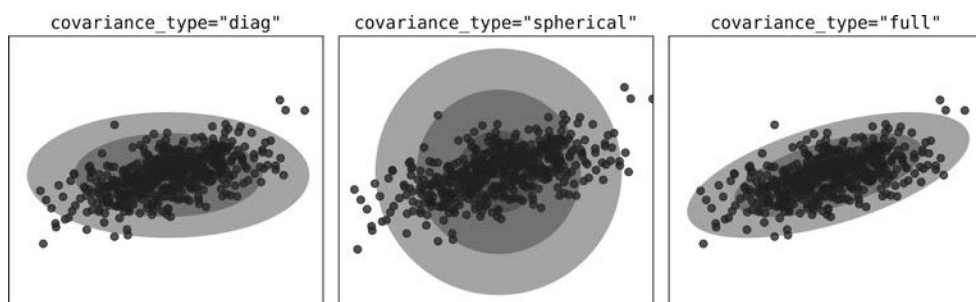


Рис. 48.8. Визуализация типов ковариации метода GMM[1]

GMM как метод оценки плотности распределения

GMM часто относят к алгоритмам кластеризации, но, по существу, это алгоритм *оценки плотности распределения*. Соответственно, аппроксимация каких-либо данных методом GMM формально является не моделью кластеризации, а генеративной вероятностной моделью, описывающей распределение данных.

В качестве примера рассмотрим данные, сгенерированные с помощью функции `make_moons` из библиотеки Scikit-Learn (рис. 48.9), которые мы уже рассматривали в главе 47:

```
In [13]: from sklearn.datasets import make_moons
         Xmoon, ymoon = make_moons(200, noise=.05, random_state=0)
         plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```

[1] Код, генерирующий этот рисунок, можно найти в онлайн-приложении (https://oreil.ly/jv0wb).

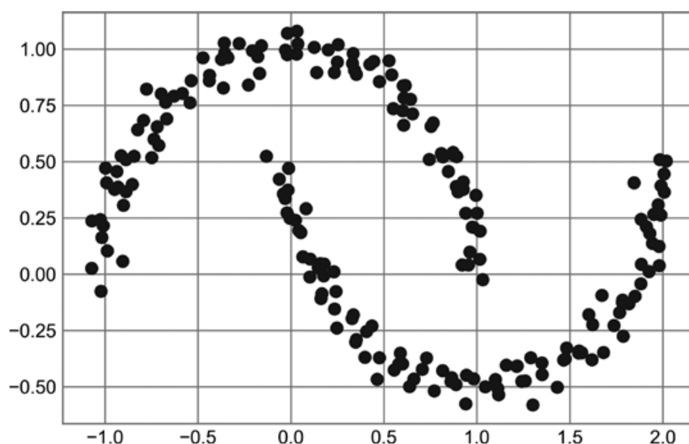


Рис. 48.9. Использование метода GMM для выделения кластеров с нелинейными границами

Если попытаться применить двухкомпонентный метод GMM как модель кластеризации, то практическая пригодность результатов окажется сомнительной (рис. 48.10):

```
In [14]: gmm2 = GaussianMixture(n_components=2, covariance_type='full',
                               random_state=0)
         plot_gmm(gmm2, Xmoon)
```

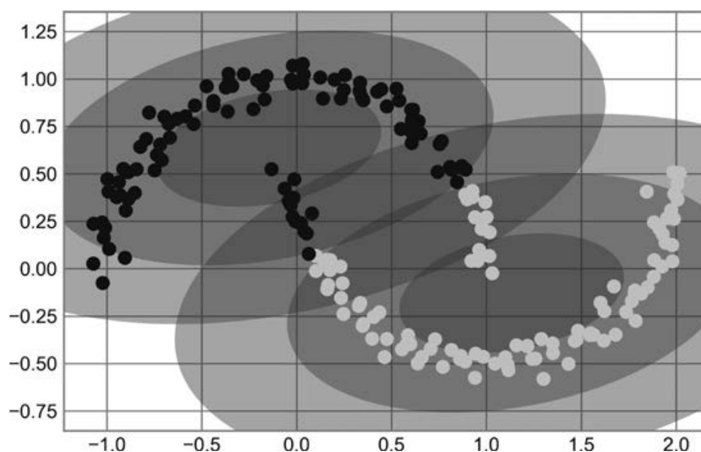


Рис. 48.10. Двухкомпонентная GMM-аппроксимация в случае нелинейных кластеров

Но если взять намного больше компонент и проигнорировать метки кластеров, то мы получим намного более точную аппроксимацию исходных данных (рис. 48.11).

```
In [15]: gmm16 = GaussianMixture(n_components=16,  covariance_type='full',
                                                    random_state=0)
            plot_gmm(gmm16, Xmoon, label=False)
```

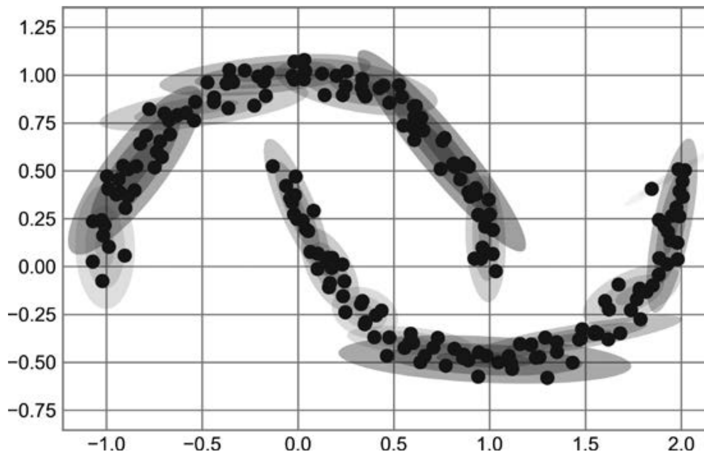


Рис. 48.11. Использование большого количества кластеров GMM для моделирования распределения точек

В данном случае смесь 16 нормальных распределений служит не для поиска отдельных кластеров данных, а для моделирования общего *распределения* входных данных. Это генеративная модель распределения, то есть GMM предоставляет нам способ генерации новых случайных данных, расположенных аналогично исходным. Например, вот 400 новых точек, полученных из этой аппроксимации 16-компонентным алгоритмом GMM (рис. 48.12):

```
In [16]: Xnew, ynew = gmm16.sample(400)
         plt.scatter(Xnew[:, 0], Xnew[:, 1]);
```

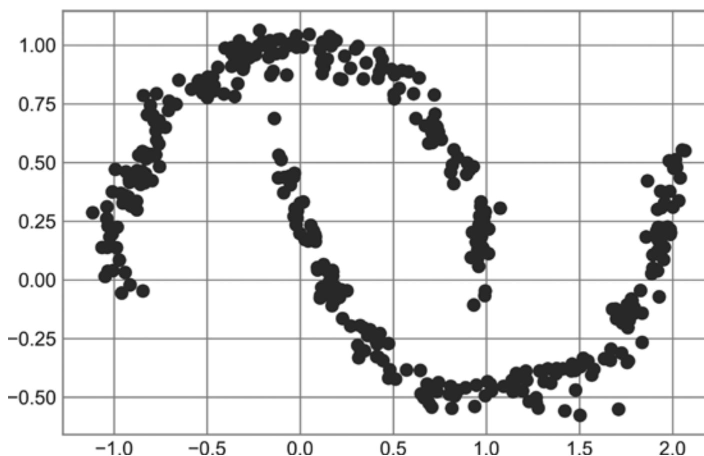


Рис. 48.12. Новые данные, полученные 16-компонентной моделью GMM

Метод GMM — удобное гибкое средство моделирования произвольного многомерного распределения данных.

Так как GMM — генеративная модель, у нас появляется естественная возможность определить оптимальное количество компонент для заданного набора данных. Генеративная модель, по существу, представляет собой распределение вероятности для набора данных, поэтому можно легко вычислить функцию *правдоподобия* (likelihood function) для лежащих в ее основе данных, используя перекрестную проверку, чтобы избежать переобучения. Другой способ введения поправки на переобучение — подстройка функции правдоподобия модели с помощью некоторого аналитического критерия, например информационного критерия Акаике (Akaike information criterion, AIC, см.: https://ru.wikipedia.org/wiki/Информационный_критерий_Акаике) или байесовского информационного критерия (bayesian information criterion, BIC, см.: https://ru.wikipedia.org/wiki/Информационный_критерий). Класс `GaussianMixture` из библиотеки Scikit-Learn имеет методы для вычисления этих критериев, что сильно упрощает указанный подход.

Посмотрим на критерии AIC и BIC как функции от количества компонент GMM для нашего набора данных moons (рис. 48.13):

```
In [17]: n_components = np.arange(1, 21)
         models = [GaussianMixture(n, covariance_type='full',
                                   random_state=0).fit(Xmoon)
                   for n in n_components]

         plt.plot(n_components, [m.bic(Xmoon) for m in models], label='BIC')
         plt.plot(n_components, [m.aic(Xmoon) for m in models], label='AIC')
         plt.legend(loc='best')
         plt.xlabel('n_components');
```

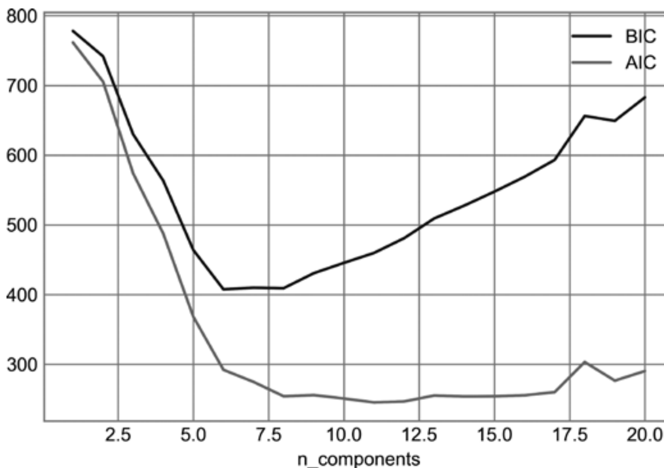


Рис. 48.13. Визуализация AIC и BIC с целью выбора количества компонент GMM

Оптимальным считается количество кластеров, минимизирующее AIC или BIC, в зависимости от требуемой аппроксимации. Согласно AIC наших 16 компонент, вероятно, слишком много, лучше взять 8–12. Как это обычно бывает в подобных задачах, критерий BIC говорит в пользу более простой модели.

Обратите внимание на важный момент: подобный метод выбора числа компонент представляет собой меру успешности GMM *как модели, оценивающей плотность распределения*, а не *как алгоритма кластеризации*. Я советовал бы вам рассматривать GMM в основном как инструмент оценки плотности и использовать его для кластеризации только заведомо простых наборов данных.

Пример: использование метода GMM для генерации новых данных

Мы увидели простой пример применения метода GMM в качестве генеративной модели данных с целью создания новых выборок на основе распределения, соответствующего исходным данным. В этом разделе мы продолжим воплощение этой идеи и сгенерируем *новые рукописные цифры* на основе корпуса стандартных цифр, который использовали ранее.

Для начала загрузим набор данных с изображениями цифр с помощью инструментов библиотеки Scikit-Learn:

```
In [18]: from sklearn.datasets import load_digits
         digits = load_digits()
         digits.data.shape
Out[18]: (1797, 64)
```

Далее выведем на экран первые 100 из них, чтобы вспомнить, с чем мы имеем дело (рис. 48.14):

```
In [19]: def plot_digits(data):
             fig, ax = plt.subplots(5, 10, figsize=(8, 4),
                                    subplot_kw=dict(xticks=[], yticks=[]))
             fig.subplots_adjust(hspace=0.05, wspace=0.05)
             for i, axi in enumerate(ax.flat):
                 im = axi.imshow(data[i].reshape(8, 8), cmap='binary')
                 im.set_clim(0, 16)
         plot_digits(digits.data)
```

Наш набор данных содержит почти 1800 цифр в 64 измерениях. Построим на их основе GMM, чтобы сгенерировать новые цифры. У смесей гауссовых распределений могут быть проблемы со сходимостью в пространстве столь высокой размерности, поэтому начнем с применения обратимого алгоритма для понижения размерности

данных. Воспользуемся для этой цели простым алгоритмом PCA с сохранением 99 % дисперсии в проекции данных:

```
In [20]: from sklearn.decomposition import PCA
         pca = PCA(0.99, whiten=True)
         data = pca.fit_transform(digits.data)
         data.shape
Out[20]: (1797, 41)
```

Рис. 48.14. Исходные рукописные цифры

Результат оказался 41-мерным, то есть размерность была снижена почти на 1/3 практически без потери информации. Воспользуемся критерием AIC и определим необходимое количество компонент GMM для спроецированных данных (рис. 48.15):

```
In [21]: n_components = np.arange(50, 210, 10)
         models = [GaussianMixture(n, covariance_type='full', random_state=0)
                   for n in n_components]
         aics = [model.fit(data).aic(data) for model in models]
         plt.plot(n_components, aics);
```

Похоже, что AIC имеет минимальное значение где-то в районе 140 компонент; этой моделью мы и воспользуемся. Обучим этот алгоритм на наших данных и убедимся, что он сошелся:

```
In [22]: gmm = GaussianMixture(140, covariance_type='full', random_state=0)
         gmm.fit(data)
         print(gmm.converged_)
Out[22]: True
```

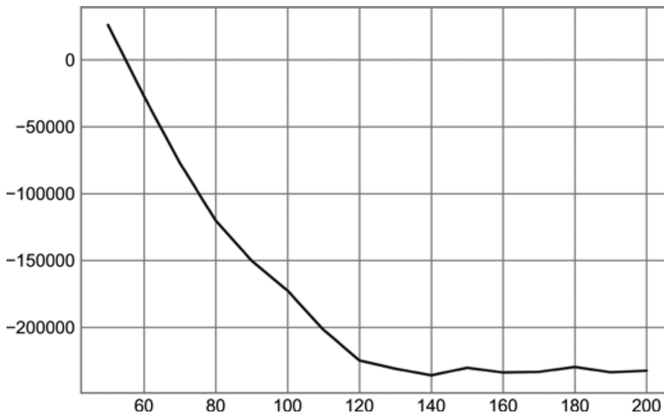


Рис. 48.15. Кривая AIC для выбора подходящего количества компонент GMM

Теперь можно сгенерировать 100 новых точек в этом 41-мерном пространстве, используя GMM как генеративную модель:

```
In [23]: data_new, label_new = gmm.sample(100)
         data_new.shape
Out[23]: (100, 41)
```

Наконец, выполним обратное преобразование объекта PCA для формирования новых цифр (рис. 48.16):

```
In [24]: digits_new = pca.inverse_transform(data_new)
         plot_digits(digits_new)
```

Рис. 48.16. Новые цифры, полученные случайным образом из модели GMM

Результаты по большей части выглядят очень похожими на цифры из набора данных!

Резюмируем сделанное: мы смоделировали распределение для заданной выборки рукописных цифр так, что на основе этих данных смогли сгенерировать совершенно новые цифры: это «рукописные цифры», не встречающиеся в исходном наборе данных, но отражающие общие признаки входных данных, смоделированные моделью смеси распределений. Подобная генеративная модель цифр может оказаться очень удобной как компонент байесовского генеративного классификатора, о котором я расскажу в следующей главе.

ГЛАВА 49

Заглянем глубже: ядерная оценка плотности распределения

В главе 48 мы рассмотрели смеси гауссовых распределений (GMM) — своеобразный гибрид моделей кластеризации и оценки плотности распределения. Напомню, что алгоритм оценки плотности выдает для D-мерного набора данных оценку D-мерного распределения вероятности, из которого получена эта выборка данных. Для этого алгоритм GMM представляет плотность распределения в виде взвешенной суммы гауссовых распределений. *Ядерная оценка плотности распределения* (kernel density estimation, KDE) — в некотором смысле алгоритм, доводящий идею смеси гауссовых функций до логического предела: в нем используется смесь, состоящая из одной гауссовой компоненты *для каждой точки*, что дает в результате непараметрическую модель плотности. В этой главе мы рассмотрим обоснование и сферу применения метода KDE.

Начнем с импортирования необходимых модулей:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Обоснование метода KDE: гистограммы

Как отмечалось выше, цель алгоритма оценки плотности — моделирование распределения вероятностей, на основе которого был сгенерирован набор данных. Вероятно, вы уже хорошо знакомы с одним из самых простых инструментов оценки плотности распределения одномерных данных — гистограммой. Гистограмма делит данные на дискретные интервалы значений, подсчитывает число точек, попадающих в каждый интервал, после чего отображает результат интуитивно понятным образом.

Для примера сгенерируем данные на основе двух нормальных распределений:

```
In [2]: def make_data(N, f=0.3, rseed=1):
            rand = np.random.RandomState(rseed)
            x = rand.randn(N)
            x[int(f * N):] += 5
            return x

        x = make_data(1000)
```

Такую обычную гистограмму, отражающую число точек, можно создать с помощью функции `plt.hist`. Задав параметр `density` гистограммы, мы получим нормализованную гистограмму, в которой высота интервалов отражает не число точек, а плотность вероятности (рис. 49.1).

```
In [3]: hist = plt.hist(x, bins=30, density=True)
```

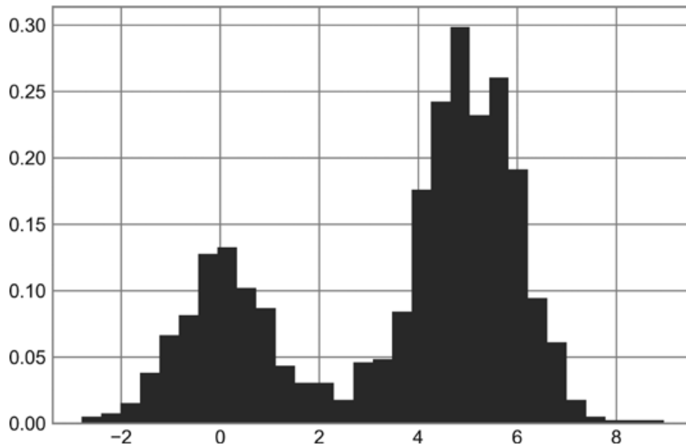


Рис. 49.1. Данные, полученные из сочетания нормальных распределений

Обратите внимание, что при равных интервалах такая нормализация просто меняет масштаб по оси *Y*, при этом относительная высота интервалов остается такой же, как и в гистограмме, отражающей число точек. Нормализация выбирается так, чтобы общая площадь под гистограммой была равна 1, в чем можно убедиться, глядя на вывод функции построения гистограммы:

```
In [4]: density, bins, patches = hist
        widths = bins[1:] - bins[:-1]
        (density * widths).sum()
Out[4]: 1.0
```

Одна из проблем использования гистограмм заключается в том, что конкретный выбор размера и расположения интервалов может привести к представлениям

с качественно разными признаками. Например, если посмотреть на версию этих данных из 20 точек, то конкретный выбор интервалов может привести к совершенно другой интерпретации данных! Рассмотрим следующий пример (рис. 49.2):

```
In [5]: x = make_data(20)
        bins = np.linspace(-5, 10, 10)

In [6]: fig, ax = plt.subplots(1, 2, figsize=(12, 4),
                               sharex=True, sharey=True,
                               subplot_kw={'xlim':(-4, 9),
                                           'ylim':(-0.02, 0.3)})
        fig.subplots_adjust(wspace=0.05)
        for i, offset in enumerate([0.0, 0.6]):
            ax[i].hist(x, bins=bins + offset, density=True)
            ax[i].plot(x, np.full_like(x, -0.01), '|k',
                       markeredgewidth=1)
```

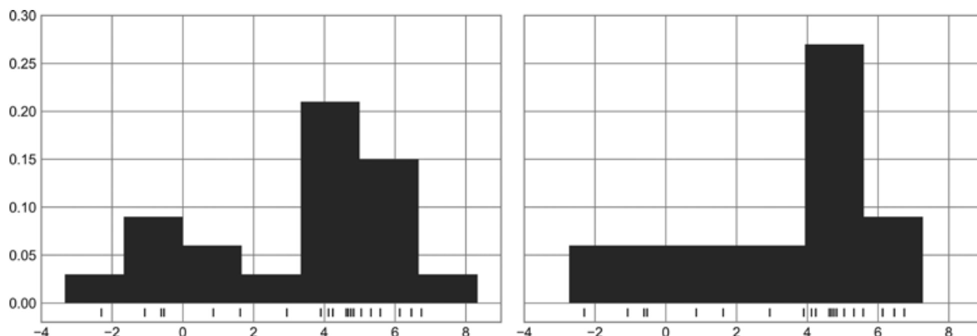


Рис. 49.2. Проблема гистограмм: различная интерпретация в зависимости от расположения интервалов

Из гистограммы слева очевидно, что мы имеем дело с бимодальным распределением. Справа же мы видим унимодальное распределение с длинным «хвостом». Без вышеприведенного кода вы бы вряд ли предположили, что эти две гистограммы были построены на одних данных. С учетом этого возникает вопрос: как можно доверять интуиции, полученной на основе подобных гистограмм? Что с этим можно сделать?

Небольшое отступление: гистограммы можно рассматривать как «стопки» блоков, где для каждой точки набора данных в соответствующий интервал помещается один блок. Посмотрим на это непосредственно (рис. 49.3):

```
In [7]: fig, ax = plt.subplots()
        bins = np.arange(-3, 8)
        ax.plot(x, np.full_like(x, -0.1), '|k',
                markeredgewidth=1)
```

```
        for count, edge in zip(*np.histogram(x, bins)):
            for i in range(count):
                ax.add_patch(plt.Rectangle(
                    (edge, i), 1, 1, ec='black', alpha=0.5))
        ax.set_xlim(-4, 8)
        ax.set_ylim(-0.2, 8)
Out[7]: (-0.2, 8.0)
```

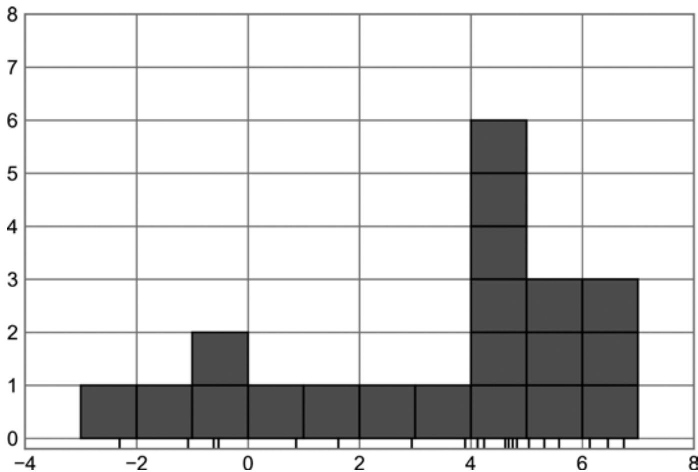


Рис. 49.3. Гистограмма как «стопки» блоков

В основе проблемы с нашими двумя разбиениями по интервалам лежит тот факт, что высота «стопки» блоков часто отражает не фактическую плотность близлежащих точек, а случайные стечения обстоятельств, выражающиеся в выравнивании интервалов по точкам данных. Это рассогласование точек и их блоков может приводить к наблюдаемым здесь неудовлетворительным результатам гистограмм. Но что, если вместо складывания в «стопки» блоков, выровненных по *интервалам*, мы складывали бы блоки, выровненные по *точкам, которым они соответствуют*? В этом случае блоки не были бы выровненными, но получить требуемый результат можно было бы, сложив их вклад в значение в каждом месте на оси X. Давайте сделаем это (рис. 49.4):

```
In [8]: x_d = np.linspace(-4, 8, 2000)
        density = sum((abs(xi - x_d) < 0.5) for xi in x)

        plt.fill_between(x_d, density, alpha=0.5)
        plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

        plt.axis([-4, 8, -0.2, 8]);
```

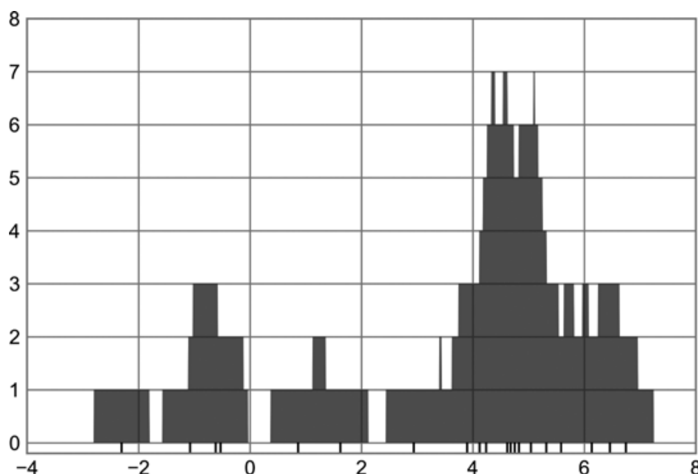


Рис. 49.4. «Гистограмма» с центрированием блоков по отдельным точкам — пример ядерной оценки плотности распределения

Результат выглядит немного неряшливо, но отражает подлинные характеристики данных гораздо надежнее, чем стандартная гистограмма. Тем не менее неровные края не слишком приятны для глаз и не отражают никаких фактических свойств данных. Для их сглаживания можно попытаться заменить блоки в каждой точке гладкой функцией, например гауссовой. Используем вместо блока в каждой точке стандартную нормальную кривую (рис. 49.5):

```
In [9]: from scipy.stats import norm
        x_d = np.linspace(-4, 8, 1000)
        density = sum(norm(xi).pdf(x_d) for xi in x)

        plt.fill_between(x_d, density, alpha=0.5)
        plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

        plt.axis([-4, 8, -0.2, 5]);
```

Этот сглаженный график со вкладом гауссового распределения в местах, соответствующих всем исходным точкам, обеспечивает намного более точное представление о форме распределения данных, причем с намного меньшей дисперсией, то есть отличия образцов приводят к минимальным его изменениям.

Два последних графика представляют примеры одномерной ядерной оценки плотности распределения: в первом используется так называемое ядро типа «цилиндр», а во втором — гауссово ядро. Рассмотрим ядерную оценку плотности распределения более подробно.

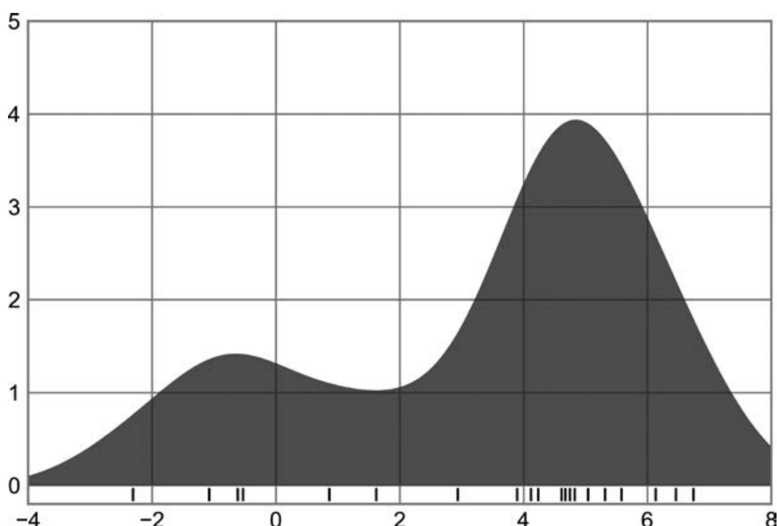


Рис. 49.5. Ядерная оценка плотности распределения с гауссовым ядром

Ядерная оценка плотности распределения на практике

Свободными параметрами ядерной оценки плотности распределения являются *ядро* (kernel), задающее форму распределения в каждой точке, и *ширина ядра* (kernel bandwidth), обусловливающая размер ядра в каждой точке. На практике для ядерной оценки плотности распределения существует множество различных ядер: в частности, реализация KDE в библиотеке Scikit-Learn поддерживает использование одного из шести ядер, о которых вы можете прочитать в документации библиотеки Scikit-Learn (https://oreil.ly/2Ae4a).

Хотя в языке Python реализовано несколько вариантов ядерной оценки плотности (особенно в пакетах SciPy и `statsmodels`), я предпочитаю использовать вариант из Scikit-Learn по причине гибкости и эффективности. Он реализован в классе `sklearn.neighbors.KernelDensity`, умеющем работать с KDE в многомерном пространстве с одним из шести ядер и одной из нескольких дюжин метрик. В силу того что метод KDE может потребовать значительных вычислительных затрат, этот класс использует «под капотом» алгоритм на основе деревьев и умеет достигать компромисса между временем вычислений и точностью с помощью параметров `atol` (absolute tolerance, допустимая абсолютная погрешность) и `rtol` (relative tolerance, допустимая относительная погрешность). Определить ширину ядра — свободный параметр — можно стандартными инструментами перекрестной проверки из библиотеки Scikit-Learn.

Рассмотрим простой пример воспроизведения предыдущего графика с помощью класса `KernelDensity` из библиотеки Scikit-Learn (рис. 49.6):

```
In [10]: from sklearn.neighbors import KernelDensity

         # Создание экземпляра модели KDE и ее обучение
         kde = KernelDensity(bandwidth=1.0, kernel='gaussian')
         kde.fit(x[:, None])

         # score_samples возвращает логарифм плотности
         # распределения вероятности
         logprob = kde.score_samples(x_d[:, None])

         plt.fill_between(x_d, np.exp(logprob), alpha=0.5)
         plt.plot(x, np.full_like(x, -0.01), '|k', markeredgewidth=1)
         plt.ylim(-0.02, 0.22);
```

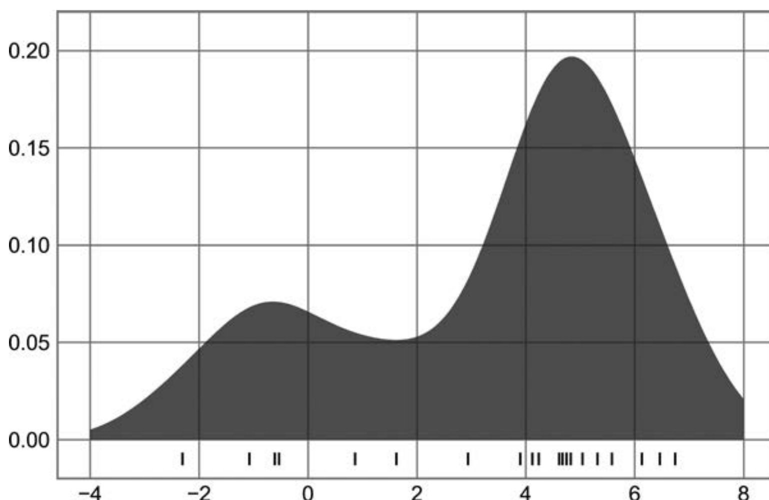


Рис. 49.6. Ядерная оценка плотности, вычисленная с помощью библиотеки Scikit-Learn

Результат нормализован так, что площадь под кривой равна 1.

Выбор ширины ядра путем перекрестной проверки

Выбор ширины ядра в методе KDE исключительно важен для получения удовлетворительной оценки плотности. Это и есть тот параметр, который при оценке плотности служит для выбора компромисса между систематической ошибкой и дисперсией. Слишком маленькая ширина ядра приводит к оценке с высокой дисперсией, то есть переобучению, при которой наличие или отсутствие одной-единственной точки может серьезно повлиять на модель. Слишком же широкое ядро

ведет к оценке со значительной систематической ошибкой, то есть недообучению, когда структура данных размывается этим широким ядром.

В статистике существует долгая предыстория методов быстрой оценки оптимальной ширины ядра на основе довольно строгих допущений относительно данных: если заглянуть в реализации метода KDE в пакетах SciPy и `statsmodels`, например, то можно увидеть реализации, основанные на некоторых из этих правил.

В контексте машинного обучения мы уже видели, что выбор подобных гиперпараметров зачастую проводится эмпирически посредством перекрестной проверки. Учитывая это, класс `KernelDensity` из библиотеки Scikit-Learn спроектирован в расчете на непосредственное использование в стандартных инструментах Scikit-Learn для поиска по сетке. В данном случае мы воспользуемся классом `GridSearchCV`, чтобы выбрать оптимальную ширину ядра для предыдущего набора данных. Поскольку наш набор данных очень невелик, используем перекрестную проверку по отдельным объектам, при которой размер обучающей последовательности максимален для каждого испытания перекрестной проверки:

```
In [11]: from sklearn.model_selection import GridSearchCV
         from sklearn.model_selection import LeaveOneOut

         bandwidths = 10 ** np.linspace(-1, 1, 100)
         grid = GridSearchCV(KernelDensity(kernel='gaussian'),
                             {'bandwidth': bandwidths},
                             cv=LeaveOneOut())
         grid.fit(x[:, None]);
```

Теперь можно узнать ширину ядра, максимизирующую оценку эффективности модели (которая по умолчанию является логарифмической функцией правдоподобия):

```
In [12]: grid.best_params_
Out[12]: {'bandwidth': 1.1233240329780276}
```

Оптимальная ширина ядра оказалась очень близка к той, которую мы использовали в примере выше, где ширина была равно 1,0 (это ширина ядра по умолчанию объекта `scipy.stats.norm`).

Пример: не столь наивный байес

В этом примере рассматривается байесовская генеративная классификация с KDE и демонстрируется создание пользовательской модели на основе архитектуры библиотеки Scikit-Learn.

В главе 41 мы рассмотрели наивную байесовскую классификацию, в которой создали простые генеративные модели для всех классов и построили на их основе

быстрый классификатор. В случае наивного байесовского классификатора генеративная модель — это просто выровненная по осям координат гауссова функция. Алгоритм оценки плотности, например KDE, позволяет убрать «наивную» составляющую и произвести ту же самую классификацию с более сложными генеративными моделями для каждого из классов. Эта классификация остается байесовской, но уже не будет «наивной».

Общая методика генеративной классификации такова.

1. Разбиение обучающих данных по меткам.
2. Для каждого набора находится генеративная модель путем обучения KDE. Это дает возможность вычислить функцию правдоподобия $P(x|y)$ для каждого наблюдения x и метки y.
3. На основе количества экземпляров каждого класса в обучающей последовательности вычисляется *априорная вероятность принадлежности к классу* (class prior), $P(y)$.
4. Для неизвестной точки x вычисляется апостериорная вероятность принадлежности к классу $P(y|x) \propto P(x|y)\, P(y)$. Метка каждой точки — класс, при котором достигается максимум апостериорной вероятности.

Данный алгоритм достаточно прост и интуитивно понятен. Несколько сложнее реализовать его с помощью фреймворка Scikit-Learn так, чтобы воспользоваться поиском по сетке и перекрестной проверкой.

Вот код, реализующий этот алгоритм на базе фреймворка Scikit-Learn, мы последовательно рассмотрим его блок за блоком:

```
In [13]: from sklearn.base import BaseEstimator, ClassifierMixin

         class KDEClassifier(BaseEstimator, ClassifierMixin):
             """Байесовская генеративная классификация на основе метода KDE

             Параметры
             ----------
             bandwidth : float
                 Ширина ядра в каждом классе
             kernel : str
                 Название ядра, передаваемое в KernelDensity
             """
             def __init__(self, bandwidth=1.0, kernel='gaussian'):
                 self.bandwidth = bandwidth
                 self.kernel = kernel

             def fit(self, X, y):
                 self.classes_ = np.sort(np.unique(y))
                 training_sets = [X[y == yi] for yi in self.classes_]
                 self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                               kernel=self.kernel).fit(Xi)
                                 for Xi in training_sets]
```

```
        self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                           for Xi in training_sets]
        return self

    def predict_proba(self, X):
        logprobs = np.array([model.score_samples(X)
                             for model in self.models_]).T
        result = np.exp(logprobs + self.logpriors_)
        return result / result.sum(axis=1, keepdims=True)

    def predict(self, X):
        return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Внутреннее устройство пользовательской модели

Рассмотрим этот код и обсудим основные его особенности:

```
from sklearn.base import BaseEstimator, ClassifierMixin

class KDEClassifier(BaseEstimator, ClassifierMixin):
    """Байесовская порождающая классификация на основе метода KDE

    Параметры
    ----------
    bandwidth : float
        the Ширина ядра в каждом классе
    kernel : str
        Название ядра, передаваемое в KernelDensity
    """
```

Каждая модель в библиотеке Scikit-Learn реализована как класс, наследующий `BaseEstimator`, а также подмешивает (mixin) класс со стандартной функциональностью. Например, помимо прочего, класс `BaseEstimator` включает логику, необходимую для клонирования/копирования модели, чтобы использовать ее в процедуре перекрестной проверки, а `ClassifierMixin` определяет используемый по умолчанию метод `score`. Мы также задали строку docstring, которую будет извлекать справочная система языка Python (см. главу 1).

Далее следует метод инициализации нашего класса:

```
def __init__(self, bandwidth=1.0, kernel='gaussian'):
    self.bandwidth = bandwidth
    self.kernel = kernel
```

Это тот код, который фактически выполняется при создании объекта вызовом конструктора `KDEClassifier`. В библиотеке Scikit-Learn важно, чтобы *в методе инициализации не выполнялось никаких операций*, кроме присваивания объекту `self` переданных значений. Причина в том, что содержащаяся в классе `BaseEstimator` логика необходима для клонирования и модификации моделей для перекрестной

проверки, поиска по сетке и других целей. Аналогично все аргументы метода __init__ должны быть объявлены, то есть следует избегать аргументов *args или **kwargs, так как они не могут быть корректно обработаны внутри процедур перекрестной проверки.

Далее следует метод fit, обрабатывающий обучающие данные:

```
def fit(self, X, y):
    self.classes_ = np.sort(np.unique(y))
    training_sets = [X[y == yi] for yi in self.classes_]
    self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                  kernel=self.kernel).fit(Xi)
                    for Xi in training_sets]
    self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                       for Xi in training_sets]
    return self
```

Он отыскивает в обучающих данных уникальные классы, обучает модель KernelDensity на каждом из них и вычисляет априорные вероятности на основе количеств исходных образцов. Наконец, метод fit должен всегда возвращать объект self, чтобы можно было связывать команды в цепочку. Например:

```
label = model.fit(X, y).predict(X)
```

Обратите внимание, что все хранимые результаты обучения сохраняются в свойствах с именами, завершающимися символом подчеркивания (например, self.logpriors_). Это соглашение используется в библиотеке Scikit-Learn, чтобы можно было быстро просмотреть список свойств модели (с помощью TAB-автодополнения в оболочке IPython) и выяснить, какие именно были обучены на тренировочных данных.

Наконец, у нас имеется логика для предсказания меток новых данных:

```
def predict_proba(self, X):
    logprobs = np.vstack([model.score_samples(X)
                          for model in self.models_]).T
    result = np.exp(logprobs + self.logpriors_)
    return result / result.sum(axis=1, keepdims=True)

def predict(self, X):
    return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Мы имеем дело с вероятностным классификатором, поэтому сначала реализовали метод predict_proba, возвращающий массив с формой [n_samples, n_classes] вероятностей классов. Элемент [i, j] этого массива представляет собой апостериорную вероятность, что образец i — член класса j, которая вычисляется как нормализованное произведение функции правдоподобия на априорную вероятность.

Наконец, эти вероятности используются в методе predict, который возвращает класс с максимальной вероятностью.

Использование пользовательской модели

Воспользуемся этой пользовательской моделью для решения задачи классификации рукописных цифр. Загрузим цифры и вычислим оценку эффективности модели для ядер разной ширины с помощью метамодели `GridSearchCV` (см. более подробную информацию по этому вопросу в главе 39):

```
In [14]: from sklearn.datasets import load_digits
         from sklearn.model_selection import GridSearchCV

         digits = load_digits()

         grid = GridSearchCV(KDEClassifier(),
                             {'bandwidth': np.logspace(0, 2, 100)})
         grid.fit(digits.data, digits.target);
```

Далее построим график зависимости оценок, полученных при перекрестной проверке эффективности модели, от ширины ядра (рис. 49.7):

```
In [15]: fig, ax = plt.subplots()
         ax.semilogx(np.array(grid.cv_results_['param_bandwidth']),
                     grid.cv_results_['mean_test_score'])
         ax.set(title='KDE Model Performance', ylim=(0, 1),
                xlabel='bandwidth', ylabel='accuracy')
         print(f'best param: {grid.best_params_}')
         print(f'accuracy = {grid.best_score_}')
Out[15]: best param: {'bandwidth': 6.135907273413174}
         accuracy = 0.9677298050139276
```

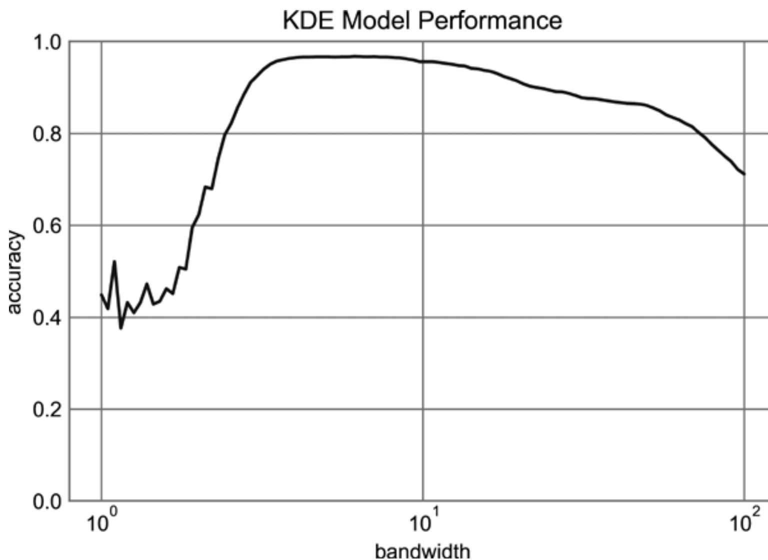


Рис. 49.7. Кривая оценки эффективности байесовского классификатора, основанного на KDE

Как видите, этот «не столь наивный» байесовский классификатор при перекрестной проверке достигает точности более чем 96 %. И это по сравнению с примерно 80 % у «наивного» байесовского классификатора:

```
In [16]: from sklearn.naive_bayes import GaussianNB
         from sklearn.model_selection import cross_val_score
         cross_val_score(GaussianNB(), digits.data, digits.target).mean()
Out[16]: 0.8069281956050759
```

Одно из преимуществ подобного генеративного классификатора — удобство интерпретации результатов: мы получаем для каждой неизвестной выборки не только вероятностную классификацию, но и *полную модель* распределения точек, с которыми мы ее сравниваем! При необходимости это позволяет пролить свет на причины того, почему конкретная классификация именно такова, причины, которые такие алгоритмы, как SVM и случайные леса, скрывают.

Чтобы достичь еще большего, можно внести в модель классификатора KDE некоторые усовершенствования:

- допустить независимое изменение ширины ядра для каждого класса;
- оптимизировать ширину ядер не на основе оценки точности предсказания, а на основе функции правдоподобия для обучающих данных при генеративной модели для каждого класса, то есть использовать оценки эффективности непосредственно из `KernelDensity`, а не общую оценку точности предсказания.

И наконец, если вы хотите приобрести опыт создания собственных моделей, попробуйте составить аналогичный байесовский классификатор с использованием смесей гауссовых распределений вместо KDE.

ГЛАВА 50

Прикладная задача: конвейер распознавания лиц

В этой части книги мы рассмотрели несколько основных идей и алгоритмов машинного обучения. Но перейти от теоретических идей к настоящим прикладным задачам может оказаться непростым делом. Реальные наборы данных часто бывают зашумлены и неоднородны, в них могут отсутствовать признаки, они могут содержать данные в таком виде, который сложно преобразовать в аккуратную матрицу `[n_samples, n_features]`. Вам придется, прежде чем воспользоваться любым из изложенных здесь методов, сначала извлечь эти признаки из данных. Не существует универсального рецепта, подходящего для всех предметных областей. В этом вопросе вам как исследователю данных придется использовать собственные интуицию и накопленный опыт.

Одно из очень интересных приложений машинного обучения — анализ изображений, и мы уже видели несколько его примеров с использованием пиксельных признаков для классификации. На практике данные редко оказываются настолько однородными, и простых пикселов часто бывает недостаточно. Это привело к появлению обширной литературы, посвященной методам *выделения признаков* (feature extraction) для изображений (см. главу 40).

В этой главе мы рассмотрим одну из подобных методик выделения признаков — гистограмму направленных градиентов (histogram of oriented gradients, HOG, см. https://ru.wikipedia.org/wiki/Гистограмма_направленных_градиентов), которая преобразует пикселы изображения в векторное представление, чувствительное к несущим информацию признакам изображения, без учета таких факторов, как освещенность. Мы воспользуемся этими признаками для разработки простого конвейера распознавания лиц, используя алгоритмы и идеи машинного обучения, которые обсуждали ранее в этой части книги.

Начнем с импортирования необходимых модулей:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        import numpy as np
```

Признаки HOG

Гистограмма направленных градиентов (HOG) — простая процедура выделения признаков, разработанная для идентификации пешеходов на изображениях. Метод HOG включает следующие этапы.

1. Необязательная предварительная нормализация изображений. В результате получаются признаки, слабо зависящие от изменений освещенности.
2. Операция свертывания изображения с помощью двух фильтров, чувствительных к горизонтальным и вертикальным градиентам яркости. Это позволяет уловить информацию о границах, контурах и текстурах изображения.
3. Разбивка изображения на ячейки заранее определенного размера и вычисление гистограммы направлений градиентов в каждой из ячеек.
4. Нормализация гистограмм в каждой из ячеек путем сравнения с несколькими близлежащими ячейками. Это еще больше подавляет влияние освещенности на изображение.
5. Формирование одномерного вектора признаков из информации по каждой ячейке.

В проект Scikit-Image встроена процедура выделения признаков на основе HOG, которую мы сможем довольно быстро применить на практике и визуализировать направленные градиенты во всех ячейках (рис. 50.1):

```
In [2]: from skimage import data, color, feature
        import skimage.data

        image = color.rgb2gray(data.chelsea())
        hog_vec, hog_vis = feature.hog(image, visualize=True)

        fig, ax = plt.subplots(1, 2, figsize=(12, 6),
                               subplot_kw=dict(xticks=[], yticks=[]))
        ax[0].imshow(image, cmap='gray')
        ax[0].set_title('input image')

        ax[1].imshow(hog_vis)
        ax[1].set_title('visualization of HOG features');
```

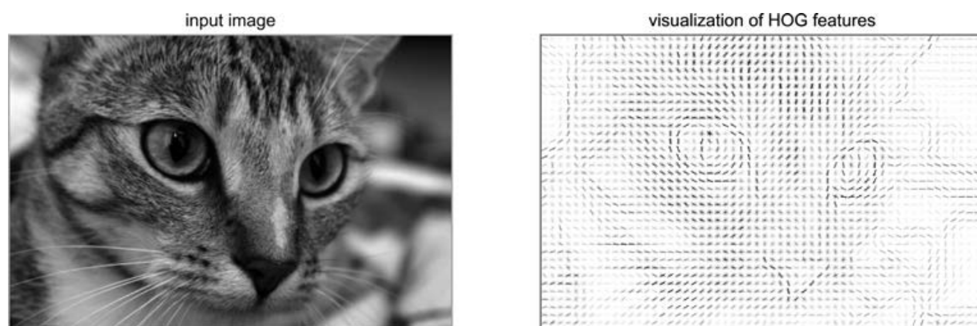


Рис. 50.1. Визуализация HOG-признаков, вычисленных для изображения

Метод HOG в действии: простой детектор лиц

На основе этих признаков HOG можно создать простой алгоритм обнаружения лиц с помощью любой из моделей в библиотеке Scikit-Learn. Мы воспользуемся линейным методом опорных векторов (см. главу 43). Алгоритм включает следующие шаги.

1. Получение миниатюр изображений, на которых присутствуют лица, для формирования набора «положительных» обучающих образцов.
2. Получение миниатюр изображений, на которых отсутствуют лица, для формирования набора «отрицательных» обучающих образцов.
3. Выделение HOG-признаков из этих обучающих образцов.
4. Обучение линейного SVM-классификатора на этих образцах.
5. В случае «незнакомого» изображения перемещаем по изображению скользящее окно, применяя модель, чтобы выяснить, присутствует ли в этом окне лицо или нет.
6. Если обнаруженные лица частично пересекаются, объединяем их в одно окно.

Пройдемся по этим шагам подробнее.

1. Получаем набор положительных обучающих образцов

Найдем положительные обучающие образцы с разнообразными лицами. У нас уже есть подходящий набор данных Labeled Faces in the Wild (LFW), который можно скачать с помощью библиотеки Scikit-Learn:

```
In [3]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people()
```

```
        positive_patches = faces.images
        positive_patches.shape
Out[3]: (13233, 62, 47)
```

Мы получили пригодную для обучения выборку с 13 000 изображений лиц.

2. Получаем набор отрицательных обучающих образцов

Далее необходимо найти набор миниатюр такого же размера, на которых отсутствуют лица. Для этой цели можно взять любой корпус изображений и извлечь из них миниатюры в различных масштабах. Воспользуемся некоторыми изображениями, поставляемыми вместе с пакетом Scikit-Image, а также классом `PatchExtractor` из библиотеки Scikit-Learn:

```
In [4]: data.camera().shape
Out[4]: (512, 512)
In [5]: from skimage import data, transform

        imgs_to_use = ['camera', 'text', 'coins', 'moon',
                       'page', 'clock', 'immunohistochemistry',
                       'chelsea', 'coffee', 'hubble_deep_field']
        raw_images = (getattr(data, name)() for name in imgs_to_use)
        images = [color.rgb2gray(image) if image.ndim == 3 else image
                  for image in raw_images]

In [6]: from sklearn.feature_extraction.image import PatchExtractor

        def extract_patches(img, N, scale=1.0, patch_size=positive_patches[0].shape):
            extracted_patch_size = tuple((scale * np.array(patch_size)).astype(int))
            extractor = PatchExtractor(patch_size=extracted_patch_size,
                                       max_patches=N, random_state=0)
            patches = extractor.transform(img[np.newaxis])
            if scale != 1:
                patches = np.array([transform.resize(patch, patch_size)
                                    for patch in patches])
            return patches

        negative_patches = np.vstack([extract_patches(im, 1000, scale)
                                      for im in images for scale in [0.5, 1.0, 2.0]])
        negative_patches.shape
Out[6]: (30000, 62, 47)
```

Теперь у нас есть 30 000 подходящих фрагментов изображений, не содержащих лиц. Рассмотрим некоторые из них, чтобы лучше представить, как они выглядят (рис. 50.2):

```
In [7]: fig, ax = plt.subplots(6, 10)
        for i, axi in enumerate(ax.flat):
            axi.imshow(negative_patches[500 * i], cmap='gray')
            axi.axis('off')
```

Надеемся, что они достаточно хорошо охватывают пространство «не лиц», которые могут встретиться нашему алгоритму.

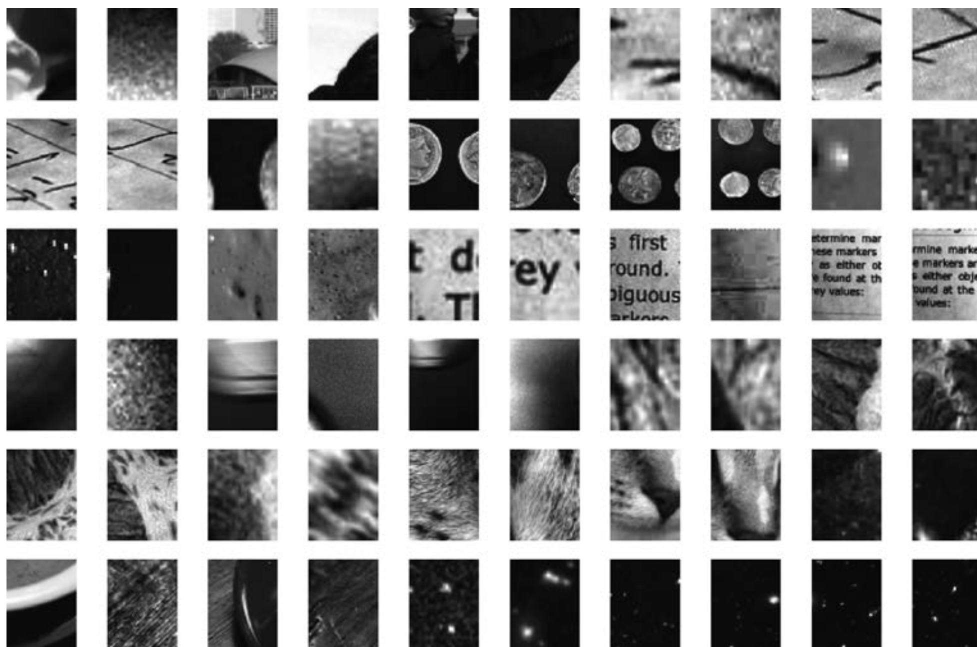


Рис. 50.2. Отрицательные фрагменты изображений, не содержащие лиц

3. Объединяем наборы и выделяем HOG-признаки

Получив положительные и отрицательные образцы, мы можем их объединить и вычислить HOG-признаки. Этот шаг займет некоторое время, поскольку для обработки каждого изображения требуется выполнить сложные вычисления.

```
In [8]: from itertools import chain
        X_train = np.array([feature.hog(im)
                            for im in chain(positive_patches,
                                            negative_patches)])
        y_train = np.zeros(X_train.shape[0])
        y_train[:positive_patches.shape[0]] = 1

In [9]: X_train.shape
Out[9]: (43233, 1215)
```

Итак, мы получили 43 000 обучающих образцов в 1215-мерном пространстве, и наши данные находятся в подходящем для библиотеки Scikit-Learn виде!

4. Обучаем метод опорных векторов

Воспользуемся изученными ранее инструментами для создания классификатора фрагментов миниатюр. Линейный метод опорных векторов — хороший выбор для задачи бинарной классификации в случае столь высокой размерности. Воспользуемся классификатором LinearSVC, поскольку он обычно лучше масштабируется с ростом числа образцов по сравнению с SVC.

Но сначала воспользуемся простым гауссовым наивным байесовским классификатором, чтобы было с чем сравнивать:

```
In [10]: from sklearn.naive_bayes import GaussianNB
         from sklearn.model_selection import cross_val_score

         cross_val_score(GaussianNB(), X_train, y_train)
Out[10]: array([0.94795883, 0.97143518, 0.97224471, 0.97501735, 0.97374508])
```

Как видите, на наших данных даже наивный байесовский алгоритм достигает более чем 95%-ной точности. Попробуем теперь метод опорных векторов с поиском по сетке из нескольких вариантов параметра C:

```
In [11]: from sklearn.svm import LinearSVC
         from sklearn.model_selection import GridSearchCV
         grid = GridSearchCV(LinearSVC(), {'C': [1.0, 2.0, 4.0, 8.0]})
         grid.fit(X_train, y_train)
         grid.best_score_
Out[11]: 0.9885272620319941

In [12]: grid.best_params_
Out[12]: {'C': 1.0}
```

Этот метод дает нам 99%-ную точность. Обучим эту более оптимальную модель на полном наборе данных:

```
In [13]: model = grid.best_estimator_
         model.fit(X_train, y_train)
Out[13]: LinearSVC()
```

5. Выполняем поиск лиц в новом изображении

Теперь, получив обученную модель, возьмем новое изображение и посмотрим, насколько хорошо она справится с ним. Воспользуемся для простоты одним из изображений астронавтов (см. обсуждение этого вопроса в следующем разделе), перемещая по нему скользящее окно и оценивая каждый фрагмент (рис. 50.3):

```
In [14]: test_image = skimage.data.astronaut()
         test_image = skimage.color.rgb2gray(test_image)
```

```
        test_image = skimage.transform.rescale(test_image, 0.5)
        test_image = test_image[:160, 40:180]

        plt.imshow(test_image, cmap='gray')
        plt.axis('off');
```

Рис. 50.3. Изображение, в котором мы попытаемся найти лицо

Далее создадим окно, которое будет перемещаться по этому изображению, и вычислим HOG-признаки для каждого фрагмента:

```
In [15]: def sliding_window(img, patch_size=positive_patches[0].shape,
                            istep=2, jstep=2, scale=1.0):
             Ni, Nj = (int(scale * s) for s in patch_size)
             for i in range(0, img.shape[0] - Ni, istep):
                 for j in range(0, img.shape[1] - Ni, jstep):
                     patch = img[i:i + Ni, j:j + Nj]
                     if scale != 1:
                         patch = transform.resize(patch, patch_size)
                     yield (i, j), patch

         indices, patches = zip(*sliding_window(test_image))
         patches_hog = np.array([feature.hog(patch) for patch in patches])
         patches_hog.shape
Out[15]: (1911, 1215)
```

Наконец, возьмем эти фрагменты, для которых вычислены признаки HOG, и воспользуемся нашей моделью, чтобы определить, содержат ли какие-то из них лицо:

```
In [16]: labels = model.predict(patches_hog)
         labels.sum()
Out[16]: 48.0
```

Как видите, среди 2000 найдено 48 фрагментов с лицом. Воспользуемся имеющейся информацией о фрагментах, чтобы определить, где в нашем контрольном изображении они располагаются, нарисовав их границы в виде прямоугольников (рис. 50.4):

```
In [17]: fig, ax = plt.subplots()
         ax.imshow(test_image, cmap='gray')
         ax.axis('off')
         Ni, Nj = positive_patches[0].shape
         indices = np.array(indices)

         for i, j in indices[labels == 1]:
             ax.add_patch(plt.Rectangle((j, i), Nj, Ni, edgecolor='red',
                                        alpha=0.3, lw=2, facecolor='none'))
```

Рис. 50.4. Окна, в которых были обнаружены лица

Все обнаруженные фрагменты перекрываются и содержат имеющееся на изображении лицо! Отличный результат для всего нескольких строк кода на языке Python.

Предостережения и дальнейшие усовершенствования

Если посмотреть на предшествующий код и примеры более внимательно, то можно обнаружить, что нужно сделать еще немало, прежде чем можно будет назвать наше приложение распознавания лиц готовым к промышленной эксплуатации. В нашем коде имеется несколько проблемных мест. Кроме того, в него не помешает внести несколько усовершенствований.

- *Наша обучающая последовательность, особенно в части отрицательных признаков, неполна.* Основная проблема заключается в наличии множества текстур, напоминающих лица, не включенных в нашу обучающую последовательность, поэтому текущая модель будет склонна выдавать ложноположительные результаты. Это будет заметно, если попытаться применить предыдущий алгоритм к полному изображению астронавта: текущая модель приведет ко множеству ложных обнаружений лиц в других областях изображения.

  Можно попытаться решить эту проблему, добавив в отрицательную обучающую последовательность множество разнообразных изображений, и это, вероятно, действительно приведет к некоторому улучшению. Другой способ — использование узконаправленного подхода, например *hard negative mining* (поиск сложных отрицательных образцов). При подходе hard negative mining берется новый, еще не виденный классификатором набор изображений и все фрагменты в нем, соответствующие ложноположительным результатам, явно добавляются в качестве отрицательных в обучающую последовательность до повторного обучения классификатора.

- *Текущий конвейер выполняет поиск только при одном значении масштаба.* В текущем виде наш алгоритм будет распознавать только лица с размером, примерно равным 62 × 47 пикселов. Эту проблему можно решить довольно просто, применяя скользящие окна различных размеров и изменяя размер каждого из фрагментов с помощью функции `skimage.transform.resize` до подачи его на вход модели. На самом деле используемая здесь вспомогательная функция `sliding_window` уже учитывает этот нюанс.

- *Желательно комбинировать перекрывающиеся фрагменты, на которых обнаружены лица.* В случае готового к промышленной эксплуатации конвейера получение 30 обнаружений одного и того же лица представляется нежелательным. Хотелось бы сократить перекрывающиеся группы обнаруженных лиц до одного. Это можно сделать с помощью одного из методов кластеризации без учителя (хороший кандидат на эту роль — кластеризация путем сдвига среднего значения (meanshift clustering)) или посредством процедурного подхода, например алгоритма *подавления немаксимумов* (non-maximum suppression), часто используемого в сфере машинного зрения.

- *Конвейер должен быть более продвинутым.* После решения вышеописанных проблем неплохо было бы создать более продвинутый конвейер, который получал бы на входе обучающие изображения и выдавал предсказания на основе скользящих окон. Именно в этом вопросе язык Python как инструмент науки о данных демонстрирует все свои возможности: приложив немного труда, мы сможем скомпоновать наш предварительный код с качественно спроектированным объектно-ориентированным API, обеспечивающим простоту использования. Оставлю это в качестве упражнения читателю.

- *Использовать последние достижения науки о данных, такие как глубокое обучение*. Наконец, мне хотелось бы добавить, что в контексте машинного обучения HOG и другие процедурные методы выделения признаков более не считаются современными. Вместо них многие современные конвейеры обнаружения объектов используют различные варианты глубоких нейронных сетей (*глубокое обучение*). Нейронные сети можно рассматривать как модели, определяющие оптимальную стратегию выделения признаков на основе самих данных и не полагающиеся на интуицию пользователя.

  Хотя в последние годы в этой области достигнуты фантастические результаты, глубокое обучение концептуально мало отличается от машинного, модели которого рассмотрены в предыдущих главах. Основным достижением является возможность использовать современное вычислительное оборудование (часто большие кластеры мощных машин) для обучения гораздо более гибких моделей на существенно бóльших объемах обучающих данных. И хотя масштабы разные, конечная цель по сути та же: построение моделей на основе данных.

На случай, если у вас появится желание двигаться дальше, в следующем разделе приводится список мест, откуда можно начать.

Дополнительные источники информации по машинному обучению

В этой части книги мы кратко рассмотрели машинное обучение в языке Python, в основном используя инструменты из библиотеки Scikit-Learn. Какими бы объемными ни были эти главы, в них все равно невозможно было охватить многие интересные и важные алгоритмы, подходы и вопросы. Я хотел бы предложить тем, кто желает узнать больше о машинном обучении, некоторые дополнительные источники информации.

- *Сайт библиотеки Scikit-Learn* (http://scikit-learn.org/). На сайте библиотеки Scikit-Learn содержатся поразительные объемы документации и примеров, охватывающие не только рассмотренные в книге модели, но и многое другое. Если вам необходим краткий обзор наиболее важных и часто используемых алгоритмов машинного обучения, то этот сайт станет для вас отличной отправной точкой.
- *Обучающие видео с конференций SciPy, PyCon и PyData*. Библиотека Scikit-Learn и другие вопросы машинного обучения — неизменные фавориты учебных пособий ежегодных конференций, посвященных языку Python, в частности PyCon, SciPy и PyData. Найти наиболее свежие материалы можно путем поиска в Интернете. Большинство этих конференций бесплатно публикуют в Сети

видео своих основных докладов и учебных пособий, вы с легкостью найдете их, сформулировав подходящий запрос в поисковой системе (например, «видео PyCon 2022»).

- Книга *Introduction to Machine Learning with Python* (O'Reilly; https://oreil.ly/kaQQs)[1], написанная Андреасом Мюллером и Сарой Гвидо. Она охватывает много вопросов машинного обучения, обсуждавшихся в этой части, и более подробно описывает различные возможности, реализованные в библиотеке Scikit-Learn, включая продвинутые модели, приемы проверки моделей и организацию конвейеров.
- Книга *Machine Learning with PyTorch and Scikit-Learn* (Pact; https://oreil.ly/p268i)[2] Себастьяна Рашки. Последняя книга Себастьяна Рашки начинается с некоторых фундаментальных тем, затронутых в этих главах, но идет дальше и показывает, как эти идеи применимы для реализации более сложных и ресурсоемких моделей глубокого обучения и обучения с подкреплением с использованием известной библиотеки PyTorch.

---

[1] *Гвидо С., Мюллер А.* Машинное обучение с помощью Python. Руководство для специалистов по работе с данными.

[2] *Рашка С., Мирджжалили В.* Python и машинное обучение.

Об авторе

Джейк Вандер Плас — инженер-программист в Google Research, работает над инструментами для исследований с привлечением больших объемов данных. Джейк создает и разрабатывает инструменты на Python для выполнения научных вычислений, в том числе такие пакеты, как Scikit-Learn, SciPy, AstroPy, Altair, JAX и многие другие. Участвует в жизни обширного сообщества специалистов по обработке данных, разрабатывая и представляя доклады и учебные пособия по темам научных вычислений на различных конференциях в мире науки о данных.

Иллюстрация на обложке

На обложке изображен мексиканский ядозуб *(Heloderma horridum)*, обитающий в Мексике и некоторых районах Гватемалы. Греческое слово *heloderma* переводится как «шипованная кожа». Такое название ящерица получила за характерную бугорчатую текстуру кожи. Эти бугорки представляют собой *остеодермы*, каждая из которых содержит небольшой кусочек костной ткани и служит защитной броней.

Мексиканский ядозуб имеет темный окрас с пятнами и полосами желтых тонов. У него широкая голова и толстый хвост, в котором накапливается жир, помогающий пережить жаркие летние месяцы, когда ящерица впадает в спячку. В среднем эти ящерицы имеют длину 50–90 см и весят около 800 г. Как и у большинства змей и ящериц, язык мексиканского ядозуба является основным органом чувств. Рептилия многократно выбрасывает его, чтобы собрать частицы запаха из окружающей среды и обнаружить добычу (или потенциального партнера во время брачного сезона).

Он и аризонский ядозуб (близкий родственник) — единственные ядовитые ящерицы в мире. При угрозе мексиканский ядозуб кусает и сжимает челюсти, выполняя жевательные движения, потому что не может сразу выпустить большое количество яда. Укус и воздействие яда чрезвычайно болезненны, хотя и редко смертельны для человека. Яд мексиканского ядозуба содержит ферменты, которые были синтезированы искусственно для лечения диабета, и в настоящее время ведутся дальнейшие фармакологические исследования. Этой рептилии угрожают браконьеры, торгующие экзотическими животными, и местные жители, убивающие ее из страха. Ядозуб охраняется законодательством обеих стран, где он обитает. Многие животные на обложках O'Reilly находятся под угрозой исчезновения; все они важны для мира.

Иллюстрацию для обложки нарисовала Карен Монтгомери на основе черно-белой гравюры из книги Вуда *Animate Creation*.

Джейк Плас вандер

Python для сложных задач: наука о данных

2-е международное издание

Перевела с английского Л. Киселева

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературные редакторы	*Н. Куликова, Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректор	*Т. Никифорова*
Верстка	*Г. Блинов*

Изготовлено в Казахстане. Изготовитель: ТОО «Спринт Бук».
Место нахождения и фактический адрес: 010000 Казахстан, г. Астана, район Алматы,
Проспект Ракымжан Кошкарбаев, дом 10/1, н. п. 18.

Дата изготовления: 12.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 21.11.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 47,730. Заказ 0000.

Отпечатано в ТОО «ФАРОС Графикс». 100004, РК, г. Караганда, ул. Молокова 106/2.