O'REILLY®

2-е издание

Машинное обучение с использованием Python

Сборник рецептов

Практические решения
от предобработки
до глубокого обучения

Кайл Галлатин,
Крис Элбон

alist

SECOND EDITION

Machine Learning with Python Cookbook

Practical Solutions from Preprocessing to Deep Learning

Kyle Gallatin and Chris Albon

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY

Кайл Галлатин,
Крис Элбон

Машинное обучение с использованием Python

Сборник рецептов

Практические решения
от предобработки
до глубокого обучения

Астана
«АЛИСТ»
2024

УДК 004.4'236
ББК 32.988.02-018
Г16

Галлатин К.

Г16 Машинное обучение с использованием Python. Сборник рецептов: Пер. с англ. / К. Галлатин, К. Элбон. — 2-е изд., перераб. и доп. — Астана: АЛИСТ, 2024. — 448 с.: ил.

ISBN 978-601-08-4119-2

Книга содержит около 200 задач машинного обучения, таких как загрузка и обработка текстовых или числовых данных, отбор модели и многие другие. Рассмотрена работа с языком Python, библиотеками pandas и scikit-learn. Коды примеров можно вставлять, объединять и адаптировать, создавая собственное приложение. Приведены рецепты решений с использованием: векторов, матриц и массивов; данных из CSV, JSON, SQL, баз данных, облачных хранилищ и других источников; обработки данных, текста, изображений, дат и времени; уменьшения размерности и методов выделения или отбора признаков; оценивания и отбора моделей; линейной и логистической регрессии, деревьев, лесов и k ближайших соседей; опорно-векторных машин (SVM), наивных байесовых классификаторов, кластеризации и нейронных сетей; сохранения и загрузки натренированных моделей.

Во втором издании все примеры обновлены, рассмотрены задачи и фреймворки глубокого обучения, расширены разделы с тензорами, нейронными сетями и библиотекой глубокого обучения PyTorch.

Для разработчиков систем машинного обучения

УДК 004.4'236
ББК 32.988.02-018

© 2024 ALIST LLP

Authorized Russian translation of the English edition of ***Machine Learning with Python Cookbook, 2nd edition*** ISBN 9781098135720 © 2023 Kyle Gallatin.
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.

Авторизованный перевод с английского языка на русский издания ***Machine Learning with Python Cookbook, 2nd edition*** ISBN 9781098135720 © 2023 Kyle Gallatin.
Перевод опубликован и продается с разрешения компании-правообладателя O'Reilly Media, Inc.

ISBN 978-1-098-13572-0 (англ.)
ISBN 978-601-08-4119-2 (каз.)

© Kyle Gallatin, 2023
© Издание на русском языке. ТОО "АЛИСТ", 2024

Содержание

Введение **13**

Условные обозначения 13
Использование примеров кода 14
Благодарности 14

ГЛАВА 1. Работа с векторами, матрицами и массивами в NumPy **17**

Введение 17
1.1. Создание вектора 17
1.2. Создание матрицы 18
1.3. Создание разреженной матрицы 19
1.4. Предварительное распределение массивов в NumPy 21
1.5. Обращение к элементу массива 22
1.6. Описание матрицы 24
1.7. Использование функций для каждого элемента 24
1.8. Поиск наибольших и наименьших значений 26
1.9. Вычисление среднего значения, дисперсии и стандартного отклонения 27
1.10. Изменение формы массива 28
1.11. Транспонирование вектора или матрицы 29
1.12. Преобразование матрицы в одномерный вектор 30
1.13. Определение ранга матрицы 31
1.14. Вывод диагонали матрицы 32
1.15. Вычисление суммы по диагонали матрицы 33
1.16. Вычисление скалярного произведения 34
1.17. Сложение и вычитание матриц 35
1.18. Умножение матриц 36
1.19. Создание обратной матрицы 37
1.20. Генерация псевдослучайных значений 38

ГЛАВА 2. Загрузка данных **40**

Введение 40
2.1. Загрузка готового набора данных 40

2.2. Создание искусственного набора данных ... 42
2.3. Загрузка CSV-файла ... 45
2.4. Загрузка файла Excel ... 47
2.5. Загрузка файла JSON ... 48
2.6. Загрузка файла Parquet ... 49
2.7. Загрузка файла Avro ... 50
2.8. Запрос к базе данных SQLite ... 51
2.9. Удаленное подключение к базе данных SQL ... 52
2.10. Загрузка данных из Google Sheet ... 53
2.11. Загрузка данных из S3 bucket ... 54
2.12. Загрузка неструктурированных данных ... 55

ГЛАВА 3. Первичная обработка данных ... 57

Введение ... 57
3.1. Создание DataFrame ... 58
3.2. Получение информации о данных ... 59
3.3. Срез DataFrame ... 61
3.4. Выбор строк на основе условных конструкций ... 64
3.5. Сортировка значений ... 65
3.6. Замена значений ... 66
3.7. Переименование столбцов ... 67
3.8. Нахождение наименьшего, наибольшего и среднего значения, суммы и количества значений ... 69
3.9. Поиск уникальных значений ... 70
3.10. Отбор отсутствующих значений ... 71
3.11. Удаление столбца ... 73
3.12. Удаление строки ... 75
3.13. Удаление дубликатов ... 76
3.14. Группировка строк по значениям ... 78
3.15. Группировка строк по времени ... 80
3.16. Агрегатные операции и статистика ... 82
3.17. Обход столбца в цикле ... 84
3.18. Применение функции ко всем элементам столбца ... 85
3.19. Применение функции к группам ... 86
3.20. Конкатенация структур данных DataFrame ... 87
3.21. Слияние структур данных DataFrame ... 88

ГЛАВА 4. Управление количественными данными ... 92

Введение ... 92
4.1. Масштабирование признака ... 92
4.2. Стандартизация признака ... 94
4.3. Нормализация наблюдаемых данных ... 96

4.4. Генерирование полиномиальных и взаимодействующих признаков 98
4.5. Преобразование признаков 100
4.6. Обнаружение выбросов 101
4.7. Управление выбросами 103
4.8. Дискретизация признаков 105
4.9. Группирование наблюдений при помощи кластеризации 107
4.10. Удаление элементов с отсутствующим значением 109
4.11. Восстановление пропущенных значений 111

ГЛАВА 5. Управление категориальными данными 114

Введение 114
5.1. Преобразование признаков с номинальными категориями 115
5.2. Преобразование признаков с порядковыми категориями 118
5.3. Преобразование словарей признаков 120
5.4. Восстановление классов пропущенных значений 122
5.5. Управление несбалансированными классами 124

ГЛАВА 6. Обработка текста 129

Введение 129
6.1. Очистка текста 129
6.2. Синтаксический анализ и очистка HTML-документов 132
6.3. Удаление знаков препинания 133
6.4. Токенизация текста 134
6.5. Удаление стоп-слов 135
6.6. Стемминг слов 136
6.7. Разметка частей речи 137
6.8. Распознавание именованных сущностей 139
6.9. Представление текста в виде мешка слов 141
6.10. Оценка значимости слов 143
6.11. Использование векторизации текстов для измерения соответствия текста поисковому запросу 145
6.12. Использование анализа тональности при классификации 147

ГЛАВА 7. Обработка данных даты и времени 149

Введение 149
7.1. Преобразование строк в даты 149
7.2. Работа с часовыми поясами 151
7.3. Выбор даты и времени 152
7.4. Разложение временных данных на отдельные признаки 153
7.5. Вычисление разницы между датами 154

7.6. Вычисление дней недели ... 155
7.7. Создание признаков с запаздыванием по времени ... 156
7.8. Использование метода скользящего окна ... 157
7.9. Обработка пропущенных значений во временных рядах ... 159

ГЛАВА 8. Работа с изображениями ... 162

Введение ... 162
8.1. Загрузка изображений ... 162
8.2. Сохранение изображений ... 165
8.3. Измерение размера изображений ... 166
8.4. Кадрирование изображений ... 167
8.5. Размытие изображений ... 168
8.6. Настройка резкости изображений ... 170
8.7. Увеличение контраста ... 171
8.8. Распознавание цветов ... 173
8.9. Бинаризация изображений ... 175
8.10. Удаление фона ... 177
8.11. Определение границ ... 180
8.12. Обнаружение углов ... 181
8.13. Создание признаков для машинного обучения ... 185
8.14. Создание гистограмм цветов в виде признаков ... 187
8.15. Использование предобученных эмбеддингов в качестве признаков ... 190
8.16. Распознавание объектов при помощи OpenCV ... 193
8.17. Классификация изображений при помощи Pytorch ... 194

ГЛАВА 9. Снижение размерности через выделение признаков ... 197

Введение ... 197
9.1. Сокращение признаков при помощи главных компонент ... 198
9.2. Сокращение признаков у линейно неразделимых данных ... 200
9.3. Сокращение признаков при помощи повышения разделимости классов ... 202
9.4. Сокращение признаков при помощи разложения матрицы ... 205
9.5. Сокращение признаков разреженных данных ... 207

ГЛАВА 10. Снижение размерности через отбор признаков ... 210

Введение ... 210
10.1. Пороговый отбор числовых признаков дисперсии ... 211
10.2. Пороговый отбор дисперсии бинарных признаков ... 212
10.3. Обработка признаков с высоким коэффициентом корреляции ... 214
10.4. Удаление нерелевантных для классификации признаков ... 215
10.5. Рекурсивное исключение признаков ... 218

ГЛАВА 11. Оценка модели ... 221

Введение ... 221
11.1. Модели на основе кросс-валидации ... 221
11.2. Создание базовой регрессионной модели ... 225
11.3. Создание базовой модели классификации ... 227
11.4. Оценка предсказаний бинарного классификатора ... 229
11.5. Оценка порогов бинарного классификатора ... 232
11.6. Оценка прогнозов многоклассового классификатора ... 236
11.7. Визуализация показателей качества классификации ... 237
11.8. Оценка регрессионных моделей ... 240
11.9. Оценки моделей кластеризации ... 242
11.10. Создание собственной оценочной метрики ... 244
11.11. Визуализация зависимости показателей от размера обучающего набора ... 245
11.12. Создание текстового отчета с метриками качества ... 248
11.13. Визуализация влияния значений гиперпараметров ... 249

ГЛАВА 12. Выбор модели ... 253

Введение ... 253
12.1. Поиск лучших моделей методом полного перебора ... 254
Обсуждение ... 255
12.2. Случайный поиск лучших моделей ... 256
12.3. Поиск лучшей модели среди множества алгоритмов обучения ... 258
12.4. Поиск лучшей модели в процессе первичной обработки ... 260
12.5. Ускорение поиска модели через распараллеливание ... 263
12.6. Ускорение поиска модели с помощью алгоритмических методов ... 264
12.7. Оценка качества после выбора модели ... 266

ГЛАВА 13. Линейная регрессия ... 269

Введение ... 269
13.1. Настройка линейной связи ... 269
13.2. Управление взаимосвязанными коэффициентами ... 271
13.3. Настройка нелинейной взаимосвязи ... 273
13.4. Уменьшение дисперсии через регуляризацию ... 276
13.5. Уменьшение дисперсии через лассо-регрессию ... 278

ГЛАВА 14. Деревья и леса ... 280

Введение ... 280
14.1. Обучение классификатора на основе деревьев решений ... 280
14.2. Обучение регрессора с помощью деревьев решений ... 282
14.3. Визуализация модели деревьев решений ... 284

14.4. Обучение классификатора на основе случайного леса 286
14.5. Обучение регрессора методом случайного леса 288
14.6. Оценка моделей случайного леса методом out-of-bag-ошибки 289
14.7. Определение ключевых признаков в моделях случайного леса 290
14.8. Выбор ключевых признаков в моделях случайного леса 292
14.9. Управление несбалансированными классами 294
14.10. Контроль размера дерева 295
14.11. Улучшение качества через бустинг 297
14.12. Обучение модели на основе алгоритма XGBoost 298
14.13. Улучшение производительности при помощи LightGBM 300

ГЛАВА 15. Метод *k* ближайших соседей 303

Введение 303
15.1. Определение ближайших соседей для наблюдения 303
15.2. Создание классификатора на основе *k* ближайших соседей 306
15.3. Определение оптимального количества ближайших соседей 308
15.4. Создание классификатора методом ближайших соседей в заданном радиусе 309
15.5. Поиск приближенных ближайших соседей 310
15.6. Оценка метода приближенных ближайших соседей 314

ГЛАВА 16. Логистическая регрессия 316

Введение 316
16.1. Обучение бинарного классификатора 316
16.2. Обучение многоклассового классификатора 318
16.3. Уменьшение дисперсии за счет регуляризации 319
16.4. Обучение классификатора на очень больших данных 320
16.5. Управление несбалансированными классами 322

ГЛАВА 17. Метод опорных векторов 324

Введение 324
17.1. Обучение линейного классификатора 324
17.2. Управление линейно неразделимыми классами через ядра свертки 327
17.3. Создание прогнозируемых вероятностей 331
17.4. Поиск опорных векторов 332
17.5. Управление несбалансированными классами 334

ГЛАВА 18. Наивный байесовский классификатор 336

Введение 336
18.1. Обучение классификатора для непрерывных признаков 337

18.2. Обучение классификатора для дискретных и количественных признаков 339
18.3. Обучение наивного байесовского классификатора для бинарных признаков 341
18.4. Калибровка прогнозируемых вероятностей 342

ГЛАВА 19. Кластеризация 344

Введение 344
19.1. Кластеризация методом *k* средних 344
19.2. Ускорение кластеризации методом *k* средних 347
19.3. Кластеризация со сдвигом среднего 348
19.4. Кластеризация DBSCAN 350
19.5. Иерархическая кластеризация 351

ГЛАВА 20. Тензоры в PyTorch 354

Введение 354
20.1. Создание тензора 354
20.2. Создание тензора из массива NumPy 355
20.3. Создание разреженного тензора 356
20.4. Выбор элементов тензора 357
20.5. Описание тензора 359
20.6. Операции над элементами 360
20.7. Поиск наибольших и наименьших значений 361
20.8. Изменение формы тензоров 362
20.9. Транспонирование тензора 363
20.10. Преобразование в одномерный тензор 364
20.11. Вычисление скалярного произведения 364
20.12. Умножение тензоров 365

ГЛАВА 21. Нейронные сети 367

Введение 367
21.1. Использование autograd в PyTorch 368
21.2. Предобработка данных для нейронных сетей 370
21.3. Разработка нейронной сети 372
21.4. Обучение бинарного классификатора 376
21.5. Обучение многоклассового классификатора 379
21.6. Обучение регрессора 382
21.7. Создание прогнозов 384
21.8. Визуализация истории обучения 387
21.9. Регуляризация весов для снижения переобучения 390
21.10. Снижение переобучения через раннюю остановку 392
21.11. Уменьшение эффекта переобучения методом исключения 396

21.12. Сохранение прогресса обучения модели ... 399
21.13. Настройка нейронных сетей ... 402
21.14. Визуализация нейронной сети ... 405

ГЛАВА 22. Нейронные сети с неструктурированными данными ... 409

Введение ... 409
22.1. Обучение нейронной сети для классификации изображений ... 410
22.2. Обучение нейронной сети для классификации текстов ... 413
22.3. Тонкая настройка предобученной модели для классификации изображений ... 415
22.4. Тонкая настройка предобученной модели для классификации текста ... 418

ГЛАВА 23. Сохранение, загрузка и развертывание обученных моделей ... 422

Введение ... 422
23.1. Сохранение и загрузка модели scikit-learn ... 422
23.2. Сохранение и загрузка модели TensorFlow ... 424
23.3. Сохранение и загрузка модели PyTorch ... 425
23.4. Развертывание моделей scikit-learn ... 428
23.5. Развертывание моделей TensorFlow ... 430
23.6. Развертывание моделей PyTorch через Seldon ... 433

Предметный указатель ... 438

Об авторах ... 446

Об изображении на обложке ... 447

Введение

Первое издание данной книги вышло в 2018 году и заняло особое место среди работ, посвященных машинному обучению (machine learning, ML). Разработчики получили набор проверенных и готовых к практическому применению решений для Python, которые можно легко добавить и адаптировать под различные варианты использования. На протяжении последних 5 лет мы наблюдаем стремительный прогресс в области машинного обучения, что, в том числе, обусловлено развитием глубокого обучения (deep learning, DL) и связанных с ним DL-фреймворков Python.

Сейчас вновь возникла необходимость в создании подобного практического руководства, позволяющего разработчикам при помощи актуальных библиотек Python решать задачи машинного *и* глубокого обучения. В основу данной книги легла существующая (потрясающая) работа автора первого издания, включающая:

- обновление существующих примеров в соответствии с актуальными версиями и фреймворками Python;
- внедрение современных практик работы с источниками и аналитикой данных, машинным и глубоким обучением;
- расширение блока глубокого обучения и включение разделов с тензорами, нейронными сетями, а также тренировкой текстовых и визуальных моделей глубокого обучения при помощи PyTorch;
- предоставление моделей через API.

В качестве модели повествования в данной книге, как и в первом издании, используется подход, основанный на решении практических задач машинного обучения. Здесь представлены более двухсот готовых решений (достаточно скопировать и запустить) для наиболее распространенных задач, с которыми сталкиваются исследователи данных и разработчики моделей машинного обучения.

Условные обозначения

В этой книге используются следующие условные обозначения:

- *курсивный шрифт* обозначает новые термины;

- **полужирный шрифт** обозначает URL-адреса, адреса электронной почты, элементы интерфейса программ;
- моноширинный шрифт используется для оформления листингов программ, а также в основном тексте для обозначения фрагментов кода программы, таких как имена переменных или функций, имена баз данных, типы данных, переменные среды, инструкции и ключевые слова;
- **моноширинный полужирный шрифт** обозначает результаты работы фрагментов кода или программ;
- *моноширинный курсивный шрифт* обозначает комментарии в программном коде.

Использование примеров кода

Дополнительные материалы доступны в репозитории на GitHub (**https://oreil.ly/MLwPython**). В нем представлены инструкции для запуска интерактивного блокнота Jupyter Notebook в контейнере Docker со всеми зависимостями, используемыми в книге. Вы сможете копировать команды из данной книги в блокнот, чтобы убедиться в воспроизводимости всех примеров.

По поводу любых технических вопросов и проблем, возникших у вас при использовании примеров кода, пожалуйста, пишите по адресу электронной почты support@oreilly.com.

Эта книга создана, чтобы помочь вам делать вашу работу. В целом если к книге прилагается какой-либо пример кода, можете применять его в своих программах и документации. Обращаться к нам за разрешением нет необходимости, разве что вы копируете значительную часть кода.

Благодарности

Второе издание данной книги стало возможным только благодаря невероятно качественному и хорошо структурированному материалу от автора первого издания Криса Элбона (Chris Albon). Благодаря этому вкладу моя работа в качестве главного автора второго издания стала существенно легче.

Для нас очевиден факт стремительного развития области машинного обучения, и представленные во втором издании дополнения и изменения являются в том числе заслугой моих коллег. Я бы хотел выразить отдельную благодарность моим коллегам из Etsy, включая Андрию Хеймана (Andrea Heyman), Марию Гомез (Maria Gomez), Алека Маэлструма (Alek Maelstrum) и Брайана Штидта (Brian Schmidt) за согласие на участие в написании некоторых глав и в навязанных ими

мозговых штурмах, что в конечном счете помогло сформировать целостный материал для данного издания. Кроме того, я бы хотел поблагодарить технических редакторов: Джигиазу Гровер (Jigyasa Grover), Мэтью Тана (Matteus Tanha) и Ганеш Харке (Ganesh Harke), а также редакторов из O'Reilly: Джеффа Блэйела (Jeff Bleiel), Николь Батерфилд (Nicole Butterfield) и Клэр Лэйлок (Clare Laylock). Стоит сказать, что количество людей, которые в той или иной мере помогли мне в написании этой книги и достижении таких результатов, очень внушительное. Я бы хотел поблагодарить всех, кто так или иначе составил мне компанию в этом путешествии через теорию и практику машинного обучения и помог написать эту книгу. Всех люблю.

ГЛАВА 1

Работа с векторами, матрицами и массивами в NumPy

Введение

Библиотека NumPy — основной инструмент в стеке технологий для машинного обучения на основе Python. С ее помощью можно эффективно осуществлять операции над структурами данных, часто используемыми в машинном обучении: векторами (vectors), матрицами (matrices) и тензорами (tensors). Хотя NumPy не является объектом исследования в данной книге, она будет часто упоминаться в последующих главах. В этой главе мы рассмотрим наиболее распространенные операции в NumPy, которые вы с большой вероятностью будете использовать в процессе работы над проектами, связанными с машинным обучением.

1.1. Создание вектора

Задача

Вам необходимо создать вектор.

Решение

При помощи NumPy создадим одномерный массив (array):

```
# Загрузить библиотеку
import numpy as np

# Создать вектор-строку
vector_row = np.array([1, 2, 3])

# Создать вектор-столбец
vector_column = np.array([[1],
                          [2],
                          [3]])
```

Обсуждение

Основной структурой данных в NumPy является многомерный массив (multidimensional array). Понятие вектора тождественно одномерному массиву. Для того чтобы получить вектор, мы просто создаем одномерный массив. Эти массивы, как и векторы, могут располагаться горизонтально (например, строки) или вертикально (например, столбцы).

Дополнительные материалы для чтения

- Vectors, Math Is Fun ("Векторы", математический ресурс Math's Fun ("Забавная математика"); **https://oreil.ly/43I-b**).
- Euclidean vector ("Эвклидов вектор", Википедия; **https://oreil.ly/er78t**).

1.2. Создание матрицы

Задача

Вам необходимо создать матрицу.

Решение

При помощи NumPy создадим двумерный массив:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2],
                   [1, 2],
                   [1, 2]])
```

Обсуждение

Для создания матрицы в NumPy мы можем использовать двумерный массив. В нашем примере матрица состоит из трех строк и двух столбцов (столбец единиц и столбец двоек).

В NumPy существует специальная структура матричных данных:

```
matrix_object = np.mat([[1, 2],
                        [1, 2],
                        [1, 2]])
```

```
matrix([[1, 2],
        [1, 2],
        [1, 2]])
```

Однако использовать структуру матричных данных не рекомендуется по двум причинам. Во-первых, стандартной структурой данных в NumPy де-факто являются массивы. Во-вторых, подавляющее большинство операций в NumPy возвращает массивы, а не матричные объекты.

Дополнительные материалы для чтения

- Matrix ("Матрица", Википедия; **https://oreil.ly/tnRJw**).
- Matrix, Wolfram MathWorld ("Матрица", математический ресурс Wolfram MathWorld; **https://oreil.ly/76jUS**).

1.3. Создание разреженной матрицы

Задача

У вас есть данные с небольшим количеством ненулевых значений, и вы хотите их представить.

Решение

Создадим разреженную матрицу (sparse matrix):

```
# Загрузить библиотеку
import numpy as np
from scipy import sparse

# Создать матрицу
matrix = np.array([[0, 0],
                   [0, 1],
                   [3, 0]])

# Создать матрицу со сжатым хранением строкой (compressed sparse row; CSR)
matrix_sparse = sparse.csr_matrix(matrix)
```

Обсуждение

В процессе машинного обучения очень часто приходится работать с большими объемами данных, где бóльшая часть элементов имеет нулевые значения. Например, представим матрицу, в которой каждому столбцу соответствует один фильм на

платформе Netflix, каждой строке — пользователь платформы, а значения показывают, сколько раз пользователь посмотрел конкретный фильм. Такая матрица будет содержать тысячи столбцов и миллионы строк. Однако ввиду того, что основная часть пользователей не смотрела большинство этих фильмов, большее количество элементов будет содержать нулевые значения.

Разреженная матрица — это матрица с преимущественно нулевыми значениями элементов. Такие матрицы хранят только ненулевые значения, тогда как другие значения по умолчанию принимаются за ноль, что в конечном счете позволяет экономить значительные ресурсы. В нашем случае мы при помощи NumPy создали массив с двумя ненулевыми значениями, который затем преобразовали в разреженную матрицу. Воспользовавшись командой вывода разреженной матрицы, мы увидим, что в ней хранятся только ненулевые значения:

```
# Вывести разреженную матрицу
print(matrix_sparse)
  (1, 1)	1
  (2, 0)	3
```

Разреженные матрицы могут быть нескольких типов. В случае использования матрицы со *сжатым хранением строкой* (compressed sparse row, CSR) (начинающиеся с нуля) индексы элементов `(1, 1)` и `(2, 0)` соответствуют ненулевым значениям `1` и `3`. К примеру, элемент `1` находится на второй строке второго столбца. Для того чтобы лучше понять преимущество использования разреженных матриц, можно создать матрицу большего размера с гораздо бóльшим числом нулевых элементов и сравнить результат вывода этой матрицы с нашим первым вариантом:

```
# Создать большую матрицу
matrix_large = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                         [3, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

# Создать матрицу со сжатым хранением строкой (CSR)
matrix_large_sparse = sparse.csr_matrix(matrix_large)

# Вывести изначальную разреженную матрицу
print(matrix_sparse)
  (1, 1)	1
  (2, 0)	3

# Вывести большую разреженную матрицу
print(matrix_large_sparse)
  (1, 1)	1
  (2, 0)	3
```

Как мы можем наблюдать, несмотря на добавление большого количества нулевых элементов в новую матрицу, результат ее вывода в виде разреженной матрицы со-

ответствует выводу нашей первой матрицы. Таким образом, добавление элементов с нулями не повлияло на размер разреженной матрицы.

Как мы отметили выше, существует несколько типов разреженных матриц, включая сжатое хранение столбцом (compressed sparse column, CSC), список списков (list of lists, LIL) и словарь ключей (dictionary of keys, DOK). Подробное рассмотрение каждого типа не входит в задачи данной книги; стоит лишь отметить, что не существует единого верного типа разреженной матрицы — они имеют свои отличительные особенности, и нам следует внимательно подбирать нужный тип под конкретные задачи.

Дополнительные материалы для чтения

- Sparse Matrices ("Разряженные матрицы", документация SciPy; **https://oreil.ly/zBBRB**).
- 101 Ways to Store a Sparse Matrix ("101 способ хранения разряженной матрицы", блог; **https://oreil.ly/sBQhN**).

1.4. Предварительное распределение массивов в NumPy

Задача

Вам нужно создать массив с заранее заданным размером и некоторыми значениями.

Решение

В NumPy есть функции создания векторов и матриц любых размеров с использованием нулей, единиц или необходимых вам значений:

```
# Загрузить библиотеку
import numpy as np

# При помощи параметра shape создать вектор формы (1, 5) и нулевыми значениями
vector = np.zeros(shape=5)

# Вывести матрицу
print(vector)
array([0., 0., 0., 0., 0.])

# Создать матрицу формы (3, 3), содержащую единицы
matrix = np.full(shape=(3,3), fill_value=1)
```

```
# Вывести вектор
print(matrix)
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Обсуждение

Создание массивов с заранее заданными данными может повысить производительность кода или пригодиться при тестировании алгоритмов с использованием синтетических данных. Предварительное распределение массива со значениями по умолчанию (например, нулями) применяется повсеместно во многих языках программирования.

1.5. Обращение к элементу массива

Задача

Вам необходимо вывести один или несколько элементов вектора или матрицы.

Решение

Используя массивы NumPy, можно легко вывести элементы векторов и матриц:

```
# Загрузить библиотеку
import numpy as np

# Создать вектор-строку
vector = np.array([1, 2, 3, 4, 5, 6])

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Вывести третий элемент вектора
vector[2]
3

# Вывести элемент второй строки, второго столбца
matrix[1,1]
5
```

Обсуждение

Следует помнить, что, как и в случае с большинством других структур данных в Python, в массивах NumPy индексация начинается с нуля, другими словами, индекс первого элемента — 0, а не 1. В NumPy представлено множество способов вывода элементов или групп элементов массивов (например, при помощи индексов или срезов (slicing)):

```
# Вывести все элементы вектора
vector[:]
array([1, 2, 3, 4, 5, 6])

# Вывести все элементы до третьего включительно
vector[:3]
array([1, 2, 3])

# Вывести все элементы после третьего
vector[3:]
array([4, 5, 6])

# Вывести последний элемент
vector[-1]
6

# Вывести все элементы вектора в обратном порядке
vector[::-1]
array([6, 5, 4, 3, 2, 1])

# Вывести элементы первых двух строк и всех столбцов матрицы
matrix[:2,:]
array([[1, 2, 3],
       [4, 5, 6]])

# Вывести элементы всех строк и второго столбца
matrix[:,1:2]
array([[2],
       [5],
       [8]])
```

1.6. Описание матрицы

Задача

Вам необходимо описать форму (shape), число элементов (size) и размерность (dimensions) матрицы.

Решение

Используем атрибуты shape, size и ndim объекта NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

# Выяснить количество строк и столбцов
matrix.shape
(3, 4)

# Выяснить количество элементов (умножим строки на столбцы)
matrix.size
12

# Выяснить размерность
matrix.ndim
2
```

Обсуждение

Хотя в этих операциях нет ничего сложного, следует время от времени проверять форму и число элементов как для последующих вычислений, так и для контроля после операций.

1.7. Использование функций для каждого элемента

Задача

Вам нужно применить некоторые функции ко всем элементам массива.

Решение

Воспользуемся представленным в NumPy классом векторизации vectorize:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Создать функцию, которая будет добавлять 100 к значению элемента
add_100 = lambda i: i + 100

# Создать функцию vectorize
vectorized_add_100 = np.vectorize(add_100)

# Применить функцию ко всем элементам матрицы
vectorized_add_100(matrix)
array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

Обсуждение

При помощи векторизации в NumPy функцию можно применить ко всем элементам массива или среза массива. Следует отметить, что в основе векторизации лежит цикл for, где в итерациях участвует каждый элемент, без прироста производительности. Более того, при работе с массивами NumPy у нас есть возможность осуществлять операции между массивами, в том числе с разным числом измерений (так называемое *транслирование* (broadcasting)). Например, мы можем использовать транслирование, чтобы упростить созданный нами ранее вариант:

```
# Добавить 100 ко всем элементам
matrix + 100
array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

Транслирование является одним из основных способов осуществления простых операций со всеми элементами массива NumPy, однако не все формы и ситуации поддаются транслированию.

1.8. Поиск наибольших и наименьших значений

Задача

Вам необходимо найти максимальные и минимальные значения массива.

Решение

Воспользуемся функциями max и min в NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Вывести элемент с наибольшим значением
np.max(matrix)
9

# Вывести элемент с наименьшим значением
np.min(matrix)
1
```

Обсуждение

Часто бывает так, что нам необходимо узнать максимальные и минимальные значения элементов массива или подмножества массива. Для этого подойдут функции max и min. При помощи параметра axis мы можем применить операцию к элементам заданной оси:

```
# Найти наибольший элемент в каждом столбце
np.max(matrix, axis=0)
array([7, 8, 9])

# Найти наибольший элемент в каждой строке
np.max(matrix, axis=1)
array([3, 6, 9])
```

1.9. Вычисление среднего значения, дисперсии и стандартного отклонения

Задача

Вы хотите вычислить наглядные статистические метрики массива.

Решение

Воспользуемся функциями NumPy `mean`, `var` и `std`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Вывести среднее значение
np.mean(matrix)
5.0

# Вывести значение дисперсии
np.var(matrix)
6.666666666666667

# Вывести значение стандартного отклонения
np.std(matrix)
2.5819888974716112
```

Обсуждение

Подобно функциям `max` и `min`, мы можем получить статистические данные как для всей матрицы, так и для отдельной оси:

```
# Определить среднее значение для каждого столбца
np.mean(matrix, axis=0)
array([ 4.,  5.,  6.])
```

1.10. Изменение формы массива

Задача

Вы хотите изменить форму (количество строк и столбцов) массива, не меняя значений элементов.

Решение

Используем метод reshape библиотекиNumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу размером 4x3
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9],
                   [10, 11, 12]])

# Изменить форму матрицы на 2x6
matrix.reshape(2, 6)
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

Обсуждение

При помощи метода reshape мы можем изменить структуру массива, сохраняя имеющиеся данные и перераспределяя их между строками и столбцами. Однако при изменении формы матрицы старый и новый варианты должны содержать одинаковое количество элементов. Для измерения количества элементов есть атрибут size:

```
matrix.size
```

```
12
```

Метод reshape может принимать очень удобный аргумент -1, который, в сущности, означает "столько, сколько нужно". Таким образом, вызов метода reshape(1, -1) означает одну строку и необходимое для наших потребностей число столбцов:

```
matrix.reshape(1, -1)
```

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
```

Наконец, если мы укажем только один аргумент, то получим одномерный массив, длина которого будет равна этому числу:

```
matrix.reshape(12)
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

1.11. Транспонирование вектора или матрицы

Задача

Вам необходимо транспонировать (transpose) вектор или матрицу.

Решение

Используем для этого функцию `T`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Транспонировать матрицу
matrix.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Обсуждение

Транспонирование (transposing) является базовой операцией в линейной алгебре, которая позволяет поменять местами индикаторы столбца и строки каждого элемента[1]. При выходе за рамки класса линейной алгебры часто упускается тот факт, что технически вектор не может быть транспонирован, т. к. он представляет собой набор значений:

```
# Транспонирование вектора
np.array([1, 2, 3, 4, 5, 6]).T
array([1, 2, 3, 4, 5, 6])
```

Под транспонированием вектора обычно подразумевается преобразование вектора-строки в вектор-столбец (обратите внимание на дополнительную пару квадратных скобок), и наоборот:

```
# Транспонирование вектора-строки
np.array([[1, 2, 3, 4, 5, 6]]).T
```

[1] То есть столбец становится строкой, а строка — столбцом. — *Прим. ред.*

```
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

1.12. Преобразование матрицы в одномерный вектор

Задача

Вам нужно преобразовать матрицу в одномерный массив.

Решение

Воспользуемся методом `flatten`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Преобразовать матрицу в одномерный массив
matrix.flatten()
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Обсуждение

Метод `flatten` — это простой способ преобразования матрицы в одномерный массив. В качестве альтернативного решения мы можем воспользоваться методом `reshape` для создания вектора-строки:

```
matrix.reshape(1, -1)
array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

Другим популярным способом превращения массивов в плоские списки (flatten arrays) является использование метода `ravel`. В отличие от метода `flatten`, возвращающего копию оригинального массива, метод `ravel` напрямую взаимодействует с объектом оригинала, что в некоторой мере ускоряет процесс ее выполнения. Кроме того, с его помощью мы можем выравнивать списки массивов, что недоступно с

методом flatten. Данную операцию удобно использовать с большими массивами и для ускорения выполнения кода:

```
# Создать первую матрицу
matrix_a = np.array([[1, 2],
                     [3, 4]])

# Создать вторую матрицу
matrix_b = np.array([[5, 6],
                     [7, 8]])

# Создать список из матриц
matrix_list = [matrix_a, matrix_b]

# Выровнять весь список матриц
np.ravel(matrix_list)
array([1, 2, 3, 4, 5, 6, 7, 8])
```

1.13. Определение ранга матрицы

Задача

Нам необходимо узнать ранг матрицы (rank of a matrix).

Решение

Используем NumPy-функцию matrix_rank из линейной алгебры (linalg):

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 1, 1],
                   [1, 1, 10],
                   [1, 1, 15]])

# Вывести ранг матрицы
np.linalg.matrix_rank(matrix)
2
```

Обсуждение

Ранг матрицы соответствует размерности пространства векторов — столбцам или строкам. Функция matrix_rank в NumPy существенно упрощает поиск ранга матрицы.

Дополнительные материалы для чтения

- The Rank of a Matrix, CliffsNotes ("Ранг матрицы", учебный курс CliffsNotes; **https://oreil.ly/Wg9ZG**).

1.14. Вывод диагонали матрицы

Задача

Необходимо получить элементы матрицы, располагающиеся по диагонали.

Решение

Воспользуемся NumPy-методом `diagonal`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Вернуть элементы по диагонали матрицы
matrix.diagonal()
array([1, 4, 9])
```

Обсуждение

В NumPy элементы по диагонали можно легко получить по помощи метода `diagonal`. Кроме вывода элементов главной диагонали, вы можете извлечь параметры побочных диагоналей:

```
# Вернуть значение диагонали над главной диагональю
matrix.diagonal(offset=1)
array([2, 6])

# Вернуть значение диагонали под главной диагональю
matrix.diagonal(offset=-1)
array([2, 8])
```

1.15. Вычисление суммы по диагонали матрицы

Задача

Вам нужно рассчитать сумму элементов, стоящих на диагонали матрицы.

Решение

Используем метод `trace`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Вернуть значение суммы элементов на диагонали матрицы
matrix.trace()
```

14

Обсуждение

След матрицы (trace) является суммой элементов диагонали и часто используется в методах машинного обучения. При помощи метода `trace` мы можем вычислить след многомерной матрицы в NumPy. В качестве альтернативы можно вернуть элементы диагонали матрицы и затем сложить их:

```
# Вернуть значения элементов диагонали и сложить их
sum(matrix.diagonal())
```

14

Дополнительные материалы для чтения

- The Trace of a Square Matrix ("След квадратной матрицы", математический ресурс MathOnline; **https://oreil.ly/AhX1b**)[2].

[2] См. **https://ru.ruwiki.ru/wiki/След_матрицы**. — *Прим. ред.*

1.16. Вычисление скалярного произведения

Задача

Необходимо рассчитать скалярное произведение (dot product) двух векторов.

Решение

Для этого в NumPy есть функция `dot`:

```
# Загрузить библиотеку
import numpy as np

# Создать два вектора
vector_a = np.array([1,2,3])
vector_b = np.array([4,5,6])

# Найти скалярное произведение
np.dot(vector_a, vector_b)
```

```
32
```

Обсуждение

Скалярное произведение двух векторов **a** и **b** описывается при помощи формулы:

$$\sum_{i=1}^{n} a_i b_i,$$

где a_i — i-й элемент вектора **a**; b_i — i-й элемент вектора **b**. Мы можем узнать скалярное произведение в NumPy при помощи функции `dot`. Начиная с Python версии 3.5, мы также можем использовать новый оператор `@`:

```
# Найти скалярное произведение
vector_a @ vector_b
```

```
32
```

Дополнительные материалы для чтения

- Vector Dot Product and Vector Length, Khan Academy ("Скалярное произведение и длина вектора", общепредметный учебный ресурс Khan Academy; **https://oreil.ly/MpBt7**).
- Dot Product, Paul's Online Math Notes ("Скалярное произведение", математический ресурс Paul's Online Math Notes; **https://oreil.ly/EprM1**).

1.17. Сложение и вычитание матриц

Задача

Вы хотите сложить и вычесть две матрицы.

Решение

Используем функции NumPy add и subtract:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix_a = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 2]])

# Создать матрицу
matrix_b = np.array([[1, 3, 1],
                     [1, 3, 1],
                     [1, 3, 8]])

# Сложить две матрицы
np.add(matrix_a, matrix_b)
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])

# Вычесть одну матрицу из другой
np.subtract(matrix_a, matrix_b)
array([[ 0, -2,  0],
       [ 0, -2,  0],
       [ 0, -2, -6]])
```

Обсуждение

Мы также можем воспользоваться операторами + и - для решения этой задачи:

```
# Сложить две матрицы
matrix_a + matrix_b
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

1.18. Умножение матриц

Задача

Необходимо перемножить матрицы.

Решение

Воспользуемся функцией NumPy `dot`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix_a = np.array([[1, 1],
                     [1, 2]])

# Создать матрицу
matrix_b = np.array([[1, 3],
                     [1, 2]])

# Перемножить две матрицы
np.dot(matrix_a, matrix_b)
array([[2, 5],
       [3, 7]])
```

Обсуждение

Начиная с Python версии 3.5, мы также можем использовать новый оператор `@`:

```
# Перемножить две матрицы
matrix_a @ matrix_b
array([[2, 5],
       [3, 7]])
```

Для поэлементного умножения можно использовать оператор `*`:

```
# Перемножить поэлементно две матрицы
matrix_a * matrix_b
array([[1, 3],
       [1, 4]])
```

Дополнительные материалы для чтения

- Array vs. Matrix Operations, MathWorks ("Операции над массивами против матричных операций", специализированный ресурс MathWorks для глубокого самообучения; **https://oreil.ly/EprM1**).

1.19. Создание обратной матрицы

Задача

Дана квадратная матрица, и вы хотите создать обратную ей матрицу.

Решение

Воспользуемся функцией `inv` из линейной алгебры NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 4],
                   [2, 5]])

# Создать обратную матрицу
np.linalg.inv(matrix)
array([[-1.66666667,  1.33333333],
       [ 0.66666667, -0.33333333]])
```

Обсуждение

Для квадратной матрицы $\mathbf{A}$ обратной матрицей является $\mathbf{A}^{-1}$, а отношение между ними можно выразить формулой:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I},$$

где $\mathbf{I}$ — единичная матрица. Если матрица обратима, то обратную матрицу $\mathbf{A}^{-1}$ в NumPy можно вычислить при помощи функции `linalg.inv`. Для наглядности можно умножить матрицу на обратную ей матрицу и в результате получить единичную матрицу:

```
# Перемножить матрицу и обратную матрицу
matrix @ np.linalg.inv(matrix)
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Дополнительные материалы для чтения

- Inverse of a Matrix ("Обратная матрица", математический ресурс mathwords.com; **https://oreil.ly/SwRXC**).

1.20. Генерация псевдослучайных значений

Задача

Вы хотите сгенерировать псевдослучайные значения.

Решение

Используем функцию NumPy `random`:

```
# Загрузить библиотеку
import numpy as np

# Установить целое начальное значение (seed)
np.random.seed(0)

# Сгенерировать три случайных дробных числа в диапазоне от 0.0 до 1.0
np.random.random(3)
array([ 0.5488135 ,  0.71518937,  0.60276338])
```

Обсуждение

В данной книге мы не сможем описать все способы генерации случайных чисел в NumPy, т. к. их слишком много. В нашем примере мы создавали числа с плавающей запятой, однако эта функция также позволяет нам генерировать целые числа:

```
# Сгенерировать три случайных целых в диапазоне от 0 до 10
np.random.randint(0, 11, 3)
array([3, 7, 9])
```

Другим способом является получение чисел при помощи распределения (помните, что технически они не являются случайными):

```
# Получить три числа нормального распределения со средним значением 0.0
# и стандартным отклонением 1.0
np.random.normal(0.0, 1.0, 3)
array([-1.42232584,  1.52006949, -0.29139398])
```

```
# Получить три числа из логистического распределения со средним значением 0.0
# и параметром уплощенности графика распределения 1.0
np.random.logistic(0.0, 1.0, 3)
array([-0.98118713, -0.08939902,  1.46416405])

# Получить три числа, которые будут больше или равны 1.0 и меньше 2.0
np.random.uniform(1.0, 2.0, 3)
array([ 1.47997717,  1.3927848 ,  1.83607876])
```

Наконец, в некоторых случаях может быть полезно несколько раз повторно получить те же случайные числа для проверки воспроизводимости и предсказуемости результата. В таком случае нам нужно задать значение seed (целое число) для генератора псевдослучайных чисел. Если для разных процессов задан единый параметр seed, мы каждый раз будем получать одинаковый результат. В данной книге мы будем постоянно обращаться к параметру seed, благодаря чему код из книги и запускаемый вами на компьютере будут иметь идентичный результат.

ГЛАВА 2

Загрузка данных

Введение

Создание любой модели машинного обучения всегда начинается с загрузки необработанных данных в систему. Эти данные могут включать логи (журнальные файлы), наборы данных или облачные хранилища BLOB-объектов (например, Amazon S3). Более того, нам зачастую необходимо получать данные из различных источников.

В этой главе мы рассмотрим методы загрузки данных из различных источников, включая CSV-файлы и базы данных SQL. Кроме того, мы изучим способы генерирования искусственных данных с заданными параметрами для тестирования. В процессе изучения различных методов загрузки данных в экосистеме Python мы уделим особое внимание методам загрузки внешних данных при помощи инструментов модуля pandas и созданию искусственных данных посредством библиотеки scikit-learn, написанной на языке Python и предназначенной для машинного обучения.

2.1. Загрузка готового набора данных

Задача

Вы хотите загрузить существующий пример набора данных из библиотеки scikit-learn.

Решение

Библиотека scikit-learn обеспечивает загрузку популярных наборов данных:

```
# Подключить возможность загрузки наборов данных библиотеки scikit-learn
from sklearn import datasets

# Загрузить набор данных digits
digits = datasets.load_digits()

# Создать матрицу признаков (features)
features = digits.data
```

```
# Определить вектор целевых данных (target)
target = digits.target

# Вывести полученный результат
features[0]
array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
        15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
         8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
         5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
         1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
         0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

Обсуждение

В начале знакомства с алгоритмами и методами машинного обучения, как правило, не очень хочется заниматься загрузкой, преобразованием и очисткой реальных наборов данных. Для этого в scikit-learn есть готовые наборы данных, которые можно легко подключить. Они получили название "игрушечных" наборов данных ("toy" datasets), т. к. являются уже уменьшенным и очищенным вариантом реальных данных, с которыми мы могли бы работать. Ниже представлены некоторые популярные наборы данных scikit-learn.

`load_iris`

Ирисы Фишера (iris flowers) состоят из данных о 150 экземплярах ириса. Этот набор данных хорошо подходит для изучения алгоритмов классификации.

`load_digits`

Данный массив состоит из 1797 изображений начертания цифр. С его помощью можно научиться классификации изображений.

Для того чтобы ознакомиться с каждым набором данных, воспользуемся функцией вывода с атрибутом `DESCR`:

```
# Подключить возможность загрузки наборов данных библиотеки scikit-learn
from sklearn import datasets

# Загрузить набор данных digits
digits = datasets.load_digits()

# Вывести атрибуты
print(digits.DESCR)

.. _digits_dataset:
Optical recognition of handwritten digits dataset
```

```
--------------------------------------------------
**Data Set Characteristics:**
    :Number of Instances: 1797
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998
...
```

Дополнительные материалы для чтения

- Игрушечные наборы данных scikit-learn (**https://oreil.ly/WS1gc**).
- Набор данных Digit (**https://oreil.ly/0hukv**).

2.2. Создание искусственного набора данных

Задача

Вам нужно сгенерировать набор с искусственными данными.

Решение

В scikit-learn существует множество способов создания искусственных данных. Отдельно стоит выделить три подхода: `make_regression`, `make_classification` и `make_blobs`.

Если наш набор данных станет частью модели линейной регрессии, нам подойдет функция `make_regression`:

```
# Загрузить библиотеку
from sklearn.datasets import make_regression

# Создать матрицу признаков (features), определить вектор
# целевых данных (target vector) и коэффициенты (true coefficients)
features, target, coefficients = make_regression(n_samples = 100,
                                                 n_features = 3,
                                                 n_informative = 3,
                                                 n_targets = 1,
                                                 noise = 0.0,
                                                 coef = True,
                                                 random_state = 1)
```

```
# Вывести матрицу признаков и вектор целевых данных
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])
```

```
Feature Matrix
 [[ 1.29322588 -0.61736206 -0.11044703]
 [-2.793085    0.36633201  1.93752881]
 [ 0.80186103 -0.18656977  0.0465673 ]]
Target Vector
 [-10.37865986  25.5124503   19.67705609]
```

Если нам нужно создать набор данных для классификации, можно использовать функцию make_classification:

```
# Загрузить библиотеку
from sklearn.datasets import make_classification

# Создать матрицу признаков и определить вектор целевых данных
features, target = make_classification(n_samples = 100,
                                       n_features = 3,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 2,
                                       weights = [.25, .75],
                                       random_state = 1)

# Вывести матрицу признаков и вектор целевых данных
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])
```

```
Feature Matrix
 [[ 1.06354768 -1.42632219  1.02163151]
 [ 0.23156977  1.49535261  0.33251578]
 [ 0.15972951  0.83533515 -0.40869554]]
Target Vector
 [1 0 0]
```

Наконец, если наш набор данных должен хорошо работать с кластеризацией, воспользуемся функцией make_blobs:

```
# Загрузить библиотеку
from sklearn.datasets import make_blobs
```

```
# Создать матрицу признаков и определить вектор целевых данных
features, target = make_blobs(n_samples = 100,
                              n_features = 2,
                              centers = 3,
                              cluster_std = 0.5,
                              shuffle = True,
                              random_state = 1)

# Вывести матрицу признаков и вектор целевых данных
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])
```

```
Feature Matrix
 [[ -1.22685609   3.25572052]
 [ -9.57463218  -4.38310652]
 [-10.71976941  -4.20558148]]
Target Vector
 [0 1 1]
```

Обсуждение

Как следует из примеров выше, функция `make_regression` возвращает матрицу признаков со значениями с плавающей запятой и вектор целевых данных со значениями с плавающей запятой, в то время как функции `make_classification` и `make_blobs` возвращают матрицы признаков со значениями с плавающей запятой и вектор целевых данных с целыми значениями, выражающими отношение к классу.

Благодаря использованию scikit-learn при создании наборов искусственных данных мы получаем множество вариантов контроля типа генерируемых данных. Полное описание всех параметров содержится в руководстве scikit-learn; мы разберем некоторые из них.

Количество признаков, используемых в функциях `make_regression` и `make_classification` для генерации вектора, определяется при помощи параметра `n_informative`. Если параметр `n_informative` меньше общего числа параметров (`n_features`), получившийся набор данных будет иметь избыточные параметры, которые можно определить с помощью методов выбора параметров.

Кроме того, функция `make_classification` содержит параметр `weights`, при помощи которого можно создать дисбаланс классов (imbalanced classes) в процессе генерации наборов данных. Например, задав параметр `weights = [.25, .75]`, мы получим набор данных, в котором 25% объектов будут принадлежать к одному классу, а 75% — к другому.

В функции make_blobs есть параметр centers, который позволяет нам задать число генерируемых кластеров. Воспользуемся библиотекой Matplotlib для визуализации наших кластеров (рис. 2.1), созданных при помощи make_blobs:

```
# Загрузить библиотеку
import matplotlib.pyplot as plt

# Вывести диаграмму рассеяния
plt.scatter(features[:,0], features[:,1], c=target)
plt.show()
```

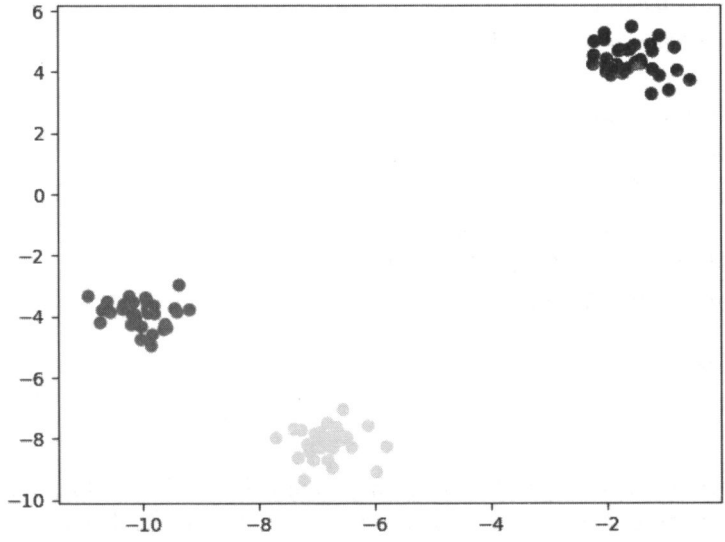

Рис. 2.1. Диаграмма рассеяния

Дополнительные материалы для чтения

- Документация по make_regression (**https://oreil.ly/VrtN3**).
- Документация по make_classification (**https://oreil.ly/rehc-**).
- Документация по make_blobs (**https://oreil.ly/1LZAI**).

2.3. Загрузка CSV-файла

Задача

Необходимо импортировать файл со значениями, разделенными запятыми (comma-separated value, CSV).

Решение

Воспользуемся функцией read_csv из библиотеки pandas для загрузки локального или хранящегося удаленно CSV-файла в структуру данных DataFrame:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/data.csv'

# Загрузить набор данных
dataframe = pd.read_csv(url)

# Вывести две первые сроки
dataframe.head(2)
```

```
   integer            datetime  category
0        5  2015-01-01 00:00:00         0
1        5  2015-01-01 00:00:01         0
```

Обсуждение

Стоит отметить две вещи, касающиеся загрузки CSV-файлов. Во-первых, будет полезно просмотреть содержимое файла перед его загрузкой. Таким образом, вы заранее ознакомитесь со структурой набора данных и параметрами, необходимыми для загрузки файла. Во-вторых, в read_csv представлено более 30 параметров, и процесс изучения документации может показаться сложным. Однако большинство этих параметров необходимы для управления разнообразными CSV-форматами.

Как следует из названия, значения в CSV-файлах буквально разделяются запятыми (например, строка может иметь такой вид: 2,"2015-01-01 00:00:00",0); тем не менее в CSV-файлах часто используются и другие разделители, такие как табуляция (в таком случае используется TSV-файл).

При помощи параметра sep в pandas мы можем задать разделитель, который будет использован в файле. В структуре CSV-файлов принято, хотя и не обязательно, использовать первую строку для заголовков столбцов (к примеру, в нашем случае это integer (целые числа), datetime (дата/время) и category (категория)). Для того чтобы определить наличие и расположение строки с заголовками, мы можем применять параметр header. Если такая строка отсутствует, необходимо указать header=None.

Функция read_csv возвращает DataFrame (фрейм данных) — основной тип данных в pandas, используемый для работы с таблицами. Мы разберем этот аспект более подробно далее в книге.

2.4. Загрузка файла Excel

Задача

Нам нужно загрузить электронную таблицу Excel.

Решение

Используем функцию `read_excel` библиотеки pandas для загрузки таблицы Excel:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/data.xlsx'

# Загрузить данные
dataframe = pd.read_excel(url, sheet_name=0, header=0)

# Вывести две первые сроки
dataframe.head(2)
```

	integer	datetime	category
	5	2015-01-01 00:00:00	0
0	5	2015-01-01 00:00:01	0
1	9	2015-01-01 00:00:02	0

Обсуждение

Наше решение похоже на предыдущий вариант для чтения CSV-файлов. Главное отличие заключается в добавлении параметра `sheet_name`, при помощи которого мы указываем номер нужного листа в документе Excel. Параметр `sheet_name` может принимать как строковые значения (strings) с названием листа, так и числовые значения (integers), указывающие на номер листа (индексация начинается с нуля). Для загрузки нескольких листов необходимо перечислить их. К примеру, если мы зададим параметр `sheet_name=[0, 1, 2, "Monthly Sales"]`, то получим словарь (dictionary) значений DataFrames библиотеки pandas, содержащий первый, второй и третий листы, а также лист с названием `Monthly Sales`.

2.5. Загрузка файла JSON

Задача

Необходимо загрузить файл JSON для предварительной обработки данных.

Решение

Файл JSON можно конвертировать в объект pandas при помощи функции `read_json`:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/data.json'

# Загрузить данные
dataframe = pd.read_json(url, orient='columns')

# Вывести две первые сроки
dataframe.head(2)
```

	category	datetime	integer
0	0	2015-01-01 00:00:00	5
1	0	2015-01-01 00:00:01	5

Обсуждение

Импорт файлов JSON в pandas похож на другие операции, описанные выше. Основным отличием является наличие параметра `orient` для описания структуры файла JSON при добавлении в pandas. Также можно попробовать разные аргументы (`split`, `records`, `index`, `columns` или `values`), чтобы опытным путем понять, какой подходит лучше. Еще одним удобным инструментом pandas является функция `json_normalize`, при помощи которой слабоструктурированные файлы JSON преобразовываются в DataFrame.

Дополнительные материалы для чтения

- Документация по `json_normalize` (**https://oreil.ly/nuvIB**).

2.6. Загрузка файла Parquet

Задача

Вы хотите добавить файл Parquet.

Решение

Функция read_parquet библиотеки pandas позволяет прочесть файлы Parquet:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://machine-learning-python-cookbook.s3.amazonaws.com/data.parquet'

# Загрузить данные
dataframe = pd.read_parquet(url)

# Вывести две первые сроки
dataframe.head(2)
```

	category	datetime	integer
0	0	2015-01-01 00:00:00	5
1	0	2015-01-01 00:00:01	5

Обсуждение

Parquet — это популярный формат хранения данных в больших объемах. Он часто используется в инструментах для работы большими данными (big data), например Hadoop и Spark. Мы не будем подробно останавливаться на PySpark в данной книге. Вместе с тем будет полезно узнать, как переносить данные из хранилищ в формате Parquet и подобных в DataFrame и работать с ними, поскольку они используются компаниями при работе с большими объемами данных.

Дополнительные материалы для чтения

- Документация Apache Parquet (**https://oreil.ly/nuvIB**).

2.7. Загрузка файла Avro

Задача

Необходимо загрузить файл Avro в DataFrame библиотеки pandas.

Решение

Воспользуемся функцией `read_avro` из библиотеки pandavro:

```
# Загрузить библиотеку
import requests
import pandavro as pdx

# Добавить URL-адрес
url = 'https://machine-learning-python-cookbook.s3.amazonaws.com/data.avro'

# Скачать файл
r = requests.get(url)
open('data.avro', 'wb').write(r.content)

# Загрузить данные
dataframe = pdx.read_avro('data.avro')

# Вывести две первые сроки
dataframe.head(2)
```

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

Обсуждение

Apache Avro — это открытый формат бинарных данных, в котором структурирование данных происходит при помощи схем (schemas). На момент написания книги он не столь широко используется, как Parquet. Тем не менее мы можем наблюдать рост популярности форматов бинарных данных, включая Avro, thrift и Protocol Buffers, по причине их эффективности. Если вы работаете с крупными системами данных, то, весьма вероятно, перейдете на один из этих форматов в ближайшем будущем.

Дополнительные материалы для чтения

- Документация Apache Avro (**https://oreil.ly/Y1TJA**).

2.8. Запрос к базе данных SQLite

Задача

Вам нужно загрузить информацию из базы данных при помощи языка структурированных запросов (structured query language, SQL).

Решение

Для создания запроса и загрузки из базы данных SQL используем функцию `read_sql_query` модуля pandas:

```
# Загрузить библиотеки
import pandas as pd
from sqlalchemy import create_engine

# Подключиться к базе данных
database_connection = create_engine('sqlite:///sample.db')

# Загрузить данные
dataframe = pd.read_sql_query('SELECT * FROM data', database_connection)

# Вывести две первые сроки
dataframe.head(2)
```

	first_name	last_name	age	preTestScore	postTestScore
0	Jason	Miller	42	4	25
1	Molly	Jacobson	52	24	94

Обсуждение

SQL является универсальным языком для обращения к базе данных. В нашем примере мы первым делом используем функцию `create_engine` для создания подключения к СУБД SQLite. Затем мы применяем функцию `read_sql_query` библиотеки pandas для создания запроса к базе данных при помощи SQL и переноса результатов в DataFrame.

SQL — это полноценный язык программирования, и хотя он не является частью предмета исследования в данной книге, его стоит изучить всем, кто планирует заниматься машинным обучением. SQL-запрос `SELECT * FROM data` к базе данных означает, что мы хотим получить данные всех столбцов (*) из таблицы с названием `data`.

Нужно отметить, что это один из немногих примеров в нашей книге, для которых необходим дополнительный код. В данном конкретном случае для использования функции `create_engine('sqlite:///sample.db')` база данных SQLite уже должна существовать.

Дополнительные материалы для чтения

- SQLite (**https://oreil.ly/8Y91T**).
- Обучающий курс W3Schools SQL (**https://oreil.ly/A7H1m**).

2.9. Удаленное подключение к базе данных SQL

Задача

Вам необходимо удаленно подключиться к базе данных SQL и получить доступ к информации в ней.

Решение

Подключимся при помощи `pymysql` и выгрузим данные в DataFrame через pandas:

```
# Импортировать библиотеки
import pymysql
import pandas as pd

# Создать подключение к БД.
# В качестве примера воспользуемся БД из
# https://github.com/kylegallatin/mysql-db-example
conn = pymysql.connect(
      host='localhost',
      user='root',
      password = "",
      db='db',
)

# Загрузить результаты SQL-запроса в DataFrame
dataframe = pd.read_sql("select * from data", conn)
```

```
# Вывести две первые сроки
dataframe.head(2)
```

```
   integer            datetime  category
0        5  2015-01-01 00:00:00         0
1        5  2015-01-01 00:00:01         0
```

Обсуждение

Среди всех примеров в данной книге этот является, возможно, самым популярным с точки зрения практического использования. Хотя способ обращения к базе данных SQLite из предыдущего примера вполне подходит для многих сценариев, в масштабах предприятия этого может быть недостаточно. При удаленном обращении к большинству баз данных SQL вам потребуется подключиться к хосту и порту на устройстве, указав имя пользователя и пароль для аутентификации. Для этого примера вам нужно локально запустить базу SQL (**https://oreil.ly/Sxjqz**) для симуляции удаленного сервера на локальном хосте; таким образом вы сможете лучше понять рабочий процесс.

Дополнительные материалы для чтения

- Документация PyMySQL (**https://oreil.ly/8zSnj**).
- Документация по `read_SQL` библиотеки pandas (**https://oreil.ly/Yb7sH**).

2.10. Загрузка данных из Google Sheet

Задача

Вы хотите напрямую обратиться к данным из Google Sheet.

Решение

Используем функцию `read_CSV` модуля pandas и добавим URL-адрес, в результате чего получим документ из Google Sheet в формате CSV:

```
# Импортировать библиотеку
import pandas as pd

# Добавить URL-адрес Google Sheet для скачивания листа в формате CSV
url = "https://docs.google.com/spreadsheets/d/" \
      "1ehC-9otcAuitqnmWksqt1mOrTRCL38dv0K9UjhwzTOA/export?format=csv"
```

```
# Загрузить файлы CSV в DataFrame
dataframe = pd.read_csv(url)

# Вывести две первые сроки
dataframe.head(2)
```

```
   integer             datetime  category
0        5  2015-01-01 00:00:00         0
1        5  2015-01-01 00:00:01         0
```

Обсуждение

Хотя мы можем легко скачать документ из Google Sheets, в некоторых случаях будет полезно иметь возможность перенести его напрямую в Python, без дополнительных операций. При помощи параметра `/export?format=csv` в конце URL-запроса, приведенного выше, мы создаем конечный объект, который можно скачать в виде файла или перенести в pandas.

Дополнительные материалы для чтения

- Google Sheets API (**https://oreil.ly/GRLzg**).

2.11. Загрузка данных из S3 bucket

Задача

Вам необходимо открыть файл CSV из хранилища S3 bucket, к которому у вас есть доступ.

Решение

Добавим параметры хранилища в pandas, предоставив доступ к объекту S3:

```
# Импортировать библиотеку
import pandas as pd

# Указать путь к файлу CSV в хранилище S3
s3_uri = "s3://machine-learning-python-cookbook/data.csv"

# Указать учетные данные пользователя AWS (замените на ваши)
ACCESS_KEY_ID = "xxxxxxxxxxxxx"
SECRET_ACCESS_KEY = "xxxxxxxxxxxxxxxx"
```

```
# Загрузить файлы CSV в DataFrame
dataframe = pd.read_csv(s3_uri,storage_options={
        "key": ACCESS_KEY_ID,
        "secret": SECRET_ACCESS_KEY,
   }
)

# Вывести две первые сроки
dataframe.head(2)
```

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

Обсуждение

В настоящее время многие предприятия хранят данные в облачных хранилищах BLOB-объектов, например Amazon S3 или Google Cloud Storage (GCS). В процессе машинного обучения нам может потребоваться часто подключаться к этим источникам для получения данных. Хотя URI-идентификатор хранилища S3 (**s3://machine-learning-python-cookbook/data.csv**) является публичным, нам по-прежнему нужно предоставлять учетные данные пользователя AWS для получения доступа. В данном случае не имеет значения, есть ли у объектов в открытом доступе HTTP URL-адрес, при помощи которого можно скачать, в том числе, интересующие нас файлы CSV.

Дополнительные материалы для чтения

- Amazon S3 (**https://oreil.ly/E-CZX**).
- AWS security credentials ("Безопасность учетных данных AWS"; **https://oreil.ly/aHBBb**).

2.12. Загрузка неструктурированных данных

Задача

Вы хотите загрузить неструктурированные данные вроде текста или изображений.

Решение

Используем стандартную функцию open в Python для загрузки информации:

```
# Импортировать библиотеку
import requests

# Добавить URL-адрес с текстовым файлом
txt_url = "https://machine-learning-python-cookbook.s3.amazonaws.com/text.txt"

# Скачать этот файл
r = requests.get(txt_url)

# Перенести информацию в локальный файл text.txt
with open('text.txt', 'wb') as f:
    f.write(r.content)

# Загрузить в файл
with open('text.txt', 'r') as f:
    text = f.read()

# Вывести содержимое
print(text)
```

```
Hello there!
```

Обсуждение

В то время как структурированные данные могут быть легко загружены из форматов CSV, JSON или других баз данные, загрузка неструктурированных данных может вызвать определенные сложности и потребовать дополнительную обработку. Иногда для открытия и загрузки файлов можно воспользоваться стандартной функцией open в Python. Таким образом, мы можем открыть и затем обработать содержимое нужного файла.

Дополнительные материалы для чтения

- Функция open в Python (**https://oreil.ly/Xuuom**).
- Менеджеры контекста в Python (**https://oreil.ly/UyZnL**).

ГЛАВА 3

Первичная обработка данных

Введение

Понятие *первичной обработки данных* (data wrangling) в широком смысле описывает процесс преобразования сырых данных в очищенные и готовые к использованию. В нашей ситуации первичная обработка является лишь одним из этапов работы с данными, который тем не менее очень важен.

Наиболее часто используемой структурой в процессе первичной обработки данных является двумерный массив DataFrame, который иногда может быть интуитивно понятным, а иногда — невероятно запутанным. DataFrame имеет форму привычной электронной таблицы со строками и столбцами. Ниже представлен пример DataFrame, в котором собрана информация о пассажирах "Титаника":

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные в виде DataFrame
dataframe = pd.read_csv(url)

# Вывести пять первых строк
dataframe.head(5)
```

```
                                     Name PClass  ...  Survived SexCode
0                   Allen, Miss Elisabeth Walton    1st  ...         1       1
1                    Allison, Miss Helen Loraine    1st  ...         0       1
2            Allison, Mr Hudson Joshua Creighton    1st  ...         0       0
3  Allison, Mrs Hudson JC (Bessie Waldo Daniels)    1st  ...         0       1
4                  Allison, Master Hudson Trevor    1st  ...         1       0
```

Стоит отметить три важные особенности в представленном DataFrame.

Во-первых, каждой строке в DataFrame соответствует одно наблюдение (observation), например пассажир, а каждому столбцу соответствует одна характеристика — пол, возраст и т. д. К примеру, если мы посмотрим на первое наблюдение, Элизабет Уолтон Ален, мы узнаем, что она плыла первым классом, ей было 29 лет, она была женщиной и пережила крушение.

Во-вторых, каждый столбец имеет название (например, `Name` (имя), `PClass` (класс билета), `Age` (возраст)), а каждая строка содержит индекс (например, `0` для везучей Элизабет Уолтон Ален). Мы можем использовать эти опции для управления наблюдениями и признаками.

В-третьих, два столбца, `Sex` (пол) и `SexCode` (код пола), содержат одинаковую информацию в разных форматах. В случае с женщиной, в графе `Sex` содержится строка `female`, тогда как в графе `SexCode` содержится числовое значение `1`. Поскольку для нас важно, чтобы все характеристики были уникальными, мы можем удалить один из этих столбцов.

В этой главе мы познакомимся с различными методами управления DataFrame при помощи библиотеки pandas для очистки и создания хорошо структурированных наборов данных наблюдений для последующей обработки.

3.1. Создание DataFrame

Задача

Вы хотите создать новый фрейм данных DataFrame.

Решение

В pandas есть много подходов к созданию новых объектов DataFrame. Воспользуемся одним из простых способов получения DataFrame при помощи словаря Python. В нашем примере словаря ключами являются названия столбцов, а значениями являются списки, элементы которых соответствуют данным строк:

```
# Загрузить библиотеку
import pandas as pd

# Создать словарь
dictionary = {
  "Name": ['Jacky Jackson', 'Steven Stevenson'],
  "Age": [38, 25],
  "Driver": [True, False]
}

# Создать DataFrame
dataframe = pd.DataFrame(dictionary)
```

```
# Вывести DataFrame
dataframe
```

```
              Name  Age  Driver
0     Jacky Jackson   38    True
1  Steven Stevenson   25   False
```

Мы можем легко добавить новые столбцы в любой DataFrame при помощи списка значений:

```
# Добавить столбец с данными о цвете глаз
dataframe["Eyes"] = ["Brown", "Blue"]

# Вывести DataFrame
dataframe
```

```
              Name  Age  Driver   Eyes
0     Jacky Jackson   38    True  Brown
1  Steven Stevenson   25   False   Blue
```

Обсуждение

С pandas мы получаем практически неограниченное количество способов создания DataFrame. В реальности мы редко сталкиваемся с необходимостью создавать пустой DataFrame и наполнять его данными. Вместо этого мы создаем DataFrame из данных, полученных из других источников (например, файл CSV или база данных).

3.2. Получение информации о данных

Задача

Вы хотите узнать какие-либо характеристики DataFrame.

Решение

Одной из простых операций после загрузки является метод head для просмотра первых нескольких строк:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'
```

```
# Загрузить данные
dataframe = pd.read_csv(url)

# Вывести две строки
dataframe.head(2)
```

```
                          Name PClass   Age     Sex  Survived  SexCode
0  Allen, Miss Elisabeth Walton    1st  29.0  female         1        1
1   Allison, Miss Helen Loraine    1st   2.0  female         0        1
```

Мы также можем узнать число строк и столбцов:

```
# Вывести данные формы
dataframe.shape
```

```
(1313, 6)
```

Используя метод describe, мы можем получить наглядные статистические данные числовых столбцов:

```
# Вывести статистические данные
dataframe.describe()
```

```
              Age     Survived      SexCode
count  756.000000  1313.000000  1313.000000
mean    30.397989     0.342727     0.351866
std     14.259049     0.474802     0.477734
min      0.170000     0.000000     0.000000
25%     21.000000     0.000000     0.000000
50%     28.000000     0.000000     0.000000
75%     39.000000     1.000000     1.000000
max     71.000000     1.000000     1.000000
```

Кроме того, мы можем получить более подробную информацию при помощи метода info:

```
# Вывести информацию
dataframe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1313 entries, 0 to 1312
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
```

```
 0   Name      1313 non-null   object
 1   PClass    1313 non-null   object
 2   Age       756 non-null    float64
 3   Sex       1313 non-null   object
 4   Survived  1313 non-null   int64
 5   SexCode   1313 non-null   int64
dtypes: float64(1), int64(2), object(3)
memory usage: 61.7+ KB
```

Обсуждение

После загрузки данных будет полезно узнать больше об их структуре и о том, какую информацию они хранят. В идеальной ситуации мы бы смогли увидеть все данные напрямую. Однако на практике набор реальных данных может состоять из сотен тысяч и миллионов строк и столбцов. Поэтому нам приходится работать с небольшими группами примеров и высчитывать общую статистику набора данных.

В нашем примере мы используем игрушечную базу данных пассажиров "Титаника". При помощи метода `head` мы можем посмотреть несколько первых строк (по умолчанию пять) с данными. Аналогично можно использовать метод `tail` для просмотра последних строк. Посредством атрибута `shape` мы можем узнать, сколько строк и столбцов содержит DataFrame. Для получения основных статистических данных числовых столбцов мы можем применить `describe`. И, наконец, мы можем использовать `info` для отображения более подробной информации о DataFrame, включая типы данных столбцов и индексов, ненулевые значения и объем используемой памяти.

Следует помнить, что общая статистика не всегда отражает полную информацию. К примеру, столбцы `Survived` и `SexCode` рассматриваются в pandas как числовые столбцы, поскольку содержат значения `1` и `0`. Тем не менее в данном случае числовые значения обозначают категорию: `1` в столбце `Survived` означает, что пассажир пережил катастрофу. В результате неправильной трактовки программой некоторые общие статистические данные могут потерять смысл, как в случае со стандартным отклонением в столбце `SexCode` (информация о половой принадлежности пассажиров).

3.3. Срез DataFrame

Задача

Вам нужно выбрать определенное подмножество данных или сделать срез DataFrame.

Решение

Воспользуемся свойствами loc или iloc объекта DataFrame для выбора одной или нескольких строк или значений:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Выбрать первую строку
dataframe.iloc[0]
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: 0, dtype: object
```

Добавив обращение при помощи знака : мы можем задать параметр среза для строк, например, выбрав вторую, третью и четвертую[1]:

```
# Выбрать три строки
dataframe.iloc[1:4]
```

```
                                       Name PClass  ...  Survived SexCode
1                     Allison, Miss Helen Loraine    1st  ...         0       1
2              Allison, Mr Hudson Joshua Creighton    1st  ...         0       0
3  Allison, Mrs Hudson JC (Bessie Waldo Daniels)    1st  ...         0       1
```

Мы также можем не прописывать первый аргумент, чтобы получить срез из строк до нужной нам точки и, например, вывести все строки до четвертой включительно:

```
# Выбрать четыре строки
dataframe.iloc[:4]
```

[1] Нумерация начинается с 0, поэтому выводятся строки и идентификаторами 1, 2 и 3. — *Прим. ред.*

```
                                        Name PClass  ...  Survived SexCode
0                     Allen, Miss Elisabeth Walton    1st  ...         1       1
1                      Allison, Miss Helen Loraine    1st  ...         0       1
2              Allison, Mr Hudson Joshua Creighton    1st  ...         0       0
3  Allison, Mrs Hudson JC (Bessie Waldo Daniels)    1st  ...         0       1
```

При работе с DataFrames нам не обязательно оперировать числовыми индексами. В качестве индекса DataFrame мы можем использовать любое значение, уникальное для каждой строки. К примеру, мы можем задать индекс пассажиров через их имена и использовать их для выбора строк:

```
# Задать индекс
dataframe = dataframe.set_index(dataframe['Name'])

# Вывести информацию строки
dataframe.loc['Allen, Miss Elisabeth Walton']
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: Allen, Miss Elisabeth Walton, dtype: object
```

Обсуждение

В pandas все строки имеют уникальный индекс. По умолчанию в качестве индекса используется числовое значение, указывающее на положение строки в DataFrame, однако это не обязательное условие. Индексы в DataFrame уникальны, а могут быть порядковым номером элемента или иметь буквенно-цифровой строковый (`string`) тип. Для того чтобы выбрать отдельную строку или срез строк, в pandas мы можем воспользоваться одним из двух свойств:

- `loc` используется для обращения к элементу DataFrame по метке (`label`), например, строкового (`string`) типа;
- `iloc` применяется для поиска расположения элемента в DataFrame. К примеру, `iloc[0]` вернет первую строку независимо от того, является ли индекс числом или меткой.

Будет полезно освоить оба свойства, `loc` и `iloc`, для последующего использования в процессе очистки данных.

3.4. Выбор строк на основе условных конструкций

Задача

Вы хотите вывести строки DataFrame в соответствии с определенным условием.

Решение

Эта задача легко решается в pandas. К примеру, нам нужно вывести строки с пассажирками "Титаника":

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Вывести две строки, в которых значение столбца Sex соответствует 'female'
dataframe[dataframe['Sex'] == 'female'].head(2)
```

```
                          Name PClass   Age     Sex  Survived  SexCode
0  Allen, Miss Elisabeth Walton    1st  29.0  female         1        1
1   Allison, Miss Helen Loraine    1st   2.0  female         0        1
```

Давайте подробнее рассмотрим используемый формат. Инструкцией с условием является `dataframe['Sex'] == 'female'`; вложив ее в `dataframe[]`, мы сообщили pandas, что хотим "вывести все строки в DataFrame, в которых значение `dataframe['Sex']` соответствует `'female'`." Таким образом в pandas активируется несколько логических операций (boolean).

Мы также можем задать множественные условия. Например, давайте выведем строки с информацией о пассажирках от 65 лет и старше:

```
# Отфильтровать строки
dataframe[(dataframe['Sex'] == 'female') & (dataframe['Age'] >= 65)]
```

```
                                          Name PClass  ...  Survived SexCode
73  Crosby, Mrs Edward Gifford (Catherine Elizabet...    1st  ...         1       1
```

Обсуждение

Вывод и фильтрация данных в соответствии с условием является одной из основных операций при первичной обработке данных. Обычно нам нужны не все необработанные данные источника, а лишь определенные подгруппы. К примеру, вам могут потребоваться данные магазинов только в конкретных локациях или записи о пациентах определенного возраста.

3.5. Сортировка значений

Задача

Вам необходимо отсортировать информацию в DataFrame в соответствии со значениями столбца.

Решение

Воспользуемся методом sort_values библиотеки pandas:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Отсортировать DataFrame по возрасту и вывести две строки
dataframe.sort_values(by=["Age"]).head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
763	Dean, Miss Elizabeth Gladys (Millvena)	3rd	0.17	female	1	1
751	Danbom, Master Gilbert Sigvard Emanuel	3rd	0.33	male	0	0

Обсуждение

В процессе анализа и изучения данных полезно сортировать DataFrame в соответствии с определенным столбцом или набором столбцов. В качестве аргумента для сортировки DataFrame метод sort_values принимает список столбцов и производит сортировку в порядке расположения названий этих столбцов.

По умолчанию аргумент ascending имеет значение True, таким образом соблюдается порядок сортировки от наименьших к наибольшим значениям. Если нам нужно вывести информацию о пожилых пассажирах, следует указать значение False.

3.6. Замена значений

Задача

Нам нужно заменить значения в DataFrame.

Решение

При помощи метода replace в pandas мы можем легко найти и заменить значения. К примеру, мы можем заменить все значения "female" в столбце Sex на "Woman":

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Заменить значения и вывести две строки
dataframe['Sex'].replace("female", "Woman").head(2)
```

```
0    Woman
1    Woman
Name: Sex, dtype: object
```

Мы также можем заменить несколько значений одновременно:

```
# Заменить "female" и "male" на "Woman" и "Man"
dataframe['Sex'].replace(["female", "male"], ["Woman", "Man"]).head(5)
```

```
0    Woman
1    Woman
2      Man
3    Woman
4      Man
Name: Sex, dtype: object
```

Если мы укажем весь DataFrame вместо одного столбца, то сможем найти и заменить элементы внутри всего объекта DataFrame.

```
# Заменить значения и вывести две строки
dataframe.replace(1, "One").head(2)
```

```
                          Name PClass   Age     Sex Survived SexCode
0  Allen, Miss Elisabeth Walton    1st  29.0  female      One     One
1   Allison, Miss Helen Loraine    1st   2.0  female        0     One
```

Метод `replace` может принимать регулярные выражения:

```
# Заменить значения и вывести две строки
dataframe.replace(r"1st", "First", regex=True).head(2)
```

```
                          Name PClass   Age     Sex  Survived  SexCode
0  Allen, Miss Elisabeth Walton  First  29.0  female         1        1
1   Allison, Miss Helen Loraine  First   2.0  female         0        1
```

Обсуждение

Метод `replace` — это простой инструмент для замены значений, который вместе с тем принимает регулярные выражения.

3.7. Переименование столбцов

Задача

Вы хотите переименовать столбец в DataFrame.

Решение

Переименуем столбцы при помощи метода `rename`:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)
```

```
# Переименовать столбец и вывести две строки
dataframe.rename(columns={'PClass': 'Passenger Class'}).head(2)
```

```
                          Name Passenger Class   Age     Sex  Survived SexCode
0  Allen, Miss Elisabeth Walton             1st  29.0  female         1       1
1   Allison, Miss Helen Loraine             1st  2.00  female         0       1
```

Нужно отметить, что в качестве параметров метод rename может принимать словари, используя которые, мы можем одновременно менять названия нескольких столбцов:

```
# Переименовать столбцы и вывести две строки
dataframe.rename(columns={'PClass': 'Passenger Class', 'Sex': 'Gender'}).head(2)
```

```
                          Name Passenger Class   Age  Gender  Survived SexCode
0  Allen, Miss Elisabeth Walton             1st  29.0  female         1       1
1   Allison, Miss Helen Loraine             1st  2.00  female         0       1
```

Обсуждение

Я отдаю предпочтение использованию метода rename со словарями в качестве параметров для переименования столбцов, поскольку мы можем указать любое количество столбцов. Если нам понадобилось переименовать все столбцы, можно применить специальный фрагмент кода, который создает словарь со старыми названиями столбцов в качестве ключей и пустыми строковыми значениями:

```
# Загрузить библиотеку
import collections

# Создать словарь
column_names = collections.defaultdict(str)

# Создать ключи
for name in dataframe.columns:
    column_names[name]

# Вывести словарь
column_names
```

```
defaultdict(str,
            {'Age': '',
             'Name': '',
```

```
'PClass': '',
'Sex': '',
'SexCode': '',
'Survived': ''})
```

3.8. Нахождение наименьшего, наибольшего и среднего значения, суммы и количества значений

Задача

Вы хотите узнать наименьшее, наибольшее и среднее значение, сумму или количество значений.

Решение

В pandas есть встроенные методы для получения общепринятых статистических показателей, такие как `min`, `max`, `mean`, `sum` и `count`:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Вычислить статистические показатели
print('Maximum:', dataframe['Age'].max())
print('Minimum:', dataframe['Age'].min())
print('Mean:', dataframe['Age'].mean())
print('Sum:', dataframe['Age'].sum())
print('Count:', dataframe['Age'].count())
```

```
Maximum: 71.0
Minimum: 0.17
Mean: 30.397989417989415
Sum: 22980.879999999997
Count: 756
```

Обсуждение

Кроме статистических показателей, получаемых в этом примере, при помощи методов в pandas можно рассчитать дисперсию (var), стандартное отклонение (std), эксцесс (kurt), коэффициент асимметрии (skew), стандартную ошибку среднего (sem), наиболее распространенный элемент последовательности — моду (mode), медиану (median), количество значений и т. д. Более того, эти методы можно применить ко всему объекту DataFrame:

```
# Вывести количество значений
dataframe.count()
Name        1313
PClass      1313
Age          756
Sex         1313
Survived    1313
SexCode     1313
dtype: int64
```

3.9. Поиск уникальных значений

Задача

Вы хотите найти все уникальные значения в столбце.

Решение

Воспользуемся методом unique для вывода массива всех уникальных значений в столбце:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Вывести уникальные значения
dataframe['Sex'].unique()
array(['female', 'male'], dtype=object)
```

При помощи метода value_counts мы можем получить схожий результат с подсчетом количества вхождений каждого значения:

```
# Вывести количество вхождений
dataframe['Sex'].value_counts()
male      851
female    462
Name: Sex, dtype: int64
```

Обсуждение

Методы unique и value_counts полезны при работе со столбцами категориального типа. В таких столбцах очень часто встречаются классы, которые необходимо распределить на этапе первичной обработки данных. В примере набора данных "Титаника" столбец PClass содержит информацию о классе билета пассажира. На "Титанике" было всего три класса билетов; однако при использовании value_counts мы столкнемся с проблемой:

```
# Вывести количество вхождений
dataframe['PClass'].value_counts()

3rd    711
1st    322
2nd    279
*        1
Name: PClass, dtype: int64
```

Как мы и ожидали, практически все пассажиры принадлежат к одному из трех классов, кроме одного, который входит в класс *. В *главе 5* мы более подробно рассмотрим стратегии решения данной проблемы, а пока нужно запомнить, что в столбцах категориального типа нередко могут встречаться "дополнительные" классы, которые не следует игнорировать.

Наконец, если мы хотим просто посчитать количество уникальных значений, используем метод nunique:

```
# Вывести количество уникальных значений
dataframe['PClass'].nunique()
4
```

3.10. Отбор отсутствующих значений

Задача

Вы хотите вывести отсутствующие значения в DataFrame.

Решение

Методы isnull и notnull возвращают логические значения, информирующие о наличии или отсутствии данных:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Выбрать отсутствующие значения и вывести две строки
dataframe[dataframe['Age'].isnull()].head(2)
```

```
                          Name  PClass  Age     Sex  Survived  SexCode
12  Aubert, Mrs Leontine Pauline     1st  NaN  female         1        1
13  Barkworth, Mr Algernon H         1st  NaN    male         1        0
```

Обсуждение

В процессе первичной обработки данных мы постоянно сталкиваемся с отсутствием каких-то данных, однако далеко не все понимают, насколько сложно с этим работать. Для обозначения отсутствующих значений в pandas используется значение NaN (Not a Number) из NumPy, однако нужно отметить, что такое решение не полностью поддерживается pandas. Например, если мы попробуем заменить во всех строках значение male отсутствующим значением, то увидим ошибку:

```
# Попытаться заменить значение на NaN
dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-5682d714f87d> in <module>()
      1 # Attempt to replace values with NaN
 ----> 2 dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)

NameError: name 'NaN' is not defined
---------------------------------------------------------------------------
```

Для того чтобы полноценно воспользоваться значением NaN, необходимо вначале подключить модуль NumPy:

```
# Загрузить библиотеку
import numpy as np

# Заменить значение на NaN
dataframe['Sex'] = dataframe['Sex'].replace('male', np.nan)
```

Зачастую в наборах данных для обозначения отсутствующих данных используются специальные значения, включая NONE, -999 или ... . Функция read_csv в pandas имеет параметр, позволяющий указать необходимое значение для отсутствующих данных:

```
# Загрузить данные и указать отсутствующие значения
dataframe = pd.read_csv(url, na_values=[np.nan, 'NONE', -999])
```

При помощи метода fillna мы можем указать отсутствующие значения столбца. Ниже посредством метода isna мы выведем строки, в которых отсутствует значение Age, а затем заменим его средним значением возраста пассажиров.

```
# Вывести одну строку с пустым значением
null_entry = dataframe[dataframe["Age"].isna()].head(1)

print(null_entry)
```

```
                          Name  PClass  Age     Sex  Survived  SexCode
12  Aubert, Mrs Leontine Pauline     1st  NaN  female         1        1
```

```
# Заменить все значения null средним возрастом пассажиров
```

```
                          Name  PClass        Age     Sex  Survived  SexCode
12  Aubert, Mrs Leontine Pauline     1st  30.397989  female         1        1
```

3.11. Удаление столбца

Задача

Вам нужно удалить столбец из DataFrame.

Решение

Для удаления столбца лучше всего подходит метод drop с параметром axis=1 (т. е. по столбцам):

```
# Загрузить библиотеку
import pandas as pd
```

```
# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Удалить столбец
dataframe.drop('Age', axis=1).head(2)
```

	Name	PClass	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	female	1	1
1	Allison, Miss Helen Loraine	1st	female	0	1

В качестве основного аргумента вы также можете указать список с названиями столбцов и таким образом удалить несколько столбцов одновременно:

```
# Удалить столбцы
dataframe.drop(['Age', 'Sex'], axis=1).head(2)
```

	Name	PClass	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	1	1
1	Allison, Miss Helen Loraine	1st	0	1

Если у столбца нет названия (такое иногда встречается), вы можете удалить его по индексу, используя параметр `dataframe.columns`:

```
# Удалить столбец
dataframe.drop(dataframe.columns[1], axis=1).head(2)
```

	Name	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	29.0	female	1	1
1	Allison, Miss Helen Loraine	2.0	female	0	1

Обсуждение

Метод `drop` является главным методом удаления столбцов. В качестве альтернативы можно использовать `del dataframe['Age']`, однако этот способ не рекомендуется ввиду особенностей его работы внутри библиотеки pandas (мы не будем подробно разбирать детали в данной книге).

Я рекомендую отказаться от аргумента `inplace=True` в pandas. Многие методы pandas имеют параметр `inplace`, который в случае равенства значению `True` взаимодействует с DataFrame напрямую. Это может привести к проблемам при работе с более сложными конвейерами обработки данных, т. к. мы рассматриваем DataFrame как мутирующие объекты (которыми они технически являются). Я же советую рассматривать DataFrame как немутирующий объект. Например:

```
# Создать новый DataFrame
dataframe_name_dropped = dataframe.drop(dataframe.columns[0], axis=1)
```

В данном примере мы не преобразуем имеющийся `dataframe`, а создаем на его измененной основе новую версию с названием `dataframe_name_dropped`. Привычка работать с DataFrame как с немутирующим объектом может избавить вас от множества проблем в будущем.

3.12. Удаление строки

Задача

Вы хотите удалить одну или несколько строк DataFrame.

Решение

Используем логическое условие для создания нового DataFrame, включая строки, которые мы хотим удалить:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Удалить строки и вывести первые три строки
dataframe[dataframe['Sex'] != 'male'].head(3)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
3	Allison, Mrs Hudson JC (Bessie Waldo...	1st	25.00	female	0	1

Обсуждение

Хотя мы можем использовать метод drop (например, dataframe.drop([0, 1], axis=0) для удаления двух первых строк), более практично просто вложить логическое условие внутрь dataframe[]. Таким образом при помощи условий мы можем удалять отдельные строки или, что более вероятно, несколько строк одновременно.

Для того чтобы удалить одну строку при помощи логического условия, мы можем указать соответствие уникальному значению:

```
# Удалить строки и вывести две первые строки
dataframe[dataframe['Name'] != 'Allison, Miss Helen Loraine'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

В качестве значения мы также можем указать индекс строки:

```
# Удалить строки и вывести две первые строки
dataframe[dataframe.index != 0].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

3.13. Удаление дубликатов

Задача

Вам необходимо удалить дублирующие строки из DataFrame.

Решение

Воспользуемся методом drop_duplicates, обращая внимание на параметры:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)
```

```
# Удалить дубликаты и вывести две первые строки
dataframe.drop_duplicates().head(2)
```

```
                          Name  PClass   Age     Sex  Survived  SexCode
0  Allen, Miss Elisabeth Walton     1st  29.0  female         1        1
1   Allison, Miss Helen Loraine     1st   2.0  female         0        1
```

Обсуждение

Внимательный читатель заметит, что при помощи данного решения мы не удалили ни одной строки:

```
# Вывести число строк
print("Number Of Rows In The Original DataFrame:", len(dataframe))
print("Number Of Rows After Deduping:", len(dataframe.drop_duplicates()))
```

```
Number Of Rows In The Original DataFrame: 1313
Number Of Rows After Deduping: 1313
```

Это связано с тем, что метод drop_duplicates по умолчанию удаляет строки только при их полном совпадении во всех столбцах. Поскольку каждая строка в нашем DataFrame уникальна, все строки сохранятся. На практике же нам зачастую требуется проверять наличие дублирующих строк в подгруппе столбцов. Для этого нам пригодится параметр subset:

```
# Удалить дубликаты
dataframe.drop_duplicates(subset=['Sex'])
```

```
                                Name  PClass   Age     Sex  Survived  SexCode
0          Allen, Miss Elisabeth Walto     1st  29.0  female         1        1
2  Allison, Mr Hudson Joshua Creighton     1st  30.0    male         0        0
```

Давайте разберем результат вывода функции: мы сообщили drop_duplicates параметр, по которому дублирование значений строк нужно искать и удалять только в столбце Sex. В результате в нашем DataFrame остались лишь две строки: одна со значением woman и другая со значением man. Вы можете задаться вопросом: почему в результате использования drop_duplicates остались именно эти строки? По умолчанию drop_duplicates сохраняет первое вхождение строки и удаляет все последующие дубликаты. Мы можем изменить такое поведение при помощи параметра keep:

```
# Удалить дубликаты
dataframe.drop_duplicates(subset=['Sex'], keep='last')
```

```
                    Name PClass   Age     Sex  Survived  SexCode
1307  Zabour, Miss Tamini    3rd   NaN  female         0        1
1312       Zimmerman, Leo    3rd  29.0    male         0        0
```

Родственный метод duplicated возвращает набор логических значений, сообщающих о наличии или отсутствии дублирующихся строк. Эта опция информативнее простого удаления дубликатов:

```
dataframe.duplicated()
```

```
0       False
1       False
2       False
3       False
4       False
        ...
1308    False
1309    False
1310    False
1311    False
1312    False
Length: 1313, dtype: bool
```

3.14. Группировка строк по значениям

Задача

Вы хотите сгруппировать строки в соответствии с их общими значениями.

Решение

Одним из наиболее мощных методов библиотеки pandas является groupby:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)
```

```
# Сгруппировать строки по значениям столбца Sex
# и вычислить среднее значение каждой группы
dataframe.groupby('Sex').mean(numeric_only=True)
```

```
              Age  Survived  SexCode
Sex
female  29.396424  0.666667      1.0
male    31.014338  0.166863      0.0
```

Обсуждение

С методом groupby первичная обработка данных придает им осмысленную форму. Если у нас есть DataFrame, в котором каждая строка означает человека или событие, мы, весьма вероятно, захотим сгруппировать их в соответствии с каким-либо критерием и подсчитать статистику. В качестве примера можно представить DataFrame, строки которого содержат информацию об отдельных продажах сети ресторанов, и нам нужно рассчитать итоговые показатели продаж каждого ресторана. В таком случае мы можем объединить строки каждого ресторана и получить суммарные значения каждой группы.

При работе с groupby новички часто пишут такой код и получают непонятный результат:

```
# Сгруппировать строки
dataframe.groupby('Sex')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x10efacf28>
```

Это не тот результат, который мы рассчитываем получить. В чем же причина? Так происходит потому, что метод groupby необходимо использовать с другой операцией, которая применяется к каждой группе, например с подсчетом совокупной статистики (среднее значение, медиана, сумма). Говоря про группирование, мы подразумеваем группировку по полу человека, однако этого недостаточно. Для того чтобы результат группировки был полезен, нам необходимо выбрать параметр и затем применить метод к каждой группе:

```
# Сгруппировать и посчитать строки
dataframe.groupby('Survived')['Name'].count()
```

```
Survived
0    863
1    450
Name: Name, dtype: int64
```

Как вы могли заметить, мы добавили Name после groupby. Причина заключается в том, что определенные статистические данные имеют смысл при работе только с

конкретными типами данных. Например, подсчет среднего возраста по полу человека имеет смысл, в то время как подсчет общего значения возраста — нет. В нашем случае мы группируем данные по параметру выживаемости и высчитываем количество имен (например, пассажиров) в каждой группе.

Мы также можем произвести группировки по первому столбцу и затем применить группировку по второму столбцу:

```
# Сгруппировать строки и вычислить среднее значение
dataframe.groupby(['Sex','Survived'])['Age'].mean()
```

```
Sex     Survived
female  0          24.901408
        1          30.867143
male    0          32.320780
        1          25.951875
Name: Age, dtype: float64
```

3.15. Группировка строк по времени

Задача

Нам необходимо сгруппировать отдельные строки в соответствии с временны́ми промежутками.

Решение

Воспользуемся методом `resample` для группировки строк по временны́м отрезкам:

```
# Загрузить библиотеки
import pandas as pd
import numpy as np

# Задать временной отрезок
time_index = pd.date_range('06/06/2017', periods=100000, freq='30S')

# Создать DataFrame
dataframe = pd.DataFrame(index=time_index)

# Создать столбец со случайными значениями
dataframe['Sale_Amount'] = np.random.randint(1, 10, 100000)

# Сгруппировать строки по неделям и вычислить сумму за неделю
dataframe.resample('W').sum()
```

```
            Sale_Amount
2017-06-11        86486
2017-06-18       100625
2017-06-25       100823
2017-07-02       100564
2017-07-09       101235
2017-07-16        10297
```

Обсуждение

В нашем стандартном примере с "Титаником" нет столбца с информацией о времени, поэтому в качестве примера для этой задачи создадим простой DataFrame, в котором каждая строка соответствует одной продаже. Для каждой продажи указываются дата, время и сумма в долларах (эти данные не очень реалистичны, поскольку интервалы между продажами составляют 30 секунд, а суммы представляют целые значения, однако это несущественное упрощение).

Сырые данные имеют следующий вид:

```
# Вывести три строки
dataframe.head(3)
```

```
                     Sale_Amount
2017-06-06 00:00:00            7
2017-06-06 00:00:30            2
2017-06-06 00:01:00            7
```

Обратите внимание, что индексом в DataFrame являются дата и время покупки; это особенность метода `resample` — принимать значения индекса в формате даты-времени.

Метод `resample` позволяет нам группировать строки в соответствии с различными периодами и рассчитывать статистические данные для каждой группы:

```
# Разделить строки на двухнедельные периоды и вычислить среднее значение
dataframe.resample('2W').mean()
```

```
            Sale_Amount
2017-06-11     5.001331
2017-06-25     5.007738
2017-07-09     4.993353
2017-07-23     4.950481
```

```
# Сгруппировать строки по месяцам и посчитать их количество
dataframe.resample('M').count()
```

```
            Sale_Amount
2017-06-30        72000
2017-07-31        28000
```

Как мы видим из примеров, в обоих случаях индекс даты-времени является датой, несмотря на то что группирование происходило по неделям и месяцам соответственно. При разделении по времени по умолчанию метод `resample` возвращает крайнюю правую (последнюю) временну́ю метку. Мы можем менять это условие при помощи параметра `label`:

```
# Сгруппировать строки по месяцам и посчитать их количество
dataframe.resample('M', label='left').count()
```

```
            Sale_Amount
2017-05-31        72000
2017-06-30        28000
```

Дополнительные материалы для чтения

- Список функций для операций с датой-временем в библиотеке pandas (**https://oreil.ly/BURbR**).

3.16. Агрегатные операции и статистика

Задача

Вам требуется применить агрегатную функцию к каждому столбцу (или набору столбцов) в DataFrame.

Решение

Для этого в pandas есть метод `agg`. С его помощью мы можем легко узнать наименьшее значение каждого столбца:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'
```

```
# Загрузить данные
dataframe = pd.read_csv(url)

# Получить наименьшее значение каждого столбца
dataframe.agg("min")
```

```
Name        Abbing, Mr Anthony
PClass                       *
Age                       0.17
Sex                     female
Survived                     0
SexCode                      0
dtype: object
```

Иногда нам необходимо применить определенную функцию к конкретному набору столбцов:

```
# Вывести среднее значение Age и наименьшее и наибольшее значения SexCode
dataframe.agg({"Age":["mean"], "SexCode":["min", "max"]})
```

```
            Age  SexCode
mean  30.397989      NaN
min         NaN      0.0
max         NaN      1.0
```

Кроме того, мы можем применять агрегатные функции к группам для получения более конкретных статистических данных:

```
# Вывести количество людей, переживших и не переживших катастрофу
dataframe.groupby(
    ["PClass","Survived"]).agg({"Survived":["count"]}
  ).reset_index()
```

```
PClass  Survived  Count
     0       *  0      1
     1     1st  0    129
     2     1st  1    193
     3     2nd  0    160
     4     2nd  1    119
     5     3rd  0    573
     6     3rd  1    138
```

Обсуждение

Агрегатные функции могут быть особенно полезны при анализе и изучении данных, поскольку позволяют получить больше информации о данных и связях между переменными. В процессе группировки данных и работы с агрегированной статистикой мы выявляем паттерны, которые могут пригодиться для машинного обучения или связанных проектов. Хотя графики и диаграммы весьма удобны, полезно иметь такую узконаправленную статистическую информацию для лучшего понимания данных.

Дополнительные материалы для чтения

- Документация по `agg` библиотеки pandas (**https://oreil.ly/5xing**).

3.17. Обход столбца в цикле

Задача

Вы хотите запустить итерацию для каждого элемента столбца с применением каких-либо действий.

Решение

Поскольку столбцы в pandas схожи с другими последовательностями в Python, мы можем применить к ним циклы при помощи инструментов стандартного синтаксиса Python:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Вывести два первых имени заглавными буквами
for name in dataframe['Name'][0:2]:
    print(name.upper())
```

```
ALLEN, MISS ELISABETH WALTON
ALLISON, MISS HELEN LORAINE
```

Обсуждение

Кроме циклов (так называемых циклов for) мы можем использовать *списковое включение* (или конструкцию включения в список):

```
# Вывести два первых имени заглавными буквами
[name.upper() for name in dataframe['Name'][0:2]]
```

```
['ALLEN, MISS ELISABETH WALTON', 'ALLISON, MISS HELEN LORAINE']
```

Несмотря на соблазн применить циклы for, в Python более естественно использовать методы pandas, которые мы рассмотрим в *рецепте 3.18*.

3.18. Применение функции ко всем элементам столбца

Задача

Вам нужно применить какую-либо функцию ко всем элементам столбца.

Решение

Воспользуемся методом apply, чтобы применить встроенную или написанную нами функцию к каждому элементу столбца:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Создать функцию
def uppercase(x):
    return x.upper()

# Применить функцию и вывести две строки
dataframe['Name'].apply(uppercase)[0:2]
```

```
0    ALLEN, MISS ELISABETH WALTON
1    ALLISON, MISS HELEN LORAINE
Name: Name, dtype: object
```

Обсуждение

Метод apply очень помогает при очистке и первичной обработке данных. Нам часто приходится писать функции для каких-либо полезных операций (разделить имя и фамилию, преобразовать строку в число и т. д.) и применять их к каждому элементу.

3.19. Применение функции к группам

Задача

Вы сгруппировали строки при помощи метода groupby и хотите применить функцию к каждой группе.

Решение

Используем вместе методы groupby и apply:

```
# Загрузить библиотеку
import pandas as pd

# Добавить URL-адрес
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Сгруппировать строки и применить функцию к группе
dataframe.groupby('Sex').apply(lambda x: x.count())
```

	Name	PClass	Age	Sex	Survived	SexCode
Sex						
female	462	462	288	462	462	462
male	851	851	468	851	851	851

Обсуждение

Мы уже встречали метод apply в *рецепте 3.18*. Его удобно использовать, когда необходимо применить функцию к группам. Объединив методы groupby и apply, мы можем вычислить нужные нам статистические данные и применить функцию к каждой группе отдельно.

3.20. Конкатенация структур данных DataFrame

Задача

Вы хотите произвести конкатенацию (concatenate) двух DataFrame.

Решение

Используем функцию `concat` со значением `axis=0` для конкатенации по строкам:

```
# Загрузить библиотеку
import pandas as pd

# Создать DataFrame
data_a = {'id': ['1', '2', '3'],
          'first': ['Alex', 'Amy', 'Allen'],
          'last': ['Anderson', 'Ackerman', 'Ali']}
dataframe_a = pd.DataFrame(data_a, columns = ['id', 'first', 'last'])

# Создать DataFrame
data_b = {'id': ['4', '5', '6'],
          'first': ['Billy', 'Brian', 'Bran'],
          'last': ['Bonder', 'Black', 'Balwner']}
dataframe_b = pd.DataFrame(data_b, columns = ['id', 'first', 'last'])

# Выполнить конкатенацию по строкам
pd.concat([dataframe_a, dataframe_b], axis=0)
```

```
  id  first     last
0  1   Alex  Anderson
1  2    Amy  Ackerman
2  3  Allen       Ali
0  4  Billy    Bonder
1  5  Brian     Black
2  6   Bran   Balwner
```

Для того чтобы произвести конкатенацию по столбцам, используйте значение `axis=1`:

```
# Выполнить конкатенацию по столбцам
pd.concat([dataframe_a, dataframe_b], axis=1)
```

```
  id  first      last id  first     last
0  1   Alex  Anderson  4  Billy   Bonder
1  2    Amy  Ackerman  5  Brian    Black
2  3  Allen       Ali  6   Bran  Balwner
```

Обсуждение

Понятие конкатенации (concatenating) не часто встречается за пределами компьютерных наук и программирования, поэтому не беспокойтесь, если вы впервые узнали это слово. Под *конкатенацией* мы понимаем склеивание двух объектов. В нашем решении мы склеили две небольшие структуры данных DataFrame, а при помощи параметра `axis` сообщили, как мы хотим их расположить — рядом или один над другим.

3.21. Слияние структур данных DataFrame

Задача

Вы хотите осуществить слияние (merge) структур данных DataFrame.

Решение

Для внутреннего соединения (inner join) данных воспользуемся функцией `merge`, указав в качестве параметра `on` столбец для слияния:

```
# Загрузить библиотеку
import pandas as pd

# Создать DataFrame
employee_data = {'employee_id': ['1', '2', '3', '4'],
                 'name': ['Amy Jones', 'Allen Keys', 'Alice Bees',
                 'Tim Horton']}
dataframe_employees = pd.DataFrame(employee_data, columns = ['employee_id', 'name'])

# Создать DataFrame
sales_data = {'employee_id': ['3', '4', '5', '6'],
              'total_sales': [23456, 2512, 2345, 1455]}
dataframe_sales = pd.DataFrame(sales_data, columns = ['employee_id',
                                                      'total_sales'])

# Объединить данные одного DataFrame с другим
pd.merge(dataframe_employees, dataframe_sales, on='employee_id')
```

```
   employee_id        name  total_sales
0            3  Alice Bees        23456
1            4  Tim Horton         2512
```

Функция merge по умолчанию используется для внутреннего соединения. Для того чтобы провести внешнее соединение (outer join), укажем параметр how:

```
# Объединить данные одного DataFrame с другим
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='outer')
```

```
   employee_id        name  total_sales
0            1   Amy Jones          NaN
1            2  Allen Keys          NaN
2            3  Alice Bees      23456.0
3            4  Tim Horton       2512.0
4            5         NaN       2345.0
5            6         NaN       1455.0
```

При помощи этого параметра мы также можем задать внешнее правое (right outer join) или левое (left outer join) соединение:

```
# Объединить данные одного DataFrame с другим
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='left')
```

```
   employee_id        name  total_sales
0            1   Amy Jones          NaN
1            2  Allen Keys          NaN
2            3  Alice Bees      23456.0
3            4  Tim Horton       2512.0
```

Кроме того, мы можем указать название столбца для слияния в каждом DataFrame:

```
# Объединить данные одного DataFrame с другим
pd.merge(dataframe_employees,
         dataframe_sales,
         left_on='employee_id',
         right_on='employee_id')
```

```
   employee_id        name  total_sales
0            3  Alice Bees      23456.0
1            4  Tim Horton       2512.0
```

Если нам нужно объединить не столбцы, а индексы каждого DataFrame, мы можем заменить параметры `left_on` и `right_on` на `left_index=True` и `right_index=True`.

Обсуждение

При работе с реальными данными они не всегда предоставляются в виде цельного набора. Нам часто приходится работать с несвязанными наборами данных, полученными в виде запросов или файлов из различных баз данных. Для того чтобы собрать их вместе, нам нужно загрузить каждый запрос или файл в pandas, преобразовав их в набор структур данных DataFrame и объединив их в один DataFrame.

Этот процесс должен быть знаком тем, кто уже работал с SQL — языком программирования, который часто используется для операций слияния (*joins*). Несмотря на отличия конкретных параметров при использовании в pandas, основные паттерны остаются неизменными во всех языках и инструментах.

В операциях слияния следует выделять три аспекта. Во-первых, нужно указать, какие две структуры DataFrame мы хотим объединить. В нашем примере это `dataframe_employees` и `dataframe_sales`. Во-вторых, необходимо указать названия столбцов для слияния, другими словами, общие значения столбцов двух структур DataFrame. К примеру, в нашем случае это общий столбец `employee_id`. Для слияния двух структур мы сопоставим значения каждого столбца `employee_id`. Если оба столбца имеют одинаковое название, мы можем использовать параметр `on`. В противном случае мы можем воспользоваться параметрами `left_on` и `right_on`.

Что же означают левый и правый DataFrame? В операции слияния `merge` левый DataFrame указывается первым, а правый DataFrame указывается вторым. Аналогичная структура используется при настройках нужных нам параметров далее.

Последний и наиболее сложный для некоторых аспект — тип операций слияния. Он указывается при помощи параметра `how`. Функция `merge` поддерживает четыре основных типа слияния.

Внутреннее соединение (inner).

Возвращает только совпадающие строки обеих структур DataFrame (например, возвращает любую строку, которой значение `employee_id` совпадает в `dataframe_employees` и `dataframe_sales`).

Внешнее соединение (outer).

Возвращает все строки обеих структур DataFrame. Если строка присутствует только в одном DataFrame, в поле отсутствующего значения указывается `NaN` (например, возвращает все строки `dataframe_employee` и `dataframe_sales`).

Левое соединение (left).

Возвращает все строки левого DataFrame и только соответствующие им строки правого DataFrame. В поле отсутствующего значения указывается `NaN` (например, возвращает все строки `dataframe_employees` и некоторые строки `dataframe_sales` с совпадающими значениями `employee_id`).

Правое соединение (right).

Возвращает все строки правого DataFrame и только соответствующие им строки левого DataFrame. В поле отсутствующего значения указывается `NaN` (например, возвращает все строки `dataframe_sales` и некоторые строки `dataframe_employees` с совпадающими значениями `employee_id`).

Если что-то из описанного выше осталось непонятным, рекомендую попробовать разные параметры `how` в вашем коде, чтобы наглядно увидеть разницу результатов слияния.

Дополнительные материалы для чтения

- A Visual Explanation of SQL Joins ("Визуальное представление соединений SQL"; **https://oreil.ly/J1A4u**).
- Merge, join, concatenate and compare ("Объединение, соединения, конкатенация и сравнение", документация pandas; **https://oreil.ly/eNalU**).

ГЛАВА 4

Управление количественными данными

Введение

Количественные данные используются при измерении чего-либо, например размера класса, ежемесячных продаж или количества студентов. Такие данные логично представить в числовом формате (например, 29 студентов, сумма продаж в размере 529 392 рублей). В этой главе мы разберем различные способы преобразования сырых количественных данных в признаки (feature), созданные для использования в алгоритмах машинного обучения.

4.1. Масштабирование признака

Задача

Вам нужно изменить масштаб значений количественного признака, чтобы поместить их между двумя значениями.

Решение

Используем класс `MinMaxScaler` библиотеки scikit-learn для изменения масштаба массива признаков:

```
# Загрузить библиотеки
import numpy as np
from sklearn import preprocessing

# Создать признак
feature = np.array([[-500.5],
                    [-100.1],
                    [0],
                    [100.1],
                    [900.9]])
```

```
# Создать масштабатор
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

# Масштабировать признак
scaled_feature = minmax_scale.fit_transform(feature)

# Вывести признак
scaled_feature
array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

Обсуждение

Масштабирование является одной из основных процедур предварительной обработки данных в машинном обучении. Для многих описываемых далее в книге алгоритмов масштаб всех признаков будет одинаковый — обычно от 0 до 1 или от –1 до 1. Существует несколько способов масштабирования, однако самым простым является *масштабирование в заданном диапазоне* (или *минимаксное шкалирование* — min-max scaling). Такое масштабирование осуществляется при помощи указания наименьшего и наибольшего значений, в пределах которых изменяются значения признака. Для этого используется следующая формула:

$$x_i' = \frac{x_i - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})},$$

где $\mathbf{x}$ — вектор признака; x_i — отдельный элемент признака x; x_i' — масштабированный элемент. Вывод массива в нашем примере показывает, что признак был успешно масштабирован в соответствии с диапазоном от 0 до 1:

```
array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

Класс `MinMaxScaler` в scikit-learn имеет два варианта масштабирования признака. В первом случае мы можем использовать метод `fit` для расчета наименьшего и наибольшего значений признака, а затем применить метод `transform` для масштабирования признака. Во втором случае можно использовать метод `fit_transform` для одновременного осуществления двух операций. С точки зрения математики оба способа эквивалентны, однако с практической точки зрения удобнее разделять опе-

рации, поскольку это позволяет нам применять одинаковые трансформации к различным *наборам* данных.

Дополнительные материалы для чтения

- Feature scaling ("Масштабирование признаков", Википедия; **https://oreil.ly/f2WiM**).
- Sebastian Raschka "About Feature Scaling and Normalization" (Себастьян Рашка "О шкалировании и нормализации признаков"; **https://oreil.ly/Da0AH**).

4.2. Стандартизация признака

Задача

Вы хотите преобразовать признак таким образом, чтобы среднее значение было 0, а стандартное отклонение равнялось 1.

Решение

Для преобразования воспользуемся классом `StandardScaler` библиотеки scikit-learn:

```
# Загрузить библиотеки
import numpy as np
from sklearn import preprocessing

# Создать признак
x = np.array([[-1000.1],
              [-200.2],
              [500.5],
              [600.6],
              [9000.9]])

# Создать масштабатор
scaler = preprocessing.StandardScaler()

# Преобразовать признак
standardized = scaler.fit_transform(x)

# Вывести признак
standardized
array([[-0.76058269],
```

```
[-0.54177196],
[-0.35009716],
[-0.32271504],
[ 1.97516685]])
```

Обсуждение

В качестве альтернативы для масштабирования в заданном диапазоне из *рецепта 4.1* мы можем масштабировать признаки при помощи стандартизации, что позволит равномерно распределить их значения. Для этого мы преобразуем данные таким образом, что их среднее значение $\bar{x}$ становится равным 0, а стандартное отклонение σ — равным 1. В частности, каждый элемент признака преобразуется таким образом, что:

$$x_i' = \frac{x_i - \bar{x}}{\sigma},$$

где x_i' — наша стандартизованная форма x_i. Преобразованный признак означает количество стандартных отклонений, на которые исходное значение отстоит от среднего значения признака (в статистике это называется *стандартизированной оценкой*, или *z-оценкой* (z-score)).

Стандартизация является ключевым методом масштабирования в машинном обучении и, на мой взгляд, используется чаще минимаксного шкалирования. Однако это зависит от алгоритмов обучения. К примеру, для метода главных компонент (principal component analysis, PCA) лучше подходит стандартизация, тогда как масштабирование в заданном диапазоне часто рекомендуется для нейронных сетей (мы рассмотрим оба алгоритма далее в этой книге). Я рекомендую по умолчанию использовать стандартизацию и прибегать к другим методам только при необходимости.

В нашем примере мы можем наблюдать результат стандартизации на основе показателей среднего значения и стандартного отклонения:

```
# Вывести среднее значение и стандартное отклонение
print("Mean:", round(standardized.mean()))
print("Standard deviation:", standardized.std())
```

```
Mean: 0.0
Standard deviation: 1.0
```

Наличие существенных выпадающих значений в наших данных может затронуть показатели среднего значения и дисперсии и тем самым негативно сказаться на стандартизации. В таком случае может помочь масштабирование признака при помощи медианы и квантильного размаха. В scikit-learn для этого есть класс `RobustScaler`:

```
# Создать масштабатор
robust_scaler = preprocessing.RobustScaler()
```

```
# Преобразовать признак
robust_scaler.fit_transform(x)
array([[ -1.87387612],
       [ -0.875     ],
       [  0.        ],
       [  0.125     ],
       [ 10.61488511]])
```

4.3. Нормализация наблюдаемых данных

Задача

Мы хотим масштабировать значения признака в наблюдаемых данных с целью привести их к некоторой единичной норме (общая длина равняется 1).

Решение

Воспользуемся классом `Normalizer` с параметром `norm`:

```
# Загрузить библиотеку
import numpy as np
from sklearn.preprocessing import Normalizer

# Создать матрицу признаков
features = np.array([[0.5, 0.5],
                     [1.1, 3.4],
                     [1.5, 20.2],
                     [1.63, 34.4],
                     [10.9, 3.3]])

# Создать нормализатор
normalizer = Normalizer(norm="l2")

# Преобразовать матрицу признака
normalizer.transform(features)
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Обсуждение

Многие способы масштабирования (например, масштабирование в заданном диапазоне и стандартизация) подразумевают работу с признаками; тем не менее мы также можем масштабировать отдельные наблюдения. При помощи класса `Normalizer` отдельные наблюдаемые значения масштабируются для соответствия некоторой шкале (сумма их длин равна 1). Этот тип масштабирования часто используется при наличии большого количества эквивалентных признаков (например, при классификации текста, где каждое слово и группа *n* слов являются признаком).

В `Normalizer` есть три параметра нормы (`norm`); в качестве аргумента по умолчанию используется евклидова норма (которую часто обозначают как L2):

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2},$$

где x — отдельное наблюдение; x_n — значение этого наблюдения для признака n.

```
# Преобразовать матрицу признаков
features_l2_norm = Normalizer(norm="l2").transform(features)

# Вывести матрицу признаков
features_l2_norm
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

В качестве альтернативы мы можем воспользоваться манхэттенской нормой (L1):

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|.$$

```
# Преобразовать матрицу признаков
features_l1_norm = Normalizer(norm="l1").transform(features)

# Вывести матрицу признаков
features_l1_norm
array([[ 0.5       ,  0.5       ],
       [ 0.24444444,  0.75555556],
       [ 0.06912442,  0.93087558],
       [ 0.04524008,  0.95475992],
       [ 0.76760563,  0.23239437]])
```

Условно норму L2 можно представить как расстояние между двумя точками в Нью-Йорке, пролетаемое птицей (т. е. по прямой линии), а L1 в таком случае будет рас-

стоянием, которое преодолевает идущий по улице человек (один квартал на север, один квартал на восток, один квартал на север, один квартал на восток и т. д.), поэтому оно имеет название "манхэттенская норма", или "метрика такси".

На практике при использовании параметра `norm="l1"` наблюдаемые значения масштабируются таким образом, чтобы их сумма равнялась 1:

```
# Вывести сумму
print("Sum of the first observation\'s values:",
    features_l1_norm[0, 0] + features_l1_norm[0, 1])
Sum of the first observation's values: 1.0
```

4.4. Генерирование полиномиальных и взаимодействующих признаков

Задача

Вам нужно создать полиномиальные (polynomial) и взаимодействующие (interaction) признаки.

Решение

Полиномиальные и взаимодействующие признаки можно создать вручную или воспользоваться встроенным методом scikit-learn:

```
# Загрузить библиотеку
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Создать матрицу признаков
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Создать объект PolynomialFeatures
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)

# Создать полиномиальные признаки
polynomial_interaction.fit_transform(features)
array([[ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.]])
```

Параметр degree определяет максимальную степень полинома. К примеру, если указать degree=2, мы получим новые признаки путем возведения в квадрат имеющихся значений:

$$x_1, x_2, x_1^2, x_1x_2, x_2^2,$$

в то время как с degree=3 мы получим новые значения при возведении во вторую и третью степень:

$$x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3.$$

Более того, в PolynomialFeatures по умолчанию включены взаимодействующие признаки:

$$x_1x_2.$$

Указав True для аргумента interaction_only, мы ограничим создаваемые признаки, оставив только взаимодействующие:

```
interaction = PolynomialFeatures(degree=2,
                interaction_only=True, include_bias=False)

interaction.fit_transform(features)
array([[ 2.,  3.,  6.],
       [ 2.,  3.,  6.],
       [ 2.,  3.,  6.]])
```

Обсуждение

Мы часто используем полиномиальные признаки, когда хотим подчеркнуть наличие нелинейных связей между признаками и целевым значением. Например, мы можем предположить, что связь между возрастом и вероятностью развития серьезного заболевания не является постоянной величиной, а увеличивается по мере изменения возраста. Мы можем выразить этот непостоянный эффект через признак x, создавая его производные формы высшего порядка (x^2, x^3 и т. д.).

Кроме того, мы часто оказываемся в ситуации, когда эффект одного признака зависит от другого. Представим, что нам нужно понять, является ли кофе сладким или нет, и для этого у нас есть два признака: (1) кофе размешали или не размешали и (2) в кофе добавили или не добавили сахар. По отдельности эти признаки не имеют смысла, однако в совокупности они позволяют определить сладость кофе. Другими словами, кофе является сладким, если в него добавили сахар и размешали. Влияние каждого признака на целевой показатель (сладость) зависит от других признаков. Мы можем передать это взаимоотношение при помощи добавления взаимодействующего признака, который суммарно выводится из отдельных признаков.

4.5. Преобразование признаков

Задача

Вам необходимо осуществить нестандартное преобразование одного или нескольких признаков.

Решение

В scikit-learn мы можем применить функцию к набору признаков при помощи класса `FunctionTransformer`:

```
# Загрузить библиотеку
import numpy as np
from sklearn.preprocessing import FunctionTransformer

# Создать матрицу признаков
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Определить простую функцию
def add_ten(x: int) -> int:
    return x + 10

# Создать преобразователь
ten_transformer = FunctionTransformer(add_ten)

# Преобразовать матрицу признаков
ten_transformer.transform(features)
array([[12, 13],
       [12, 13],
       [12, 13]])
```

В pandas мы можем создать аналогичный преобразователь при помощи метода `apply`:

```
# Загрузить библиотеку
import pandas as pd

# Создать DataFrame
df = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Применить функцию
df.apply(add_ten)
```

```
   feature_1  feature_2
0         12         13
1         12         13
2         12         13
```

Обсуждение

У нас часто возникает необходимость произвести нестандартные преобразования одного или нескольких признаков. Например, нам может понадобиться создать признак, который будет естественным образом отражать значения другого признака. Для этого можно создать функцию и затем применить ее к признакам при помощи класса `FunctionTransformer` в scikit-learn или метода `apply` в pandas. В данном примере мы использовали простую функцию `add_ten`, прибавляющую 10 к каждому значению, однако ничего не мешает нам создавать более сложные функции.

4.6. Обнаружение выбросов

Задача

Вы хотите выявить экстремальные наблюдаемые значения (extreme observations).

Решение

Обнаружение выбросов (outliers), к сожалению, скорее относится к искусству, чем к науке. Предположим, что данные распределяются нормально (normally distributed), и очертим эллипс вокруг них, где наблюдаемые данные внутри эллипса будут неявным выбросом (inlier) со значением `1`, а наблюдаемые данные вне эллипса будут выбросом (outlier) со значением `-1`:

```
# Загрузить библиотеку
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs

# Создать искусственные данные
features, _ = make_blobs(n_samples = 10,
                         n_features = 2,
                         centers = 1,
                         random_state = 1)

# Заменить первые наблюдаемые значения на экстремальные
features[0,0] = 10000
features[0,1] = 10000
```

```
# Создать детектор
outlier_detector = EllipticEnvelope(contamination=.1)

# Обучить детектор
outlier_detector.fit(features)

# Предугадать выбросы
outlier_detector.predict(features)
array([-1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

В этих массивах значения `-1` обозначают выбросы, а значения `1` — неявные выбросы. Главным ограничением представленного подхода является необходимость задавать параметр `contamination`, другими словами, пропорцию наблюдаемых данных к выбросам, значение которой мы не знаем. Параметр `contamination` следует рассматривать как показатель чистоты данных. Если мы ожидаем, что выбросов немного, то мы укажем небольшое значение параметра. Если же мы полагаем, что выбросов больше, то и значение этого параметра мы укажем высокое.

Вместо взгляда на все наблюдаемые значения в целом мы можем сосредоточиться на отдельных признаках и определить их экстремальные значения при помощи межквартильного размаха (interquartile range, IQR):

```
# Создать один признак
feature = features[:,0]

# Создать функцию, возвращающую индекс выброса
def indicies_of_outliers(x: int) -> np.array(int):
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# Запустить функцию
indicies_of_outliers(feature)
(array([0]),)
```

Межквартильный размах (IQR) — это разность между третьим и первым квартилями набора данных. Вы можете представить межквартильный размах как большой набор данных, где выбросы наблюдаемых значений будут находиться в стороне от основного скопления данных. К выбросам обычно относят все значения, которые на 1,5 IQR меньше первого квартиля или на 1,5 IQR больше третьего квартиля.

Обсуждение

В настоящее время не существует единого верного способа обнаружения выбросов. В нашем распоряжении набор методов со своими преимуществами и недостатками. Для достижения лучшего результата нам стоит попробовать различные способы (например, класс `EllipticEnvelope` и обнаружение на основе IQR) и оценить полученный результат.

В идеальном варианте нам следует также обратить внимание на наблюдаемые значения, отмеченные как выбросы, и попробовать понять их. К примеру, если у нас есть база данных жилых домов, в которой один из признаков означает количество комнат, мы можем предположить, что выброс со значением в 100 комнат может на самом деле являться не домом, а гостиницей с неправильной классификацией.

Дополнительные материалы для чтения

- Three ways to detect outliers ("Три способа обнаружения выбросов" (блог) и исходный код функции межквартильного размаха IQR, используемой в данном рецепте; **https://oreil.ly/wlwmH**).

4.7. Управление выбросами

Задача

Вы обнаружили выбросы в ваших данных и хотите уменьшить их влияние на распространение данных.

Решение

Существуют три основные стратегии управления выбросами. Во-первых, мы можем удалить их:

```
# Загрузить библиотеку
import pandas as pd

# Создать DataFrame
houses = pd.DataFrame()
houses['Price'] = [534433, 392333, 293222, 4322032]
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

# Отфильтровать наблюдения
houses[houses['Bathrooms'] < 20]
```

```
   Price  Bathrooms  Square_Feet
0  534433        2.0         1500
1  392333        3.5         2500
2  293222        2.0         1500
```

Во-вторых, мы можем пометить их как выбросы, добавив столбец `Outlier` (выброс) в качестве признака:

```
# Загрузить библиотеку
import numpy as np

# Создать булевы признаки
houses["Outlier"] = np.where(houses["Bathrooms"] < 20, 0, 1)

# Вывести данные
houses
```

```
     Price  Bathrooms  Square_Feet  Outlier
0   534433        2.0         1500        0
1   392333        3.5         2500        0
2   293222        2.0         1500        0
3  4322032      116.0        48000        1
```

Наконец, мы можем преобразовать признак, чтобы ослабить влияние выброса:

```
# Вычислить логарифм признака
houses["Log_Of_Square_Feet"] = [np.log(x) for x in houses["Square_Feet"]]

# Вывести данные
houses
```

```
     Price  Bathrooms  Square_Feet  Outlier  Log_Of_Square_Feet
0   534433        2.0         1500        0            7.313220
1   392333        3.5         2500        0            7.824046
2   293222        2.0         1500        0            7.313220
3  4322032      116.0        48000        1           10.778956
```

Обсуждение

Как и в случае с обнаружением выбросов, для управления ими нет строгих правил. При выборе стратегии нам нужно руководствоваться двумя аспектами. Во-первых, нам следует понять, почему это выбросы. Если мы допускаем наличие ошибок в

данных, например из-за сломанного сенсора или неправильной кодировки значения, то такие значения ввиду ненадежности можно удалить или заменить выбросы на NaN. Однако если же у нас есть основания полагать, что выбросы являются экстремальными значениями (например, дом (поместье), имеющий 200 ванных комнат), было бы правильнее пометить их как выбросы или преобразовать их в более подходящие значения.

Во-вторых, выбор способа управления выбросами должен основываться на целях, которые мы пытаемся достичь в процесс машинного обучения. Например, если мы хотим предсказать стоимость дома на основе его признаков, то можно логически предположить, что динамика цен поместья, насчитывающего 100 ванных комнат, отличается от обычного семейного дома. Более того, при тренировке модели для онлайн-сервиса по оказанию услуг потребительских займов на жилье нам стоит принимать во внимание, что в числе наших потенциальных клиентов вряд ли будут миллиардеры, планирующие купить замок.

Так что же нам делать с выбросами? Подумайте, почему эти значения являются выбросами, держите в голове конечную цель вашей работы с данными и, что самое важное, не забывайте, что отсутствие решения что-либо сделать с выбросами само по себе является решением, имеющим последствия.

Есть еще один нюанс: при наличии выбросов стандартизация может быть некорректной ввиду неправильного среднего значения и дисперсии, сильно искаженных ошибочными данными. В этом случае наиболее устойчивым к выбросам оказывается масштабирование, например при помощи класса RobustScaler.

Дополнительные материалы для чтения

- Документация по RobustScaler (**https://oreil.ly/zgm-1**).

4.8. Дискретизация признаков

Задача

Есть числовой признак, который необходимо разбить на отдельные корзины (bins).

Решение

В зависимости от конечного результата мы можем использовать один из двух методов. Во-первых, мы можем использовать бинаризацию признаков (binarize the feature) в соответствии с заданным порогом:

```
# Загрузить библиотеку
import numpy as np
from sklearn.preprocessing import Binarizer
```

```
# Создать признаки
age = np.array([[6],
                [12],
                [20],
                [36],
                [65]])

# Определить бинаризацию
binarizer = Binarizer(threshold=18)

# Преобразовать признаки
binarizer.fit_transform(age)
array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Во-вторых, мы можем разбить числовые признаки в соответствии с несколькими порогами:

```
# Провести бинаризацию признаков
np.digitize(age, bins=[20,30,64])
array([[0],
       [0],
       [1],
       [2],
       [3]])
```

Следует учитывать, что аргументы для параметров `bins` определяют левую границу каждой корзины. Например, аргумент 20 включает только два значения ниже 20, но не сам элемент со значением 20. Мы можем изменить это, указав параметр `right` равным `True`:

```
# Провести бинаризацию признаков
np.digitize(age, bins=[20,30,64], right=True)
array([[0],
       [0],
       [0],
       [2],
       [3]])
```

Обсуждение

Дискретизация может быть полезна в ситуации, когда мы полагаем, что числовые признаки должны в большей степени походить на категориальные. К примеру, мы можем предположить, что разница в привычных тратах людей в возрасте 19 и 20 лет не столько значительна, как разница между людьми в возрасте 20 и 21 (в США с этого возраста разрешается употребление алкоголя). В этом примере нам может понадобиться разделить людей на группы тех, кто может и не может потреблять алкоголь. По аналогии с этим мы можем разбить данные на три корзины и более.

В примере выше мы использовали два способа дискретизации: класс `Binarizer` из scikit-learn для двух корзин и метод `digitize` из NumPy для трех и более; при этом мы также можем использовать метод `digitize` для бинаризации вместо `Binarizer`, указав один порог:

```
# Провести бинаризацию признаков
np.digitize(age, bins=[18])
array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Дополнительные материалы для чтения

- Документация по `digitize` (**https://oreil.ly/KipXX**).

4.9. Группирование наблюдений при помощи кластеризации

Задача

Вы хотите произвести кластеризацию наблюдаемых значений таким образом, чтобы они были разделены на группы по схожести значений.

Решение

Если вам известно k — количество групп, вы можете использовать метод k средних (k-means) для кластеризации в группы схожих наблюдаемых значений и добавить новый признак с информацией о принадлежности значения к группе:

```
# Загрузить библиотеку
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
```

```
# Создать искусственную матрицу признаков
features, _ = make_blobs(n_samples = 50,
                         n_features = 2,
                         centers = 3,
                         random_state = 1)

# Создать DataFrame
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Создать кластеризацию при помощи метода k средних
clusterer = KMeans(3, random_state=0)

# Настроить кластеризацию
clusterer.fit(features)

# Спрогнозировать значение
dataframe["group"] = clusterer.predict(features)

# Вывести первые наблюдаемые данные
dataframe.head(5)
```

	feature_1	feature_2	group
0	-9.877554	-3.336145	2
1	-7.287210	-8.353986	0
2	-6.943061	-7.023744	0
3	-7.440167	-8.791959	0
4	-6.641388	-8.075888	0

Обсуждение

Мы несколько забегаем вперед, т. к. более подробно алгоритмы кластеризации будут рассмотрены далее в этой книге. Здесь я хотел показать, как мы можем использовать кластеризацию на этапе первоначальной обработки. В частности, мы используем алгоритмы неуправляемого обучения (unsupervised learning), например метод *k* средних, для разделения наблюдаемых значений на группы. В результате мы объединяем категориальные признаки со схожими значениями в отдельные группы.

Не стоит беспокоиться, если что-то осталось непонятным для вас; просто имейте в виду, что кластеризацию можно использовать на этапе первичной обработки данных. Если вам все же не терпится узнать больше, то смело открывайте *главу 19*.

4.10. Удаление элементов с отсутствующим значением

Задача

Вам необходимо удалить наблюдаемые элементы с пустыми значениями.

Решение

Мы можем удалить наблюдения с пустыми значениями при помощи правильного использования стандартных инструментов библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу признаков
features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]])

# Оставить элементы, имеющие значения (при помощи ~)
features[~np.isnan(features).any(axis=1)]
array([[  1.1,  11.1],
       [  2.2,  22.2],
       [  3.3,  33.3],
       [  4.4,  44.4]])
```

Мы также можем удалить пропуски при помощи библиотеки pandas:

```
# Загрузить библиотеку
import pandas as pd

# Загрузить данные
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Удалить элементы с отсутствующими значениями
dataframe.dropna()

   feature_1  feature_2
0        1.1       11.1
1        2.2       22.2
2        3.3       33.3
3        4.4       44.4
```

Обсуждение

Большинство алгоритмов машинного обучения не может работать с отсутствующими значениями в целевых массивах и массивах признаков. По этой причине нам следует не игнорировать пустые значения при работе с данными, а избавляться от них уже на этапе предварительной обработки.

Проще всего будет удалить данные с пустыми значениями, что можно быстро и эффективно сделать при помощи библиотеки NumPy или pandas.

Важно отметить, что удаление пропусков для нас не самый подходящий вариант. Это скорее крайний вариант, т. к. в результате мы теряем связь нашего алгоритма и информации с непустыми значениями.

Важно и то, что в зависимости от причины отсутствия значения удаление элементов может по-разному влиять на наши данные. Существует три механизма формирования пропусков.

MCAR (missing completely at random — отсутствует совершенно случайно).

Вероятность формирования пустого значения ни от чего не зависит. К примеру, перед ответом в процессе опроса респондентка бросает игральную кость: если выпадает шесть, то она пропускает вопрос.

MAR (missing at random — отсутствует случайно).

Вероятность формирования пустого значения не полностью случайна, а соотносится с другими признаками. К примеру, респондентов просят предоставить информацию о половой принадлежности и ежегодной зарплате, при этом женщины с большей вероятностью игнорируют вопрос про зарплату; в таком случае отсутствие ответа коррелирует с полученными ранее данными признака о половой принадлежности.

MNAR (missing not at random — отсутствует неслучайно).

Вероятность формирования пустого значения не является случайной и не зависит от других признаков. К примеру, при опросе женщины с большей вероятностью пропускают пункт с ежегодной зарплатой, при этом в наших данных полностью отсутствует информация о половой принадлежности.

В случае MCAR или MAR иногда допустимо просто удалить значения. Однако при MNAR отсутствие значения само по себе является важной информацией. Удаление пропуска, атрибутированного как MNAR, накладывает отпечаток на все данные, поскольку мы удаляем значения, получившиеся в результате ненаблюдаемого систематического эффекта.

Дополнительные материалы для чтения

- Identifying the 3 Types of Missing Data ("Определение трех типов пропущенных данных", блог; **https://oreil.ly/sz9Fx**).

- Missing-Data Imputation ("Восстановление пропущенных данных"; **https://oreil.ly/swU2j**).

4.11. Восстановление пропущенных значений

Задача

В ваших данных имеются пропуски, которые вы хотите восстановить (impute) через общий метод или прогнозирование.

Решение

Вы можете восстановить пропущенные значения при помощи метода *k* ближайших соседей (k-nearest neighbors, KNN) или класса `SimpleImputer` библиотеки scikit-learn. Если вы работаете с небольшим объемом данных, можно спрогнозировать и восстановить значения пропусков, используя метод *k* ближайших соседей:

```
# Загрузить библиотеки
import numpy as np
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Создать матрицу искусственных признаков
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)

# Стандартизировать признаки
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Сделать пустым значение первого признака
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan

# Спрогнозировать значение пропуска матрицы признаков
knn_imputer = KNNImputer(n_neighbors=5)
features_knn_imputed = knn_imputer.fit_transform(standardized_features)
```

```
# Сравнить реальное и восстановленное значения
print("True Value:", true_value)
print("Imputed Value:", features_knn_imputed[0,0])
```

```
True Value: 0.8730186114
Imputed Value: 1.09553327131
```

В качестве альтернативы мы можем использовать класс `SimpleImputer` библиотеки scikit-learn для замены пропуска медианой, средним или наиболее распространенным значением признаков. Однако в большинстве случаев получившийся результат будет хуже, чем с использованием метода *k* ближайших соседей:

```
# Загрузить библиотеку
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Создать матрицу искусственных признаков
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)

# Стандартизировать признаки
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Сделать пустым значение первого признака
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan

# Добавить переменную для восстановления, используя среднее значение
mean_imputer = SimpleImputer(strategy="mean")

# Восстановить значения
features_mean_imputed = mean_imputer.fit_transform(features)

# Сравнить реальное и восстановленное значения
print("True Value:", true_value)
print("Imputed Value:", features_mean_imputed[0,0])
```

```
True Value: 0.8730186114
Imputed Value: -3.05837272461
```

Обсуждение

В нашем распоряжении есть две стратегии замены пропусков данных какими-либо значениями, каждый из которых обладает своими преимуществами и недостатками. Во-первых, мы можем использовать машинное обучение для прогнозирования вероятных значений. Для этого мы берем отсутствующее значение в качестве целевого вектора и получаем предполагаемое значение на основе оставшегося подмножества признаков. Существует множество алгоритмов восстановления данных на основе машинного обучения, самым распространенным из которых является метод *k* ближайших соседей. Кратко, в случае использования этого алгоритма восстановление значение пропуска основывается на значениях *k* ближайших элементов (в соответствии с заданными параметрами). Мы познакомимся с методом *k* ближайших соседей более подробно в *главе 15*. В этом примере для вычисления отсутствующего значения мы использовали данные пяти ближайших элементов.

Недостаток такого подхода заключается в том, что для определения ближайших элементов нам необходимо рассчитать расстояние между пропуском и каждым наблюдаемым значением. Это целесообразно при работе с небольшими наборами данных, однако все становится намного сложнее, когда в базе данных миллионы элементов. В такой ситуации нам лучше подойдет метод приближенных ближайших соседей (approximate nearest neighbors, ANN). Более подробно этот подход рассматривается в *рецепте 15.5*.

В качестве альтернативного и лучшего масштабируемого решения выступает восстановление пропусков числовых данных с использованием среднего значения, медианы или наиболее распространенного элемента последовательности. В примере выше мы использовали инструменты библиотеки scikit-learn для подстановки среднего значения. Такие значения могут сильнее отличаться от оригинальных, чем при использовании метода *k* ближайших соседей, однако нам намного проще масштабировать метод подстановки средних значений для работы с базами данных, насчитывающими миллионы элементов.

При восстановлении значений я рекомендую создать дополнительный признак в двоичном формате, который будет индикатором наличия или отсутствия восстановленных значений.

Дополнительные материалы для чтения

- Imputation of Missing Values ("Восстановление пропущенных данных", документация scikit-learn; **https://oreil.ly/1M4bn**).
- A Study of K-Nearest Neighbour as an Imputation Method ("Изучение метода *k* ближайших соседей для восстановления данных"; **https://oreil.ly/012--**).

ГЛАВА 5

Управление категориальными данными

Введение

При работе с данными полезно оценивать не только количественные, но и качественные признаки объектов. Мы часто представляем качественную информацию в виде различных категорий, например пол человека, цвета или марка автомобиля. Однако не все категориальные данные одинаковы. Наборы категорий без естественного упорядочения называются *номинальными* (nominal). Примерами значений номинальных категорий являются:

- синий, красный, зеленый;
- мужчина, женщина;
- банан, клубника, яблоко.

Напротив, когда набор категорий обладает естественным упорядочением, они называются *порядковыми* (ordinal). Например:

- низкий, средний, высокий;
- молодой, старый;
- согласный, нейтральный, несогласный.

Более того, категориальная информация часто представлена в данных в виде вектора или столбца строк (string), например `"Maine"` (Мэйн), `"Texas"` (Техас), `"Delaware"` (Делавэр). Проблема заключается в том, что большинство алгоритмов машинного обучения могут принимать на вход только числовые значения.

Примером такого алгоритма, требующего числовые данные, является метод *k* ближайших соседей. При выполнении алгоритма происходит вычисление расстояния между наблюдениями, зачастую при помощи евклидовой метрики:

$$\sqrt{\sum_{i=1}^{n}\left(x_i - y_i\right)^2},$$

где x и y — наблюдаемые значения; i — номер признака наблюдений. Если значение x_i является строкой (например, `"Texas"`), очевидно, что посчитать расстояние не получится. Вместо этого нам нужно преобразовать строковое значение в какое-

либо числовое, чтобы затем использовать его в формуле для определения евклидова расстояния. Наша цель заключается в правильной передаче информации, заложенной в категориях (упорядоченность, соотнесенность интервалов между категориями и т. д.). В данной главе мы разберем методы, при помощи которых сможем произвести необходимые преобразования, а также решить задачи, возникающие в процессе работы с категориальными данными.

5.1. Преобразование признаков с номинальными категориями

Задача

У вас есть признаки с номинальными классами, не имеющие естественного упорядочения (например, яблоко, банан), и вы хотите преобразовать их в числовые значения.

Решение

Воспользуемся унитарным кодированием (one-hot encode) признаков при помощи класса `LabelBinarizer` библиотеки scikit-learn:

```
# Импортировать библиотеки
import numpy as np
from sklearn.preprocessing import LabelBinarizer, MultiLabelBinarizer

# Создать признак
feature = np.array([["Texas"],
                    ["California"],
                    ["Texas"],
                    ["Delaware"],
                    ["Texas"]])

# Создать переменную для унитарного кодирования
one_hot = LabelBinarizer()

# Преобразовать признак
one_hot.fit_transform(feature)
array([[0, 0, 1],
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])
```

Для вывода классов мы можем воспользоваться атрибутом classes_:

```
# Вывести классы признака
one_hot.classes_
array(['California', 'Delaware', 'Texas'],
      dtype='<U10')
```

Для обратного преобразования можно использовать метод inverse_transform:

```
# Сделать обратное преобразование
one_hot.inverse_transform(one_hot.transform(feature))
array(['Texas', 'California', 'Texas', 'Delaware', 'Texas'],
      dtype='<U10')
```

Для унитарного кодирования признаков мы также можем использовать pandas:

```
# Подключить библиотеку
import pandas as pd

# Создать фиктивные переменные из признака
pd.get_dummies(feature[:,0])
```

```
   California  Delaware  Texas
0       False     False   True
1        True     False  False
2       False     False   True
3       False      True  False
4       False     False   True
```

В scikit-learn есть полезная особенность, позволяющая работать с наблюдениями, имеющими несколько классов:

```
# Создать многоклассовый признак
multiclass_feature = [("Texas", "Florida"),
                      ("California", "Alabama"),
                      ("Texas", "Florida"),
                      ("Delaware", "Florida"),
                      ("Texas", "Alabama")]

# Создать многоклассовую переменную для унитарного кодирования
one_hot_multiclass = MultiLabelBinarizer()

# Преобразовать многоклассовый признак
one_hot_multiclass.fit_transform(multiclass_feature)
```

```
array([[0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1],
       [0, 0, 1, 1, 0],
       [1, 0, 0, 0, 1]])
```

Для вывода классов мы снова можем воспользоваться атрибутом `classes_`:

```
# Вывести классы
one_hot_multiclass.classes_
array(['Alabama', 'California', 'Delaware', 'Florida', 'Texas'], dtype=object)
```

Обсуждение

На первый взгляд может показаться удачной идея преобразования классов в числовые значения (например, `Texas = 1`, `California = 2`). Однако, поскольку наши классы не являются естественно упорядоченными (например, `Texas` не может быть "меньше", чем `California`), замена на числовые значения привела бы к ошибке ввиду появления некорректного упорядочивания.

Правильной стратегией в данном случае будет создание двоичного признака для каждого класса изначальных признаков. Такой подход называется *унитарным кодированием* (one-hot encoding) в области машинного обучения или *моделью с фиктивными переменными* (dummying) в статистике и исследованиях. В нашем примере признаком является вектор, содержащий три класса (например, `Texas`, `California` и `Delaware`). При унитарном кодировании каждый класс становится отдельным признаком, где 1 означает его наличие, а 0 используется во всех остальных случаях. Поскольку наш признак имеет три класса, в результате унитарного кодирования мы получаем три двоичных признака (соответствующих каждому классу). При помощи этого метода мы можем получить информацию о классовой принадлежности значений без появления ошибочной иерархии классов.

Наконец, при использовании унитарного кодирования рекомендуется удалять один добавленный признак из получившейся матрицы, чтобы избежать линейной зависимости.

Дополнительные материалы для чтения

- Dummy Variable Trap in Regression Models, Algosome ("Ловушка фиктивной переменной в регрессионных моделях", блог Algosome; **https://oreil.ly/xjBhG**).
- Dropping one of the columns when using one-hot encoding, Cross Validated ("Удаление одного из столбцов при использовании унитарного кодирования", вопросно-ответный статистический ресурс CrossValidated; **https://oreil.ly/CTdpG**).

5.2. Преобразование признаков с порядковыми категориями

Задача

У вас есть признак с порядковыми категориями (например, высокий, средний, низкий), которые вы хотите преобразовать в количественные значения.

Решение

Создадим DataFrame и воспользуемся методом `replace` библиотеки pandas для преобразования строковых меток в числовые эквиваленты:

```
# Загрузить библиотеку
import pandas as pd

# Создать признаки
dataframe = pd.DataFrame({"Score": ["Low", "Low", "Medium", "Medium", "High"]})

# Создать разметку
scale_mapper = {"Low":1,
                "Medium":2,
                "High":3}

# Заменить значения признаков на шкалу
dataframe["Score"].replace(scale_mapper)
```

```
0    1
1    1
2    2
3    2
4    3
Name: Score, dtype: int64
```

Обсуждение

Мы часто работаем с признаками, в которых классы в разной степени естественно упорядочены. Простым примером является шкала Ликерта (Likert scale):

- полностью согласен;
- согласен;
- не определился;
- не согласен;
- полностью не согласен.

В процессе кодирования признаков для использования в машинном обучении нам необходимо преобразовывать порядковые классы в числовые значения, чтобы мы могли их упорядочить. Наиболее популярным подходом является создание словаря, в котором сопоставляются строковые метки классов и числовые значения для последующего применения в признаках.

Важно помнить, что выбор числовых значений должен основываться на информации в порядковых классах. В нашем примере high (высокий) в буквальном смысле означает в три раза выше low (низкий). Хотя во многих случаях это не является проблемой, появление неравных интервалов между классами может нарушить работу нашей модели:

```
dataframe = pd.DataFrame({"Score": ["Low",
                                    "Low",
                                    "Medium",
                                    "Medium",
                                    "High",
                                    "Barely More Than Medium"]})

scale_mapper = {"Low":1,
                "Medium":2,
                "Barely More Than Medium":3,
                "High":4}

dataframe["Score"].replace(scale_mapper)
0    1
1    1
2    2
3    2
4    4
5    3
Name: Score, dtype: int64
```

В этом примере разница между `Low` (низкий) и `Medium` (средний) совпадает с разницей между `Medium` и `Barely More Than Medium` (немного больше среднего), что почти наверняка является неточностью. Нам следует внимательнее относиться к числовым значениям при разметке классов:

```
scale_mapper = {"Low":1,
                "Medium":2,
                "Barely More Than Medium":2.1,
                "High":3}

dataframe["Score"].replace(scale_mapper)
0    1.0
1    1.0
```

```
2    2.0
3    2.0
4    3.0
5    2.1
Name: Score, dtype: float64
```

5.3. Преобразование словарей признаков

Задача

У нас есть словарь, который необходимо преобразовать в матрицу признаков.

Решение

Используем класс-векторизатор словаря `DictVectorizer`:

```
# Подключить модуль
from sklearn.feature_extraction import DictVectorizer

# Создать словарь
data_dict = [{"Red": 2, "Blue": 4},
             {"Red": 4, "Blue": 3},
             {"Red": 1, "Yellow": 2},
             {"Red": 2, "Yellow": 2}]

# Создать переменную векторизации словаря
dictvectorizer = DictVectorizer(sparse=False)

# Преобразовать словарь в матрицу признаков
features = dictvectorizer.fit_transform(data_dict)

# Вывести матрицу признаков
features
```

```
array([[ 4.,  2.,  0.],
       [ 3.,  4.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  2.]])
```

По умолчанию `DictVectorizer` создает разреженную матрицу, в которой хранятся только элементы с ненулевыми значениями. Это может быть полезно при работе с большими матрицами (часто встречаются при обработке текстов на естественном

языке) для уменьшения расхода памяти. Если мы укажем параметр `sparse=False`, `DictVectorizer` выведет плотную матрицу.

При помощи метода `get_feature_names_out` мы можем получить название каждого сгенерированного признака:

```
# Получить названия признаков
feature_names = dictvectorizer.get_feature_names_out()

# Вывести названия признаков
feature_names
```

```
['Blue', 'Red', 'Yellow']
```

Хотя это не обязательно, для наглядности результата мы можем создать DataFrame при помощи библиотеки pandas:

```
# Подключить модуль
import pandas as pd

# Создать DataFrame из признаков
pd.DataFrame(features, columns=feature_names)
```

```
   Blue  Red  Yellow
0   4.0  2.0     0.0
1   3.0  4.0     0.0
2   0.0  1.0     2.0
3   0.0  2.0     2.0
```

Обсуждение

Словари являются распространенным вариантом структуры данных и используются во многих языках программирования; однако в случае с алгоритмами машинного обучения данные должны быть представлены в форме матрицы. Для решения этой задачи в scikit-learn есть преобразователь `DictVectorizer`.

Мы часто сталкиваемся с такой ситуацией в процессе обработки текстов на естественном языке. К примеру, у нас есть набор документов, и в каждом документе имеется словарь со счетчиком вхождений уникальных слов в документе. При помощи `DictVectorizer` мы можем легко создать матрицу признаков, в которой каждый признак является количеством вхождений слова в каждом документе:

```
# Создать словарь со счетчиком слов для четырех документов
doc_1_word_count = {"Red": 2, "Blue": 4}
doc_2_word_count = {"Red": 4, "Blue": 3}
doc_3_word_count = {"Red": 1, "Yellow": 2}
doc_4_word_count = {"Red": 2, "Yellow": 2}
```

```
# Создать список
doc_word_counts = [doc_1_word_count,
                   doc_2_word_count,
                   doc_3_word_count,
                   doc_4_word_count]

# Преобразовать список словарей со счетчиком слов в матрицу признаков
dictvectorizer.fit_transform(doc_word_counts)
array([[ 4.,  2.,  0.],
       [ 3.,  4.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  2.]])
```

В нашем искусственном примере есть только три слова (`Red`, `Yellow`, `Blue`), поэтому наша матрица состоит из трех признаков. Тем не менее вы можете представить, что если бы каждый документ был книгой из университетской библиотеки, у нас получилась бы очень большая матрица признаков (и в таком случае мы бы поменяли значение параметра `sparse` на `True`).

Дополнительные материалы для чтения

- How to create dictionaries in Python ("Как создавать словари в Python"; **https://oreil.ly/zu5hU**).
- SciPy Spares Matrices ("Разреженные матрицы SciPy"; **https://oreil.ly/5nAsU**).

5.4. Восстановление классов пропущенных значений

Задача

У вас есть категориальный признак с пропущенными значениями, которые необходимо восстановить.

Решение

Идеальным решением было бы создание алгоритма классификации на основе машинного обучения для предугадывания значений пропусков, например классификатора на основе метода *k* ближайших соседей:

```
# Загрузить библиотеки
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
```

```
# Создать матрицу признаков с категориальными элементами
X = np.array([[0, 2.10, 1.45],
              [1, 1.18, 1.33],
              [0, 1.22, 1.27],
              [1, -0.21, -1.19]])

# Создать матрицу признаков с пропущенными значениями категориальных элементов
X_with_nan = np.array([[np.nan, 0.87, 1.31],
                       [np.nan, -0.67, -0.22]])

# Обучить классификатор на основе метода k ближайших соседей
clf = KNeighborsClassifier(3, weights='distance')
trained_model = clf.fit(X[:,1:], X[:,0])

# Восстановить классы пропусков
imputed_values = trained_model.predict(X_with_nan[:,1:])

# Объединить столбец с восстановленными классами и другие признаки
X_with_imputed = np.hstack((imputed_values.reshape(-1,1), X_with_nan[:,1:]))

# Объединить две матрицы признаков
np.vstack((X_with_imputed, X))
array([[ 0. ,  0.87,  1.31],
       [ 1. , -0.67, -0.22],
       [ 0. ,  2.1 ,  1.45],
       [ 1. ,  1.18,  1.33],
       [ 0. ,  1.22,  1.27],
       [ 1. , -0.21, -1.19]])
```

В качестве альтернативного решения мы можем восстановить пропуски при помощи наиболее распространенного значения признаков:

```
from sklearn.impute import SimpleImputer

# Объединить две матрицы признаков
X_complete = np.vstack((X_with_nan, X))

imputer = SimpleImputer(strategy='most_frequent')

imputer.fit_transform(X_complete)
array([[ 0. ,  0.87,  1.31],
       [ 0. , -0.67, -0.22],
```

```
[ 0. ,  2.1 ,  1.45],
[ 1. ,  1.18,  1.33],
[ 0. ,  1.22,  1.27],
[ 1. , -0.21, -1.19]])
```

Обсуждение

Когда в категориальных признаках встречаются пропуски, нам следует обратиться к инструментарию алгоритмов машинного обучения для восстановления отсутствующих значений. При таком подходе признак с пропусками становится целевым вектором, а остальные признаки объединяются в матрицу признаков. Одним из популярных алгоритмов является метод *k* ближайших соседей (более подробно разбирается в *главе 15*), при котором пропущенному значению присваивается класс, наиболее распространенный среди заданного количества ближайших элементов.

Другим способом является присвоение пропущенному элементу значения наиболее распространенного класса признака или вовсе удаление пропуска. Хотя такой подход является менее продвинутым, чем метод *k* ближайших соседей, он намного проще масштабируется для работы с большими объемами данных. В таком случае рекомендуется добавлять признак с двоичными значениями для индикации восстановленных пропусков.

Дополнительные материалы для чтения

- Imputation of Missing Values ("Восстановление пропущенных значений", документация scikit-learn; **https://oreil.ly/joZ6J**).
- Overcoming Missing Values in a Random Forest Classifier ("Работа с пропущенными значениями при классификации методом случайного леса"; **https://oreil.ly/TcvOf**).
- A Study of K-Nearest Neighbour as an Imputation Method ("Изучение метода *k* ближайших соседей для восстановления данных"; **https://oreil.ly/kDFEC**).

5.5. Управление несбалансированными классами

Задача

У вас есть вектор с высокой несбалансированностью классов, и вы хотите внести изменения для их балансировки.

Решение

Соберите больше данных. Если же это невозможно, измените метрики, используемые для оценки вашей модели. Если это также не сработает, воспользуйтесь встро-

енными параметрами веса (weight) класса (если доступен), субдискретизацией (downsampling) или передискретизацией (upsampling). Мы более подробно разберем оценивающие метрики модели в другой главе; пока что сосредоточимся на параметрах веса класса, субдискретизации и передискретизации.

Для демонстрации нам понадобится создать набор данных с несбалансированными классами. Набор данных ирисов Фишера содержит три сбалансированных класса по 50 элементов с информацией о разных видах цветов (*Iris setosa* (ирис щетинистый), *Iris virginica* (ирис виргинский) и *Iris versicolor* (ирис разноцветный)). Удалим 40 из 50 элементов с информацией об ирисе щетинистом и объединим получившийся результат с данными об ирисе виргинском и ирисе разноцветном, тем самым нарушив баланс классов. В результате мы получим целевой двоичный вектор с информацией о том, является элемент ирисом щетинистым или нет. Другими словами, 10 элементов вектора относятся к ирису щетинистому (класс 0) и 100 элементов не относятся *к* ирису щетинистому (класс 1):

```
# Загрузить библиотеки
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Загрузить набор данных об ирисах
iris = load_iris()

# Создать матрицу признаков
features = iris.data

# Создать целевой вектор
target = iris.target

# Удалить 40 первых элементов
features = features[40:,:]
target = target[40:]

# Создать целевой двоичный вектор с индикацией класса 0
target = np.where((target == 0), 0, 1)

# Вывести несбалансированный целевой вектор
target
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 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])
```

Во многих алгоритмах scikit-learn присутствует параметр веса классов, используемый в процессе обучения модели для снижения влияния дисбаланса. Одним из популярных алгоритмов классификации, включающим параметр `class_weight`, является новый для нас алгоритм `RandomForestClassifier`; более подробное описание `RandomForestClassifier` находится в *рецепте 14.4*. Вы можете самостоятельно задать аргумент с весами классов:

```
# Создать веса
weights = {0: 0.9, 1: 0.1}

# Добавить классификацию методом случайного леса с весами
RandomForestClassifier(class_weight=weights)
RandomForestClassifier(class_weight={0: 0.9, 1: 0.1})
```

Вы также можете воспользоваться параметром `balanced` для автоматического создания параметров весов, обратно пропорциональных частоте вхождений класса:

```
# Обучить классификатор при помощи весов для балансировки классов
RandomForestClassifier(class_weight="balanced")
RandomForestClassifier(class_weight='balanced')
```

В качестве альтернативного метода мы можем использовать субдискретизацию (т. е. удаление части примеров мажоритарного класса) или передискретизацию (т. е. увеличение примеров миноритарного класса). В случае *субдискретизации* мы выбираем случайные примеры мажоритарного класса (т. е. класса с наибольшим числом наблюдаемых элементов) без замены для создания нового набора значений, равного по размеру миноритарному классу. К примеру, если миноритарный класс насчитывает 10 значений, нам нужно будет выбрать 10 значений мажоритарного класса и использовать суммарные данные из 20 значений. Давайте опробуем этот метод на нашем примере с несбалансированными данными ирисов:

```
# Добавить индексы элементов каждого класса
i_class0 = np.where(target == 0)[0]
i_class1 = np.where(target == 1)[0]

# Узнать количество элементов в каждом классе
n_class0 = len(i_class0)
n_class1 = len(i_class1)

# Для каждого элемента класса 0 выбрать случайный пример
# класса 1 без замены
i_class1_downsampled = np.random.choice(i_class1, size=n_class0, replace=False)

# Объединить целевой вектор класса 0
# и целевой вектор класса 1 после субдискретизации
np.hstack((target[i_class0], target[i_class1_downsampled]))
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
# Объединить матрицу признаков класса 0
# и матрицу признаков класса 1 после субдискретизации
np.vstack((features[i_class0,:], features[i_class1_downsampled,:]))[0:5]
array([[ 5. ,  3.5,  1.3,  0.3],
       [ 4.5,  2.3,  1.3,  0.3],
       [ 4.4,  3.2,  1.3,  0.2],
       [ 5. ,  3.5,  1.6,  0.6],
       [ 5.1,  3.8,  1.9,  0.4]])
```

Другим способом является *передискретизация*, при которой для каждого элемента мажоритарного класса мы выбираем случайное значение миноритарного класса с заменой. В результате мы получаем одинаковое количество элементов миноритарного и мажоритарного классов. Процесс передискретизации схож с субдискретизацией, только происходит в обратном порядке:

```
# Для каждого элемента класса 1 выбрать случайный пример
# класса 0 с заменой
i_class0_upsampled = np.random.choice(i_class0, size=n_class1, replace=True)

# Объединить целевой вектор класса 0 и целевой вектор класса 1 после передискретизации
np.concatenate((target[i_class0_upsampled], target[i_class1]))
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 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, 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])

# Объединить матрицу признаков класса 0
# и матрицу признаков класса 1 после передискретизации
np.vstack((features[i_class0_upsampled,:], features[i_class1,:]))[0:5]
array([[ 5. ,  3.5,  1.6,  0.6],
       [ 5. ,  3.5,  1.6,  0.6],
       [ 5. ,  3.3,  1.4,  0.2],
       [ 4.5,  2.3,  1.3,  0.3],
       [ 4.8,  3. ,  1.4,  0.3]])
```

Обсуждение

При работе с реальными данными несбалансированные классы встречаются постоянно: многие посетители онлайн-магазинов не нажимают кнопку "купить", а мно-

гие типы рака, к счастью, встречаются редко. По этой причине в машинном обучении часто используется управление несбалансированными классами.

Лучшей стратегией будет сбор дополнительных данных, особенно для миноритарных классов. Однако это не всегда возможно, поэтому нам приходится прибегать к иным методам.

Альтернативным способом является использование других метрик оценки модели, подходящих для работы с несбалансированными классами. Метрика правильности (accuracy) часто используется для оценки производительности модели, но она мало подходит при наличии несбалансированных классов. Например, если информация о редком типе рака содержится только в 0,5% всех наблюдений, даже самая простая модель, предугадывающая отсутствие больных раком, будет точна на 99,5%. Это совершенно точно далеко от идеала. В других главах мы рассмотрим более подходящие метрики, в том числе матрицы ошибок (confusion matrices), точность (precision), полноту (recall), F1-меру (F1 scores) и ROC-кривые (ROC curves).

Третьей стратегией является использование параметров веса класса, который присутствует в некоторых моделях. Это позволяет алгоритму настраивать несбалансированные классы. Данный способ весьма удобен, поскольку параметр `class_weight` есть во многих классификаторах scikit-learn.

Четвертая и пятая стратегии схожи: субдискретизация и передискретизация. При субдискретизации мы создаем случайное подмножество мажоритарного класса, соответствующего размеру миноритарного. При передискретизации мы несколько раз добавляем данные с заменой на основе миноритарного класса, чтобы сравнять его размер с мажоритарным. Выбор между субдискретизацией и передискретизацией зависит от ситуации, и следует пробовать оба варианта для получения лучшего результата.

ГЛАВА 6

Обработка текста

Введение

Неструктурированные текстовые данные вроде книг или твитов являются одновременно самыми интересными источниками признаков и самыми сложными для обработки. В этой главе мы рассмотрим стратегии преобразования текста в информативные признаки, а также попробуем поработать с некоторыми готовыми признаками (так называемый *эмбеддинг* (embedding)), которые повсеместно используются в задачах по обработке текстов на естественном языке (natural language processing, NLP).

Стоит ли говорить, что примеры в данной главе не являются всеобъемлющими. Целые научные дисциплины занимаются вопросом управления неструктурированными данными, такими как текст. В этой главе мы рассмотрим некоторые широко используемые методы, владение которыми расширит наш инструментарий для предварительной обработки данных. Кроме общих примеров обработки текста мы также увидим на практике, как можно добавлять и улучшать некоторые предварительно обученные модели для обогащения получаемых из текста признаков.

6.1. Очистка текста

Задача

У вас есть неструктурированные текстовые данные, и вы хотите произвести базовую очистку.

Решение

Давайте рассмотрим на примере, как можно очистить текст, взятый из трех книг, при помощи встроенных операторов Python для работы со строками (string), включая `strip`, `replace` и `split`:

```
# Создать текст
text_data = ["   Interrobang. By Aishwarya Henriette     ",
             "Parking And Going. By Karl Gautier",
             "    Today Is The night. By Jarek Prakash   "]
```

```
# Удалить лишние пробелы
strip_whitespace = [string.strip() for string in text_data]

# Вывести текст
strip_whitespace
['Interrobang. By Aishwarya Henriette',
 'Parking And Going. By Karl Gautier',
 'Today Is The night. By Jarek Prakash']

# Удалить точки
remove_periods = [string.replace(".", "") for string in strip_whitespace]

# Вывести текст
remove_periods
['Interrobang By Aishwarya Henriette',
 'Parking And Going By Karl Gautier',
 'Today Is The night By Jarek Prakash']
```

Мы также можем добавить и применить собственную трансформационную функцию:

```
# Создать функцию
def capitalizer(string: str) -> str:
    return string.upper()

# Применить функцию
[capitalizer(string) for string in remove_periods]
['INTERROBANG BY AISHWARYA HENRIETTE',
  'PARKING AND GOING BY KARL GAUTIER',
 'TODAY IS THE NIGHT BY JAREK PRAKASH']
```

Наконец, мы можем использовать регулярные выражения для операций со строковыми значениями:

```
# Подключить модуль
import re

# Создать функцию
def replace_letters_with_X(string: str) -> str:
    return re.sub(r"[a-zA-Z]", "X", string)

# Применить функцию
[replace_letters_with_X(string) for string in remove_periods]
['XXXXXXXXXXX XX XXXXXXXXX XXXXXXXXX',
 'XXXXXXX XXX XXXXX XX XXXX XXXXXXX',
 'XXXXX XX XXX XXXXX XX XXXXX XXXXXXX']
```

Обсуждение

В некоторых случаях нам может понадобиться очистить данные перед созданием признаков на их основе или предварительно обработать их определенным образом, чтобы затем добавить в алгоритм. При обработке простых текстов достаточно стандартных операций Python для работы со строками. На практике мы скорее всего захотим создать собственную функцию для очистки (например, capitalizer), в которой совместим несколько задач для очистки, чтобы затем применить их к тексту. Хотя процесс очистки может привести к утрате незначительной части информации, нам будет намного проще работать с данными. Для очистки и обработки данных мы можем использовать различные инструменты для работы со строковыми данными; ниже приведены некоторые дополнительные примеры:

```
# Объявить переменную строкового типа
s = "machine learning in python cookbook"

# Узнать индекс первого вхождения буквы "n"
find_n = s.find("n")

# Начинается ли строка с буквы "m"?
starts_with_m = s.startswith("m")

# Заканчивается ли строка на "python"?
ends_with_python = s.endswith("python")

# Является ли строка буквенно-цифровой?
is_alnum = s.isalnum()

# Состоит ли строка только из буквенных символов (не включая пробелы)?
is_alpha = s.isalpha()

# Кодировать в utf-8
encode_as_utf8 = s.encode("utf-8")

# Декодировать из utf-8
decode = encode_as_utf8.decode("utf-8")

print(
    find_n,
    starts_with_m,
    ends_with_python,
    is_alnum,
    is_alpha,
```

```
    encode_as_utf8,
    decode,
    sep = "|"
)
```

```
5|True|False|False|False|b'machine learning in python cookbook'|machine learning in
python cookbook
```

Дополнительные материалы для чтения

- Beginners Tutorial for Regular Expressions in Python ("Руководство для начинающих по регулярным выражениям в Python"; **https://oreil.ly/hSqsa**).

6.2. Синтаксический анализ и очистка HTML-документов

Задача

У вас есть текстовые данные с HTML-элементами, из которых вы хотите извлечь только текст.

Решение

Воспользуемся средствами библиотеки Beautiful Soup для парсинга и извлечения данных из HTML-документа:

```
# Загрузить библиотеку
from bs4 import BeautifulSoup

# Написать некоторый HTML-код
html = "<div class='full_name'>"\
       "<span style='font-weight:bold'>Masego"\
       "</span> Azra</div>"

# Соберем и структурируем информацию HTML-документа
soup = BeautifulSoup(html, "lxml")

# Найдем элемент div с классом full_name и выведем текст
soup.find("div", { "class" : "full_name" }).text
```

```
'Masego Azra'
```

Обсуждение

Несмотря на странное название, Beautiful Soup является мощной библиотекой Python для скрейпинга (очистки) HTML-документов. Этот инструмент обычно используется для обработки HTML-страниц в процессе веб-скрейпинга, однако мы можем легко применить его для извлечения текста из статических HTML-документов. Подробный разбор операций Beautiful Soup выходит за пределы темы данной книги, но нашего примера вполне достаточно, чтобы понять, насколько легко можно собрать и извлечь информацию из HTML-документа при помощи метода `find()` с определенными тегами.

Дополнительные материалы для чтения

- Веб-сайт библиотеки Beautiful Soup (**https://oreil.ly/vh8h3**).

6.3. Удаление знаков препинания

Задача

Вы создали признак из текстовых данных и теперь хотите убрать из него все пунктуационные знаки.

Решение

Объявим функцию, в которой будем использовать метод `translate` со словарем символов пунктуации:

```
# Загрузить библиотеку
import unicodedata
import sys

# Создать текст
text_data = ['Hi!!!! I. Love. This. Song....',
             '10000% Agree!!!! #LoveIT',
             'Right?!?!']

# Создать словарь символов пунктуации
punctuation = dict.fromkeys(
    (i for i in range(sys.maxunicode)
        if unicodedata.category(chr(i)).startswith('P')
    ),
    None
)
```

```
# Удалить все символы пунктуации в каждой строке
[string.translate(punctuation) for string in text_data]
['Hi I Love This Song', '10000 Agree LoveIT', 'Right']
```

Обсуждение

Метод Python `translate` часто используется ввиду высокой скорости работы. В нашем примере мы сначала создаем словарь — переменную `punctuation`, — куда в соответствии с кодировкой Unicode добавляем все символы пунктуации в качестве ключей и `None` в качестве пустых значений. Далее мы преобразуем все символы пунктуации строки в `None`, тем самым избавляясь от них. Существуют другие, более понятные способы удаления знаков препинания, однако это, не совсем интуитивное, решение является наиболее быстрым.

Важно понимать, что в пунктуации заключена определенная информация (например, "Правильно?" и "Правильно!"). При создании признаков вручную удаление пунктуации является необходимым злом, однако сначала нужно убедиться, что мы учли всю содержащуюся в ней важную информацию. В зависимости от последующих задач, которые наша модель должна решать, нам может потребоваться сохранить пунктуационную информацию (например, знак "?" для классификации вопроса).

6.4. Токенизация текста

Задача

У вас есть текст, который нужно разбить на отдельные слова.

Решение

В библиотеке Python NLTK (Natural Language Toolkit) содержится большой набор инструментов для работы с текстом, включая токенизацию:

```
# Загрузить библиотеку
from nltk.tokenize import word_tokenize

# Создать текст
string = "The science of today is the technology of tomorrow"

# Разделить текст на слова
word_tokenize(string)
['The', 'science', 'of', 'today', 'is', 'the', 'technology', 'of', 'tomorrow']
```

Мы также можем осуществить токенизацию по предложениям:

```
# Загрузить библиотеку
from nltk.tokenize import sent_tokenize
```

```
# Создать текст
string = "The science of today is the technology of tomorrow. Tomorrow is today."

# Разделить текст на предложения
sent_tokenize(string)
['The science of today is the technology of tomorrow.', 'Tomorrow is today.']
```

Обсуждение

Токенизация, особенно разделение на слова, является обычным этапом после очистки текстовых данных, с которого начинается их преобразование для последующего создания признаков. В некоторых заранее натренированных NLP-моделях (например, Google BERT) используются собственные методы токенизации; тем не менее токенизация на уровне слов остается распространенным подходом при создании признаков на основе отдельных слов.

6.5. Удаление стоп-слов

Задача

В процессе токенизации текстовых данных вам может понадобиться удалить слишком общие стоп-слова (например, английские артикли и предлоги *a*, *is*, *of*, *on*), которые обладают крайне малой информативностью.

Решение

Воспользуемся инструментом `stopwords` библиотеки NLTK:

```
# Загрузить библиотеку
from nltk.corpus import stopwords

# При первом использовании необходимо скачать список стоп-слов
# import nltk
# nltk.download('stopwords')

# Создать токены из слов
tokenized_words = ['i',
                   'am',
                   'going',
                   'to',
                   'go',
                   'to',
```

```
                    'the',
                    'store',
                    'and',
                    'park']

# Загрузить стоп-слова
stop_words = stopwords.words('english')

# Удалить стоп-слова
[word for word in tokenized_words if word not in stop_words]
['going', 'go', 'store', 'park']
```

Обсуждение

Хотя "шумовые" или "стоп-слова" могут означать к любой набор слов, которые мы хотим удалить перед дальнейшей обработкой, данный термин часто означает слишком общие слова, сами по себе не содержащие ценной информации. Решение об удалении тех или иных слов будет зависеть от конкретных целей. В NLTK представлен список распространенных шумовых слов, которые мы можем использовать для поиска и удаления токенов из нашей модели:

```
# Вывести стоп-слова
stop_words[:5]
['i', 'me', 'my', 'myself', 'we']
```

Обратите внимание, что в функции `stopwords` библиотеки NLTK в токенах слов используются строчные буквы.

6.6. Стемминг слов

Задача

Вы хотите преобразовать имеющиеся токены (слова), чтобы преобразовать их в корневую основу.

Решение

Используем класс `PorterStemmer` из библиотеки NLTK:

```
# Загрузить библиотеку
from nltk.stem.porter import PorterStemmer

# Создать токены из слов
tokenized_words = ['i', 'am', 'humbled', 'by', 'this', 'traditional', 'meeting']
```

```
# Определить стеммер
porter = PorterStemmer()

# Применить стеммер
[porter.stem(word) for word in tokenized_words]
['i', 'am', 'humbl', 'by', 'thi', 'tradit', 'meet']
```

Обсуждение

Алгоритм *стемминга* позволяет сократить слова до их основы (до корня) путем определения и удаления аффиксов (например, герундий) с сохранением оригинального значения слова. Например, такие слова, как "tradition" и "traditional", имеют основу "tradit", другими словами, это разные слова, выражающие одну общую концепцию. После стемминга наши данные становятся менее понятными для чтения, однако они более точно выражают исходное значение, а значит, лучше подходят для сравнения наблюдений. В основе класса `PorterStemmer` библиотеки NLTK лежит широко используемый алгоритм стемминга Портера для удаления и замены общих суффиксов и получения основы слова.

Дополнительные материалы для чтения

- The Porter Stemming Algorithm ("Алгоритм стемминга Портера"; **https://oreil.ly/Z4NTp**).

6.7. Разметка частей речи

Задача

У вас есть текстовые данные, и вы хотите пометить каждое слово или символ соответствующей частью речи.

Решение

Воспользуемся готовым инструментом для разметки частей речи библиотеки NLTK:

```
# Загрузить библиотеку
from nltk import pos_tag
from nltk import word_tokenize

# Создать текст
text_data = "Chris loved outdoor running"
```

```
# Разметить части речи
text_tagged = pos_tag(word_tokenize(text_data))

# Вывести части речи
text_tagged
[('Chris', 'NNP'), ('loved', 'VBD'), ('outdoor', 'RP'), ('running', 'VBG')]
```

Результатом является список (list) кортежей (tuple), в которых соотносятся слова и части речи. В NLTK используется разметка из синтаксического аннотированного корпуса Penn Treebank. Некоторые примеры разметки корпуса включают:

Тег	Часть речи
NNP	Имя собственное, единственное число
NN	Существительное, единственное или множественное число
RB	Наречие
VBD	Глагол, прошедшее время
VBG	Глагол, герундий или причастие настоящего времени
JJ	Прилагательное
PRP	Личное местоимение

После разметки текста мы можем использовать теги для поиска определенных частей речи. К примеру, выведем только существительные:

```
# Отсортировать слова
[word for word, tag in text_tagged if tag in ['NN','NNS','NNP','NNPS'] ]
['Chris']
```

В более естественном примере наблюдаемые элементы наших данных скорее всего будут содержать твиты, и нам может потребоваться разбить предложения на отдельные признаки по принадлежности к частям речи (например, 1 для существительных и 0 для остальных признаков):

```
# Импортировать библиотеки
from sklearn.preprocessing import MultiLabelBinarizer

# Создать текст
tweets = ["I am eating a burrito for breakfast",
          "Political science is an amazing field",
          "San Francisco is an awesome city"]

# Создать список
tagged_tweets = []

# Разметить каждое слово и каждый твит
for tweet in tweets:
    tweet_tag = nltk.pos_tag(word_tokenize(tweet))
    tagged_tweets.append([tag for word, tag in tweet_tag])
```

```
# Преобразовать теги в признаки при помощи унитарного кода
one_hot_multi = MultiLabelBinarizer()
one_hot_multi.fit_transform(tagged_tweets)
array([[1, 1, 0, 1, 0, 1, 1, 1, 0],
       [1, 0, 1, 1, 0, 0, 0, 0, 1],
       [1, 0, 1, 1, 1, 0, 0, 0, 1]])
```

С помощью `classes_` мы можем увидеть, что каждый признак является тегом, выражающим часть речи:

```
# Вывести названия признаков
one_hot_multi.classes_
array(['DT', 'IN', 'JJ', 'NN', 'NNP', 'PRP', 'VBG', 'VBP', 'VBZ'], dtype=object)
```

Обсуждение

Использование готового инструмента для разметки частей речи из библиотеки NLTK является самым простым решением в случае, если мы работаем с текстом на английском языке неспециализированной тематики (например, из области медицины). При этом, если результаты `pos_tag` недостаточно точны, мы можем самостоятельно обучить модель разметки частей речи. Существенным недостатком обучения такой модели является необходимость наличия большого корпуса текстов с размеченными словами. Создание такого размеченного корпуса — очень трудоемкий процесс, к которому стоит переходить в крайнем случае.

Дополнительные материалы для чтения

- Список тегов частей речи в алфавитном порядке, используемый в корпусе Penn (**https://oreil.ly/31xKf**).

6.8. Распознавание именованных сущностей

Задача

Вам нужно произвести распознавание именованных сущностей (named-entity recognition) в неструктурированном (freeform) тексте (например, "Person" (человек), "State" (состояние) и т. д.).

Решение

Используем установленные по умолчанию конвейер и модели распознавания именованных сущностей библиотеки spaCy для извлечения сущностей из текста:

```
# Импортировать библиотеки
import spacy
```

```
# Загрузить пакет spaCy и провести синтаксический анализ текста.
# Убедиться, что предварительно запустили "python -m spacy download en"
nlp = spacy.load("en_core_web_sm")
doc = nlp("Elon Musk offered to buy Twitter using $21B of his own money.")

# Вывести на экран каждую сущность
print(doc.ents)

# Для каждой сущности вывести текст и метку
for entity in doc.ents:
    print(entity.text, entity.label_, sep=",")
```

```
(Elon Musk, Twitter, 21B)
Elon Musk, PERSON
Twitter, ORG
21B, MONEY
```

Обсуждение

Распознавание именованных сущностей — это механизм распознавания особых сущностей в тексте. В инструментах вроде spaCy есть предварительно настроенные конвейеры, а в некоторых случаях — предварительно обученные или точно настроенные модели машинного обучения, которые могут без труда определять эти сущности. В данном случае мы используем spaCy для идентификации человека ("Elon Musk"), организации ("Twitter") и финансового состояния ("21B" (миллиардов)) в исходном тексте. В результате мы можем получить структурированную информацию из неструктурированных текстовых данных. Эту информацию можно использовать в дальнейшем для обучения моделей или анализа данных.

Обучение собственной модели распознавания именованных сущностей не входит в задачи данной книги; для этого обычно используются глубокое обучение и другие методы обработки текстов на естественном языке.

Дополнительные материалы для чтения

- Документация функции распознавания именованных сущностей в spaCy (**https://oreil.ly/cN8KM**).
- Named-entity recognition ("Распознавание именованных сущностей", Википедия; **https://oreil.ly/G8WDF**).

6.9. Представление текста в виде мешка слов

Задача

Даны текстовые данные, на основе которых необходимо создать набор признаков с информацией о количестве вхождений слова в тексте.

Решение

Воспользуемся классом CountVectorizer из библиотеки scikit-learn:

```
# Загрузить библиотеки
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

# Создать текст
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])

# Представить мешок слов в виде матрицы признаков
count = CountVectorizer()
bag_of_words = count.fit_transform(text_data)

# Вывести матрицу признаков
bag_of_words
<3x8 sparse matrix of type '<class 'numpy.int64'>'
    with 8 stored elements in Compressed Sparse Row format>
```

В результате мы получаем разреженный массив, который необходим при работе с объемными текстами. Для просмотра матрицы счетчиков слов для каждого наблюдения мы можем использовать метод toarray:

```
bag_of_words.toarray()
array([[0, 0, 0, 2, 0, 0, 1, 0],
       [0, 1, 0, 0, 0, 1, 0, 1],
       [1, 0, 1, 0, 1, 0, 0, 0]], dtype=int64)
```

При помощи метода get_feature_names_out мы можем просмотреть связанное с признаком слово:

```
# Вывести названия признаков
count.get_feature_names_out()
['beats' 'best' 'both' 'brazil' 'germany' 'is' 'love' 'sweden']
```

Обратите внимание, что слово "I" из фразы "I love Brazil" не является токеном, поскольку при настройке `token_pattern` по умолчанию токены могут состоять из двух и более буквенно-цифровых символов.

Для внесения большей ясности в этом сложном вопросе ниже показана матрица признаков со словами в качестве названий столбцов (каждая строка является отдельным наблюдением):

beats	best	both	brazil	germany	is	love	sweden
0	0	0	2	0	0	1	0
0	1	0	0	0	1	0	1
1	0	1	0	1	0	0	0

Обсуждение

Использование модели мешка слов (bag-of-words) является одним из наиболее распространенных методов преобразования текста в признаки. Данная модель возвращает признак для каждого уникального слова в тексте, а также счетчик вхождений для каждого наблюдения. В нашем примере, "I love Brazil. Brazil!", признак "brazil" имеет значение 2, поскольку слово *"Brazil"* дважды появляется в тексте.

В рассмотренном примере мы специально использовали небольшой объем данных. На практике один наблюдаемый элемент текстовых данных может быть целой книгой! Поскольку наша модель мешка слов создает признак из каждого уникального слова, получившаяся матрица может насчитывать тысячи признаков. Таким образом, в некоторых случаях матрицы могут быть весьма требовательными к объему памяти. К счастью, у матрицы с признаками из мешка слов есть одна характеристика, которую мы можем использовать для уменьшения объема занимаемой памяти.

Большинство слов, весьма вероятно, не появятся в наблюдениях, поэтому такая матрица будет во многом состоять из нулевых значений. Матрицы такого типа называются *разреженными* (sparse). В них нам не обязательно хранить все значения — достаточно оставить ненулевые значения, а остальные принимать за 0. Это позволит использовать меньше памяти при работе с большими матрицами признаков. Одной из полезных особенностей `CountVectorizer` является то, что на выходе по умолчанию мы получаем разреженную матрицу.

В `CountVectorizer` есть множество полезных параметров, которые упрощают процесс создания матриц признаков из мешков слов. Во-первых, хотя по умолчанию признаком является слово, мы можем поменять его под свои задачи. Мы можем сделать каждый признак комбинацией из двух слов или даже трех слов (так называемые *N*-граммы). Мы можем задать минимальный и максимальный размеры *N*-грамм при помощи параметра `ngram_range`. К примеру, значение параметра `(2, 3)` вернет 2-граммы и 3-граммы. Во-вторых, мы можем легко удалить неинформативные слова при помощи метода `stop_words`, используя встроенный или собственный списки. Наконец, при помощи `vocabulary` мы можем ограничить вывод слов или

фраз, относящихся к какому-либо списку. Скажем, мы могли бы создать матрицу признаков из мешков слов, который отображает только названия стран:

```
# Создать матрицу признаков, указав аргументы
count_2gram = CountVectorizer(ngram_range=(1,2),
                              stop_words="english",
                              vocabulary=['brazil'])
bag = count_2gram.fit_transform(text_data)

# Вывести матрицу признаков
bag.toarray()
array([[2],
       [0],
       [0]])

# Вывести 1-граммы и 2-граммы
count_2gram.vocabulary_
{'brazil': 0}
```

Дополнительные материалы для чтения

- *n*-gram (*N*-грамма, Википедия; **https://oreil.ly/XWIrM**).
- Bag of Words Meets Bags of Popcorn ("Мешок слов встречается с мешком попкорна"; **https://oreil.ly/IiyRV**).

6.10. Оценка значимости слов

Задача

Нам необходимо узнать значимость элементов мешка слов для наблюдений.

Решение

Мы можем сравнить частоту использования слова в документе (твит, отзыв на фильм, расшифровка речи и т. д.) с частотой его использования в других документах при помощи статистической меры *tf-idf* (term frequency — inverse document frequency), используемой для оценки важности слова в контексте документа. Для решения этой задачи в scikit-learn есть класс `TfidfVectorizer`:

```
# Загрузить библиотеки
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
# Создать текст
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])

# Создать матрицу признаков tf-idf
tfidf = TfidfVectorizer()
feature_matrix = tfidf.fit_transform(text_data)

# Вывести матрицу признаков tf-idf
feature_matrix
```

```
<3x8 sparse matrix of type '<class 'numpy.float64'>'
    with 8 stored elements in Compressed Sparse Row format>
```

Как и в *рецепте 6.9*, на выходе мы получили разреженную матрицу. Если нам нужно вывести плотную матрицу, используем метод toarray:

```
# Вывести плотную матрицу признаков tf-idf
feature_matrix.toarray()
```

```
array([[ 0. ,         0. ,         0. ,         0.89442719,  0. ,
         0. ,         0.4472136 ,  0. ],
       [ 0. ,         0.57735027,  0. ,         0. ,         0. ,
         0.57735027,  0. ,         0.57735027],
       [ 0.57735027,  0. ,         0.57735027,  0. ,         0.57735027,
         0. ,         0. ,         0. ]])
```

Используем атрибут vocabulary_ для вывода слов признаков:

```
# Вывести названия признаков
tfidf.vocabulary_
```

```
{'love': 6, 'brazil': 3, 'sweden': 7, 'is': 5, 'best': 1, 'germany': 4, 'beats': 0,
'both': 2}
```

Обсуждение

Значимость слова в документе повышается с частотой его использования. Например, если в тексте часто упоминается слово *"экономика"*, мы можем предположить, что в документе идет речь про экономику. Мы называем это *частотой слова* (term frequency, tf).

Напротив, если же слово появляется в разных документах, оно скорее всего не так значимо для отдельного документа. К примеру, когда в тексте каждого документа содержится слово "после", его значимость с большой вероятностью снижается. Мы называем это *частотой документа* (document frequency, df).

Вместе эти метрики позволяют нам приписать значение слову, которое отражает его значимость в документе. Мы, в частности, умножаем частоту слова *tf* на обратную частоту документа (inverse of document frequency, idf):

$$tf\text{-}idf(t, d) = tf(t, d) \times idf(t),$$

где t — слово (элемент); d — документ. Существуют разные варианты подсчета показателей *tf* и *idf*. В scikit-learn *tf* означает количество вхождений слова в документе, а *idf* высчитывается следующим образом:

$$idf(t) = \log \frac{1 + n_d}{1 + df(d, t)} + 1,$$

где n_d — количество документов; $df(d, t)$ — количество документов с элементом t.

По умолчанию в библиотеке scikit-learn используется нормализация векторов *tf-idf* при помощи евклидовой нормы (норма L2). Чем выше итоговое значение, тем значимее слово в документе.

Дополнительные материалы для чтения

- Tf-idf term weighting ("Оценка терминов при помощи tf-idf", документация scikit-learn; **https://oreil.ly/40WeT**).

6.11. Использование векторизации текстов для измерения соответствия текста поисковому запросу

Задача

Вы хотите использовать векторы *tf-idf* для внедрения функции текстового поиска в Python.

Решение

При помощи scikit-learn вычислим косинусный коэффициент (cosine similarity) сходства между векторами *tf-idf*:

```
# Загрузить библиотеку
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import linear_kernel

# Создать текстовые данные с возможностью поиска
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])
```

```
# Создать матрицу признаков tf-idf
tfidf = TfidfVectorizer()
feature_matrix = tfidf.fit_transform(text_data)

# Создать поисковый запрос и преобразовать его в вектор tf-idf
text = "Brazil is the best"
vector = tfidf.transform([text])

# Рассчитать косинусный коэффициент сходства между входным вектором
# и остальными векторами
cosine_similarities = linear_kernel(vector, feature_matrix).flatten()

# Получить индексы элементов в порядке их релевантности
related_doc_indicies = cosine_similarities.argsort()[:-10:-1]

# Вывести косинус и наилучшие совпадения с запросом
print([(text_data[i], cosine_similarities[i]) for i in related_doc_indicies])
```

```
[
    ('Sweden is best', 0.6666666666666666),
    ('I love Brazil. Brazil!', 0.5163977794943222),
    ('Germany beats both', 0.0)
]
```

Обсуждение

Текстовые векторы весьма полезны при обработке текстов на естественном языке, например в поисковых машинах. После расчета векторов *tf-idf* из наборов предложений или документов мы можем использовать полученные на их основе объекты `tfidf` для векторизации других наборов текстов. Затем мы можем узнать косинусный коэффициент сходства между входным вектором и матрицей остальных векторов, а также сортировать получившиеся результаты по релевантности.

Косинусный коэффициент имеет диапазон значений [0; 1,0], где 0 означает наименьшее совпадение, а 1 — наибольшее. Поскольку при вычислении схожести векторов мы используем векторы *tf-idf*, важную роль также играет частота появления слова. Конечно, в случае небольших корпусов (набора документов) даже "часто используемое" слово встречается не так уже часто. В нашем примере поисковому запросу "Brazil is the best" наиболее соответствует фраза "Sweden is best". Мы могли бы предположить, что из-за упоминания слова "Brazil" наиболее релевантным должен быть результат "I love Brazil. Brazil!"; на практике наибольшее сходство имеет фраза "Sweden is best" ввиду наличия слов "is" и "best". С увеличением количества документов в нашем корпусе влияние менее значимых слов при вычислении косинусного коэффициента будет снижаться.

Дополнительные материалы для чтения

- Cosine Similarity, Geeks for Geeks ("Косинусный коэффициент сходства"; **https://oreil.ly/-5Odv**).
- Nvidia Gave Me a $15K Data Science Workstation — Here's What I Did with It ("Я получил 15 000 долларов от Nvidia на эксперименты с данными — что из этого вышло", создание поисковой машины на основе Pubmed в Python); **https://oreil.ly/pAxbR**).

6.12. Использование анализа тональности при классификации

Задача

Вам нужно произвести классификацию тональности (sentiment) текста для создания признака или последующего анализа данных.

Решение

Используем библиотеку transformers для классификации тональности.

```
# Импортировать библиотеки
from transformers import pipeline

# Создать NLP-конвейер для запуска анализа тональности
classifier = pipeline("sentiment-analysis")

# Классифицировать текст (при первом использовании
# может понадобиться скачивание дополнительных данных и модели)
sentiment_1 = classifier("I hate machine learning! It's the absolute worst.")
sentiment_2 = classifier(
    "Machine learning is the absolute"
    "bees knees I love it so much!" )

# Вывести результат анализа
print(sentiment_1, sentiment_2)
```

```
[
  {
    'label': 'NEGATIVE',
    'score': 0.9998020529747009
  }
```

```
]
[
  {
    'label': 'POSITIVE',
    'score': 0.9990628957748413
  }
]
```

Обсуждение

Библиотека transformers весьма часто используется в задачах обработки текстов на естественном языке и содержит набор простых API для работы с готовыми моделями или обучения собственных. Мы рассмотрим инструменты для обработки текстов на естественном языке более подробно в *главе 22*; пример выше призван продемонстрировать возможности заранее обученных классификаторов, которые вы можете использовать в процессе машинного обучения для создания признаков, классификации текстов или анализа неструктурированных данных.

Дополнительные материалы для чтения

- Обзор преобразователей Hugging Face (**https://oreil.ly/7hT6W**).

ГЛАВА 7

Обработка данных даты и времени

Введение

В процессе первичной обработки данных для машинного обучения нам часто встречаются различные показатели даты и времени, например время определенной покупки или даты в статистике системы здравоохранения. *Панельные данные* (longitudinal data), или *данные временны́х рядов* (time series data), — это данные, собираемые для одних и тех же переменны́х за несколько периодов времени. В этой главе мы выработаем стратегии работы с данными временны́х рядов, включая часовые пояса, и создание признаков с функцией с запаздыванием по времени (lagged time features). В частности, мы более подробно разберем инструменты для работы с панельными данными библиотеки pandas, которая имеет функциональность многих других стандартных библиотек вроде `datetime`.

7.1. Преобразование строк в даты

Задача

У вас есть вектор, содержащий даты и время в строковом формате (string), и вы хотите преобразовать эти данные во временны́е ряды.

Решение

Используем функцию `to_datetime` библиотеки pandas, задав формат даты и/или времени через параметр `format`:

```
# Загрузить библиотеку
import numpy as np
import pandas as pd

# Создать набор строковых данных
date_strings = np.array(['03-04-2005 11:35 PM',
                         '23-05-2010 12:01 AM',
                         '04-09-2009 09:09 PM'])
```

```
# Преобразовать данные в формат даты и времени
[pd.to_datetime(date, format='%d-%m-%Y %I:%M %p') for date in date_strings]
```

```
[Timestamp('2005-04-03 23:35:00'),
 Timestamp('2010-05-23 00:01:00'),
 Timestamp('2009-09-04 21:09:00')]
```

Мы также можем добавить аргумент в параметр `errors` на случай сбоев:

```
# Преобразовать данные в формат даты и времени
[pd.to_datetime(date, format="%d-%m-%Y %I:%M %p", errors="coerce") for date in
date_strings]
```

```
[Timestamp('2005-04-03 23:35:00'),
 Timestamp('2010-05-23 00:01:00'),
 Timestamp('2009-09-04 21:09:00')]
```

Если мы добавим `errors="coerce"`, появление сбоев не приведет к ошибке (поведение по умолчанию), а вместо этого проблемное значение изменится на `NaT` (отсутствующее значение). Таким образом мы сможем присваивать нулевое значение всем выбросам, чтобы не исправлять ошибки в каждой отдельной записи.

Обсуждение

В случае использования строкового типа для даты и времени нам необходимо преобразовать их в понятный для Python формат. В Python существует много инструментов для преобразования строк во временнóй формат, однако в наших рецептах мы работаем с pandas, поэтому предлагаем здесь воспользоваться функцией `to_datetime`. Данные даты и времени в строковом формате могут иметь совершенно разный вид в различных источниках, что осложняет работу с ними. К примеру, в одном векторе 25 марта 2015 может иметь вид "03-23-15", тогда как в другом это будет "3|23|2015". Мы можем указать конкретный формат строки при помощи параметра `format`. В табл. 7.1 показаны наиболее распространенные символы форматирования даты-времени.

Таблица 7.1

Код	Описание	Пример
%Y	Год	2001
%m	Месяц с дополнительным нулем	04
%d	День месяца с дополнительным нулем	09
%I	Час (12-часовой формат) с дополнительным нулем	02
%p	AM (до полудня) или PM (после полудня)	AM
%	Минута с дополнительным нулем	05
%S	Секунда с дополнительным нулем	09

Дополнительные материалы для чтения

- Python strftime Cheatsheet (полный список строкового форматирования даты-времени в Python; **https://oreil.ly/4-tN6**).

7.2. Работа с часовыми поясами

Задача

У вас есть данные временны́х рядов, и вы хотите добавить или изменить информацию о часовом поясе.

Решение

По умолчанию в объектах pandas не указываются часовые пояса. Для того чтобы добавить часовой пояс, в процессе создания укажем параметр `tz`:

```
# Загрузить библиотеку
import pandas as pd

# Создать данные даты и времени
pd.Timestamp('2017-05-01 06:00:00', tz='Europe/London')
```

```
Timestamp('2017-05-01 06:00:00+0100', tz='Europe/London')
```

При вызове метода `tz_localize` мы можем добавить часовой пояс к созданным ранее данным даты и времени:

```
# Создать данные даты и времени
date = pd.Timestamp('2017-05-01 06:00:00')

# Настроить часовой пояс
date_in_london = date.tz_localize('Europe/London')

# Вывести на экран данные даты и времени
date_in_london
```

```
Timestamp('2017-05-01 06:00:00+0100', tz='Europe/London')
```

Мы также можем изменить часовой пояс:

```
# Изменить часовой пояс
date_in_london.tz_convert('Africa/Abidjan')
```

```
Timestamp('2017-05-01 05:00:00+0000', tz='Africa/Abidjan')
```

Наконец, методы `tz_localize` и `tz_convert` можно применять ко всем элементам объектов библиотеки pandas для представления одномерных структур данных Series:

```
# Создать три даты
dates = pd.Series(pd.date_range('2/2/2002', periods=3, freq='M'))
```

```
# Настроить часовой пояс
dates.dt.tz_localize('Africa/Abidjan')
0   2002-02-28 00:00:00+00:00
1   2002-03-31 00:00:00+00:00
2   2002-04-30 00:00:00+00:00
dtype: datetime64[ns, Africa/Abidjan]
```

Обсуждение

В pandas есть два набора строковых данных для отображения часовых поясов; я, в свою очередь, предлагаю использовать библиотеку pytz. Для доступа к сроковым данным всех часовых поясов подключим библиотеку all_timezones:

```
# Загрузить библиотеку
from pytz import all_timezones

# Вывести два часовых пояса
all_timezones[0:2]
['Africa/Abidjan', 'Africa/Accra']
```

7.3. Выбор даты и времени

Задача

У вас есть вектор данных с датами, из которых необходимо выбрать одну или несколько.

Решение

Добавим логические условия в качестве начальной и конечной даты:

```
# Загрузить библиотеку
import pandas as pd

# Создать DataFrame
dataframe = pd.DataFrame()

# Создать данные даты и времени
dataframe['date'] = pd.date_range('1/1/2001', periods=100000, freq='H')

# Вывести наблюдения за определенный интервал времени
dataframe[(dataframe['date'] > '2002-1-1 01:00:00') &
          (dataframe['date'] <= '2002-1-1 04:00:00')]
```

```
                    date
8762 2002-01-01 02:00:00
8763 2002-01-01 03:00:00
8764 2002-01-01 04:00:00
```

Кроме того, мы можем указать данные столбца даты в качестве индексов DataFrame и выбирать срезы при помощи свойства `loc`:

```
# Настроить индексы
dataframe = dataframe.set_index(dataframe['date'])

# Выбрать наблюдения между определенными датами
dataframe.loc['2002-1-1 01:00:00':'2002-1-1 04:00:00']
```

```
                                    date
date
2002-01-01 01:00:00 2002-01-01 01:00:00
2002-01-01 02:00:00 2002-01-01 02:00:00
2002-01-01 03:00:00 2002-01-01 03:00:00
2002-01-01 04:00:00 2002-01-01 04:00:00
```

Обсуждение

Выбор между логическими выражениями и срезами по индексам зависит от конкретной ситуации. Для сложных манипуляций над данными временны́х рядов стоит уделить внимание настройке столбца даты в качестве индексов DataFrame, однако для простых операций при первичной обработке данных будет достаточно относительно простых логических условий.

7.4. Разложение временных данных на отдельные признаки

Задача

У вас есть столбец с данными даты и времени, из которого вы хотите получить отдельные признаки для года, месяца, дня, часа и минуты.

Решение

Используем метод `Series.dt` в pandas:

```
# Загрузить библиотеку
import pandas as pd
```

```
# Создать DataFrame
dataframe = pd.DataFrame()

# Создать пять дат
dataframe['date'] = pd.date_range('1/1/2001', periods=150, freq='W')

# Создать признаки для года, месяца, дня, часа и минуты
dataframe['year'] = dataframe['date'].dt.year
dataframe['month'] = dataframe['date'].dt.month
dataframe['day'] = dataframe['date'].dt.day
dataframe['hour'] = dataframe['date'].dt.hour
dataframe['minute'] = dataframe['date'].dt.minute

# Вывести три строки
dataframe.head(3)
```

	date	year	month	day	hour	minute
0	2001-01-07	2001	1	7	0	0
1	2001-01-14	2001	1	14	0	0
2	2001-01-21	2001	1	21	0	0

Обсуждение

В некоторых случаях нам может пригодиться разложить столбец даты на составляющие. Например, мы захотим получить признаки с указанием только года или только месяца наблюдения для абстрагирования от других параметров при сравнении.

7.5. Вычисление разницы между датами

Задача

Имея два временны́х признака, вы хотите рассчитать время между этими наблюдениями.

Решение

При помощи pandas вычтем два временны́х признака:

```
# Загрузить библиотеку
import pandas as pd

# Создать DataFrame
dataframe = pd.DataFrame()
```

```
# Создать два временных признака
dataframe['Arrived'] = [pd.Timestamp('01-01-2017'), pd.Timestamp('01-04-2017')]
dataframe['Left'] = [pd.Timestamp('01-01-2017'), pd.Timestamp('01-06-2017')]

# Вычислить временной интервал между признаками
dataframe['Left'] - dataframe['Arrived']
```

```
0   0 days
1   2 days
dtype: timedelta64[ns]
```

Мы можем избавиться от days в выводе функции, оставив только числовые значения:

```
# Вычислить временной интервал между признаками
pd.Series(delta.days for delta in (dataframe['Left'] - dataframe['Arrived']))
```

```
0    0
1    2
dtype: int64
```

Обсуждение

Иногда интересующий нас признак может являться промежутком (дельтой) между двумя временны́ми точками. К примеру, нам известны дата въезда и дата выезда постояльца отеля, и мы хотим узнать продолжительность его пребывания. Задача вычисления данных для нас существенно упрощается благодаря классу `TimeDelta` в pandas.

Дополнительные материалы для чтения

- Time Deltas ("Дельты времени", документация pandas; **https://oreil.ly/fbgp-**).

7.6. Вычисление дней недели

Задача

У вас есть вектор с датами, которые вы хотите сопоставить с днями недели.

Решение

Воспользуемся методом `day_name()` в классе `Series.dt` библиотеки pandas:

```
# Загрузить библиотеку
import pandas as pd

# Создать даты
dates = pd.Series(pd.date_range("2/2/2002", periods=3, freq="M"))
```

```
# Вывести дни недели
dates.dt.day_name()
0    Thursday
1      Sunday
2     Tuesday
dtype: object
```

Если нам нужно получить данные о днях недели в числовом формате для последующего использование признака в машинном обучении, применим метод `weekday` (понедельник (monday) — 0):

```
# Вывести дни недели
dates.dt.weekday
0    3
1    6
2    1
dtype: int64
```

Обсуждение

Сортировка по дням недели может пригодиться, когда мы, например, хотим подсчитать объем продаж в воскресенье за последние три года. Инструменты pandas упрощают процесс создания вектора признаков с информацией о днях недели.

Дополнительные материалы для чтения

- pandas Series datetimelike properties ("Функции времени в pandas Series"; **https://oreil.ly/3Au86**).

7.7. Создание признаков с запаздыванием по времени

Задача

Вам необходимо создать признак, который запаздывает на *n* периодов времени.

Решение

Воспользуемся методом `shift` библиотеки pandas:

```
# Загрузить библиотеку
import pandas as pd

# Создать DataFrame
dataframe = pd.DataFrame()
```

```
# Создать данные: даты и цена акций
dataframe["dates"] = pd.date_range("1/1/2001", periods=5, freq="D")
dataframe["stock_price"] = [1.1,2.2,3.3,4.4,5.5]

# Создать столбец с запаздыванием на одну строку: цена акций в предыдущий день
dataframe["previous_days_stock_price"] = dataframe["stock_price"].shift(1)

# Вывести DataFrame
dataframe
```

	dates	stock_price	previous_days_stock_price
0	2001-01-01	1.1	NaN
1	2001-01-02	2.2	1.1
2	2001-01-03	3.3	2.2
3	2001-01-04	4.4	3.3
4	2001-01-05	5.5	4.4

Обсуждение

В основе данных часто лежат равномерно структурированные временны́е промежутки (например, каждый день, час, три часа), и мы можем воспользоваться значениями прошлых данных для создания прогнозов (так называемые признаки с *запаздыванием*). Например, мы можем спрогнозировать стоимость ценных бумаг на основе данных за предыдущий день. При помощи метода `shift` в pandas мы можем получить набор новых признаков с прошлыми значениями, создав задержку в одну строку.

В нашем примере первое значение столбца `previous_days_stock_price` отсутствует, т. к. нет более раннего соответствующего значения.

7.8. Использование метода скользящего окна

Задача

Вам нужно получить скользящее среднее (rolling time) наблюдений за определенный период.

Решение

Воспользуемся методом `rolling` библиотеки pandas для DataFrame:

```
# Загрузить библиотеку
import pandas as pd
```

```
# Создать данные даты и времени
time_index = pd.date_range("01/01/2010", periods=5, freq="M")

# Создать DataFrame и указать индексы
dataframe = pd.DataFrame(index=time_index)

# Создать признаки
dataframe["Stock_Price"] = [1,2,3,4,5]

# Вычислить скользящее среднее
dataframe.rolling(window=2).mean()
```

```
            Stock_Price
2010-01-31          NaN
2010-02-28          1.5
2010-03-31          2.5
2010-04-30          3.5
2010-05-31          4.5
```

Обсуждение

Концепция *метода скользящего окна* (rolling (или moving) time windows) довольно простая, однако ее не всегда получается понять с первого раза. Представим, что у нас есть данные стоимости акций по месяцам. В такой ситуации мы можем использовать временно́е окно в несколько месяцев, которое мы можем перемещать по наблюдениям, чтобы в результате получить статистические данные всех элементов в этом промежутке.

К примеру, если у нас есть временно́е окно размером в три месяца и мы хотим получить скользящее среднее, мы можем высчитать следующие значения:

```
mean(January, February, March)
mean(February, March, April)
mean(March, Apri, May)
```

т. д.

Еще одним способом будет перемещение трехмесячного временно́го промежутка вдоль наблюдений с вычислением среднего значения окна на каждом этапе.

При помощи параметра `window` в методе `rolling` в pandas мы можем задать размер окна и получить некоторые статистические данные, включая наибольшее значение (`max()`), среднее значение (`mean()`), счетчик значений (`count()`) и скользящие корреляции (`corr()`).

Метод скользящих средних обычно используется со сглаженными временными рядами, поскольку при работе со всем окном можно снизить эффект краткосрочных колебаний.

Дополнительные материалы для чтения

- Rolling Windows ("Скользящие окна", документация pandas; **https://oreil.ly/a5gZQ**).
- What Are Moving Average or Smoothing Techniques? ("Что такое методы скользящих средних?"; **https://oreil.ly/aoOSe**).

7.9. Обработка пропущенных значений во временных рядах

Задача

У вас есть пропуски во временны́х рядах.

Решение

Помимо описанных ранее стратегий добавления пропущенных значений, при работе с временны́ми рядами мы можем использовать интерполяцию (interpolation) для заполнения пропусков:

```
# Загрузить библиотеку
import pandas as pd
import numpy as np

# Создать данные
time_index = pd.date_range("01/01/2010", periods=5, freq="M")

# Создать DataFrame и указать индексы
dataframe = pd.DataFrame(index=time_index)

# Создать признаки с пропусками
dataframe["Sales"] = [1.0,2.0,np.nan,np.nan,5.0]

# Интерполировать пропущенные значения
dataframe.interpolate()
```

```
            Sales
2010-01-31    1.0
2010-02-28    2.0
2010-03-31    3.0
2010-04-30    4.0
2010-05-31    5.0
```

Кроме того, мы можем заменить пропуски предшествующими известными значениями (так называемое forward filling — перенаправление данных из предыдущего столбца или строки):

```
# Перенаправить предыдущие данные
dataframe.ffill()
            Sales
2010-01-31    1.0
2010-02-28    2.0
2010-03-31    2.0
2010-04-30    2.0
2010-05-31    5.0
```

Мы также можем заменить пропуски последующими известными значениями (так называемое backfilling — перенаправление данных из следующего столбца или строки):

```
# Перенаправить последующие данные
dataframe.bfill()
            Sales
2010-01-31    1.0
2010-02-28    2.0
2010-03-31    5.0
2010-04-30    5.0
2010-05-31    5.0
```

Обсуждение

При помощи *интерполяции* мы можем восстановить пропущенные значения путем создания прямой или кривой между известными граничащими значениями и вычисления на ее основе подходящих значений. Этот способ особенно удобен при наличии постоянных временны́х интервалов, отсутствии колебаний данных из-за шумов и небольшом размере пропусков. В нашем примере два пропущенных значения граничат со значениями `2.0` и `5.0`. При вычислении линии между `2.0` и `5.0` мы можем предположить, что пропущенные значения находятся в диапазоне от `3.0` до `4.0`.

Если мы уверены, что кривая между двумя точками нелинейна, мы можем использовать метод `interpolate`, задав нужный параметр:

```
# Интерполировать пропущенные значения
dataframe.interpolate(method="quadratic")
              Sales
2010-01-31  1.000000
2010-02-28  2.000000
2010-03-31  3.059808
2010-04-30  4.038069
2010-05-31  5.000000
```

Наконец, у нас может быть большой пропуск, но без необходимости интерполяции значений целиком. В таком случае при помощи параметра `limit` мы можем ограничить число значений для интерполяции, а посредством параметра `limit_direction` указать направление интерполяции — из известного значения, предшествующего пропуску (forward) или наоборот:

```
# Интерполировать пропущенные значения
dataframe.interpolate(limit=1, limit_direction="forward")
            Sales
2010-01-31    1.0
2010-02-28    2.0
2010-03-31    3.0
2010-04-30    NaN
2010-05-31    5.0
```

Перенаправление последующих (backfilling) и предшествующих (forward) значений является разновидностью простой интерполяции, когда у нас получается ровная прямая из известного значения для заполнения пропусков. Одним (незначительным) преимуществом перенаправления перед интерполяцией является необходимость наличия только *одного* значения вокруг пропуска.

ГЛАВА 8

Работа с изображениями

Введение

Классификация изображений — одна из самых увлекательных областей машинного обучения. Умение компьютера распознавать паттерны и объекты значительно обогащает наш инструментарий. Однако, чтобы использовать машинное обучение при работе с изображениями, нам, как правило, необходимо преобразовать исходные изображения в более понятные для алгоритмов наборы признаков. Подобно текстовым данным существует множество предварительно обученных классификаторов изображений, из которых мы можем извлечь необходимые признаки или объекты для собственных моделей.

При работе с изображениями мы в первую очередь будем использовать библиотеку алгоритмов компьютерного зрения с открытым исходным кодом OpenCV (Open Source Computer Vision Library). Среди всего представленного множества хороших библиотек для работы с изображениями OpenCV имеет наибольшую популярность и подробную документацию. Несмотря на определенную сложность установки, вы всегда можете найти решение возникающего вопроса в многочисленных инструкциях на просторах Интернета. В процессе написания данной книги использовался пакет `opencv-python-headless==4.7.0.68`. При помощи инструмента автоматизации проверки кода ML in Python Cookbook Runner (**https://oreil.ly/MLwPython**) вы можете запустить команды из глав и удостовериться в их работоспособности.

Все используемые в этой главе наборы изображений доступны для скачивания с GitHub (**https://oreil.ly/gV5Zc**).

8.1. Загрузка изображений

Задача

Вы хотите загрузить изображение для предварительной обработки.

Решение

Воспользуемся функцией `imread` библиотеки OpenCV:

```
# Загрузить библиотеки
import cv2
```

```
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)
```

Для просмотра изображения мы можем использовать написанную на языке программирования Python библиотеку Matplotlib для визуализации данных двумерной (рис. 8.1) и трехмерной графикой:

```
# Вывести изображение
plt.imshow(image, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.1

Обсуждение

Изображения, в сущности, являются данными, которые при помощи `imread` можно преобразовать в хорошо знакомый нам формат массивов NumPy:

```
# Вывести тип данных
type(image)
numpy.ndarray
```

Мы преобразовали изображение в матрицу, элементы которой соответствуют отдельным пикселам. При желании мы даже можем просмотреть значения этой матрицы:

```
# Вывести данные изображения
image
array([[140, 136, 146, ..., 132, 139, 134],
       [144, 136, 149, ..., 142, 124, 126],
       [152, 139, 144, ..., 121, 127, 134],
       ...,
       [156, 146, 144, ..., 157, 154, 151],
       [146, 150, 147, ..., 156, 158, 157],
       [143, 138, 147, ..., 156, 157, 157]], dtype=uint8)
```

Размер матрицы равен 3600 × 2270, что соответствует разрешению картинки:

```
# Вывести размер
image.shape
```

```
(2270, 3600)
```

Что же в действительности представляет собой каждый элемент матрицы? В случае изображений в оттенках серого значение каждого отдельного пиксела соответствует оттенку серого. Оттенок находится в диапазоне от черного (0) до белого (255). К примеру, оттенок левого верхнего пиксела нашего изображения соответствует значению 140:

```
# Вывести первый пиксел
image[0,0]
```

```
140
```

Если матрица представляет изображение в цветовом пространстве RGB, то каждый элемент содержит три значения, соответствующие синему (B — blue), зеленому (G — green) и красному (R — red) цветам:

```
# Загрузить цветное изображение
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)

# Вывести пикселы
image_bgr[0,0]
```

```
array([195, 144, 111], dtype=uint8)
```

Обратите внимание, что по умолчанию в OpenCV используется BGR, в то время как во многих приложениях для работы с изображениями, в том числе Matplotlib, используется RGB — другими словами, значения для красного и синего цветов меняются местами.

Рис. 8.2

Для корректного отображения изображений при помощи OpenCV в Matplotlib нам сначала необходимо преобразовать цветовую модель в RGB (приношу извинения владельцам печатных копий, в которых нет цветных изображений), рис. 8.2:

```
# Преобразовать в RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
```

```
# Вывести изображение
plt.imshow(image_rgb)
plt.axis("off")
plt.show()
```

Дополнительные материалы для чтения

- Разница между RGB и BGR (**https://oreil.ly/N1Ub6**).
- RGB (Википедия; **https://oreil.ly/OEesQ**).

8.2. Сохранение изображений

Задача

Вам необходимо сохранить изображение для предварительной обработки.

Решение

Используем функцию `imwrite` библиотеки OpenCV:

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)

# Сохранить изображение
cv2.imwrite("images/plane_new.jpg", image)
True
```

Обсуждение

При помощи `imwrite` из OpenCV мы можем сохранить изображение по указанному пути. Формат изображения определяется расширением (jpg, png и т. д.), указанным в имени файла. Следует помнить, что функция `imwrite` может перезаписать существующее изображение без вывода ошибки и необходимости подтверждения.

8.3. Измерение размера изображений

Задача

Вам нужно изменить размер изображения для последующей первичной обработки.

Решение

Воспользуемся функцией для изменения размера изображения (рис. 8.3):

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Изменить размер на 50x50 пикселов
image_50x50 = cv2.resize(image, (50, 50))

# Вывести изображение
plt.imshow(image_50x50, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.3

Обсуждение

В процессе первичной обработки часто приходится изменять размер изображения, и на это есть две причины. Во-первых, к нам могут попадать изображения различных форм и размеров, однако создаваемые на их основе признаки должны быть одной размерности. При стандартизации (изменении размера) изображение теряет

определенное количество информации в сравнении с исходником большего разрешения, что можно наблюдать на примере самолета на рис. 8.3. Поскольку изображение фактически является матрицей с информацией, уменьшение его размера означает сокращение числа элементов матрицы, содержащих информацию. Во-вторых, для обучения модели могут понадобиться тысячи и сотни тысяч изображений. Обработка изображений в большом разрешении требует немалого количества памяти, однако мы можем существенно сократить необходимые объемы благодаря уменьшению разрешения изображения. В машинном обучении часто используются размеры 32 × 32, 64 × 64, 96 × 96 и 256 × 256. В сущности, выбор метода уменьшения изображения является компромиссом между сохранением информативности нашей модели и затратами вычислительных мощностей на ее обучение. Для этого в библиотеке Pillow существует множество опций для изменения размера изображения (**https://oreil.ly/NiJn_**).

8.4. Кадрирование изображений

Задача

Вы хотите удалить внешние части изображения для изменения его размера.

Решение

Изображение кодируется в виде двумерного массива NumPy, поэтому мы можем легко кадрировать изображение при помощи срезов массива (рис. 8.4):

Рис. 8.4

```
# Загрузить библиотеку
import cv2
import numpy as np from matplotlib
import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Выбрать первую половину столбцов и все строки
image_cropped = image[:,:128]

# Вывести изображение
plt.imshow(image_cropped, cmap="gray")
plt.axis("off")
plt.show()
```

Обсуждение

Поскольку в OpenCV изображения представлены в виде матрицы элементов, мы легко можем кадрировать изображение, выбирая строки и столбцы, которые необходимо оставить. Это особенно полезно в том случае, когда мы знаем, что нужно оставить лишь конкретную часть каждого изображения. Например, это могут быть кадры камеры видеонаблюдения, на которых нас интересует только определенная область.

Дополнительные материалы для чтения

- Slicing NumPy ("Срез массивов NumPy"; **https://oreil.ly/8JN5p**).

8.5. Размытие изображений

Задача

Вы хотите размыть изображение.

Решение

В процессе размытия (также называемого сглаживанием или усреднением в математике) изображения каждый элемент заменяется средним значением соседних пикселов. Эти соседние значения и операции их преобразования называются ядрами свертки (kernel) в математике (не стоит волноваться, если понятие ядра вам не знакомо). Размер ядра определяет величину усреднения (blur), т. е. с увеличением размеров ядер картинка становится более размытой. Мы попробуем применить размытие изображения, указав размер ядра 5 × 5 для каждого пиксела (рис. 8.5):

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Усреднить изображение
image_blurry = cv2.blur(image, (5,5))

# Вывести изображение
plt.imshow(image_blurry, cmap="gray")
plt.axis("off")
plt.show()
```

Для того чтобы лучше продемонстрировать эффект (рис. 8.6), изменим значение размера ядра на 100 × 100:

```
# Усреднить изображение
image_very_blurry = cv2.blur(image, (100,100))

# Вывести изображение
plt.imshow(image_very_blurry, cmap="gray")
plt.xticks([])
plt.yticks([])
plt.show()
```

Рис. 8.5

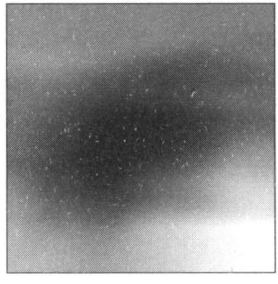

Рис. 8.6

Обсуждение

В этой главе мы еще не раз встретим ядра, поскольку они повсеместно используются при первичной обработке изображений, от настройки резкости до обнаружения границ. Используемое нами ядро для усреднения выглядит следующим образом:

```
# Создать ядро
kernel = np.ones((5,5)) / 25.0

# Вывести ядро на экран
kernel
array([[ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04]])
```

В центре ядра находится нужный нам пиксел, а остальные элементы являются соседями. Поскольку все элементы имеют одинаковое значение (нормализованы до 1), каждый из них оказывает равноценное влияние на конечное значение целевого пиксела.

Мы можем применить ядро свертки к изображению вручную при помощи функции `filter2D` для получения схожего эффекта усреднения (рис. 8.7):

```
# Применить ядро
image_kernel = cv2.filter2D(image, -1,_kernel)

# Вывести изображение
plt.imshow(image_kernel, cmap="gray")
plt.xticks([])
plt.yticks([])
plt.show()
```

Рис. 8.7

Дополнительные материалы для чтения

- Image Kernels Explained Visually ("Визуальное объяснение ядер свертки"; **https://oreil.ly/9yvdg**).
- Kernel (image processing) ("Ядро свертки", Википедия; **https://oreil.ly/ByREC**).

8.6. Настройка резкости изображений

Задача

Вам необходимо увеличить резкость изображения.

Решение

Создадим ядро, обозначающее целевой пиксел. Затем при помощи `filter2D` применим его к изображению (рис. 8.8):

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```
# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Создать ядро
kernel = np.array([[0, -1, 0],
                   [-1, 5,-1],
                   [0, -1, 0]])

# Изменить резкость изображения
image_sharp = cv2.filter2D(image, -1, kernel)

# Вывести изображение
plt.imshow(image_sharp, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.8

Обсуждение

Настройка резкости схожа со сглаживанием, однако мы используем ядро свертки не для усреднения значений соседних элементов, а для выделения пиксела. В результате границы становятся более четкими.

8.7. Увеличение контраста

Задача

Вы хотите увеличить контраст пикселов изображения.

Решение

Эквализация, или *выравнивание гистограммы* (histogram equalization), — это инструмент для обработки изображений, при помощи которого можно менять интен-

сивность выделения объектов и форм. При обработке изображения в оттенках серого мы можем применить функцию equalizeHist библиотеки OpenCV напрямую к изображению (рис. 8.9):

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Увеличить контраст изображения
image_enhanced = cv2.equalizeHist(image)

# Вывести изображение
plt.imshow(image_enhanced, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.9

Если же мы работаем с изображением в цвете, то сперва потребуется конвертировать изображение в формат цветовой модели YUV. Компонент Y означает яркость (luma или brightness), а компоненты U и V — цветовое отличие. После преобразования мы можем применить функцию equalizeHist, а затем обратно вернуть формат BGR или RGB (приношу извинения владельцам печатных копий, в которых нет цветных изображений), рис. 8.10:

```
# Загрузить изображение
image_bgr = cv2.imread("images/plane.jpg")

# Конвертировать в YUV
image_yuv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2YUV)

# Выполнить выравнивание гистограммы
image_yuv[:, :, 0] = cv2.equalizeHist(image_yuv[:, :, 0])
```

```
# Конвертировать в RGB
image_rgb = cv2.cvtColor(image_yuv, cv2.COLOR_YUV2RGB)

# Вывести изображение
plt.imshow(image_rgb)
plt.axis("off")
plt.show()
```

Рис. 8.10

Обсуждение

Подробное объяснение принципа эквализации гистограммы выходит за рамки данной книги; достаточно знать, что данная процедура преобразования позволяет более широко использовать интенсивность пикселов (pixel intensities) изображения.

Хотя получившееся изображение не всегда выглядит реалистично, нужно помнить, что это лишь визуализация лежащих в основе данных. Если эквализация гистограммы позволит лучше отличать интересующие нас объекты от других объектов или фонов (что требуется не всегда), это будет хорошим дополнением к нашему конвейеру обработки изображений.

8.8. Распознавание цветов

Задача

Вы хотите выделить какой-либо цвет изображения.

Решение

Определим диапазон цветов и применим маску к изображению (приношу извинения владельцам печатных копий, в которых нет цветных изображений), рис. 8.11:

```
# Загрузить библиотеку
import cv2
```

```
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение
image_bgr = cv2.imread('images/plane_256x256.jpg')

# Конвертировать BGR в HSV
image_hsv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2HSV)

# Определить диапазон значений синего в HSV
lower_blue = np.array([50,100,50])
upper_blue = np.array([130,255,255])

# Создать маску
mask = cv2.inRange(image_hsv, lower_blue, upper_blue)

# Применить маску к изображению
image_bgr_masked = cv2.bitwise_and(image_bgr, image_bgr, mask=mask)

# Конвертировать BGR в RGB
image_rgb = cv2.cvtColor(image_bgr_masked, cv2.COLOR_BGR2RGB)

# Вывести изображение
plt.imshow(image_rgb)
plt.axis("off")
plt.show()
```

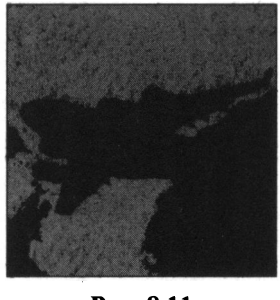

Рис. 8.11

Обсуждение

Распознавание цветов в OpenCV является достаточно тривиальной задачей. Во-первых, нам нужно преобразовать изображение в формат HSV (hue (тон), saturation (насыщенность) и value (значение)). Во-вторых, мы определяем диапазон значе-

ний, которые нужно выделить. Это, пожалуй, самая сложная и длительная часть. В-третьих, мы создаем маску для изображения. Маскирование изображения является популярной техникой извлечения нужной области. В данном случае наша маска содержит только области белого (рис. 8.12):

```
# Вывести изображение
plt.imshow(mask, cmap='gray')
plt.axis("off")
plt.show()
```

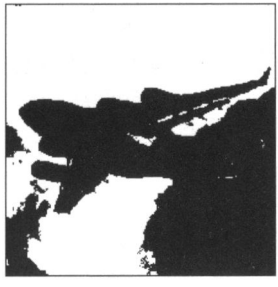

Рис. 8.12

Наконец, при помощи `bitwise_and` мы применяем маску к изображению и конвертируем его в нужный нам формат.

8.9. Бинаризация изображений

Задача

Вы хотите создать упрощенный вариант имеющегося изображения.

Решение

Пороговая обработка (thresholding) — это процесс настройки интенсивности пикселов, при которой значения одних повышаются до белого цвета, а значения других понижаются до черного цвета. Более продвинутой техникой является *адаптивная пороговая обработка* (adaptive thresholding), при использовании которой значение пиксела определяется интенсивностью соседних пикселов (рис. 8.13). Это может пригодиться при неоднородной освещенности участков изображения:

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image_grey = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)
```

```
# Применить адаптивную пороговую обработку
max_output_value = 255
neighborhood_size = 99
subtract_from_mean = 10
image_binarized = cv2.adaptiveThreshold(image_grey,
                                        max_output_value,
                                        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                        cv2.THRESH_BINARY,
                                        neighborhood_size,
                                        subtract_from_mean)

# Вывести изображение
plt.imshow(image_binarized, cmap="gray")
plt.axis("off")
plt.show()
```

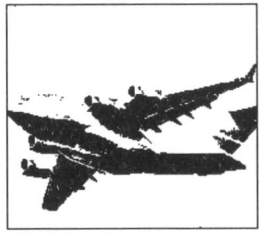

Рис. 8.13

Обсуждение

Процесс бинаризации изображения включает его преобразование из оттенков серого в черно-белый формат (рис. 8.14). В нашем примере функция `adaptiveThreshold` имеет четыре важных аргумента. Параметр `max_output_value` означает наибольшую величину итоговой интенсивности пикселов. Параметр `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` означает, что пороговое значение пиксела является взвешенной суммой значений интенсивности соседних пикселов. Веса определяются окном Гаусса (Gaussian window). Если вместо этого мы укажем `cv2.ADAPTIVE_THRESH_MEAN_C`, пороговое значение будет равно среднему значению (mean) соседних пикселов:

```
# Применить cv2.ADAPTIVE_THRESH_MEAN_C
image_mean_threshold = cv2.adaptiveThreshold(image_grey,
                                             max_output_value,
                                             cv2.ADAPTIVE_THRESH_MEAN_C,
                                             cv2.THRESH_BINARY,
                                             neighborhood_size,
                                             subtract_from_mean)
```

```
# Вывести изображение
plt.imshow(image_mean_threshold, cmap="gray")
plt.axis("off")
plt.show()
```

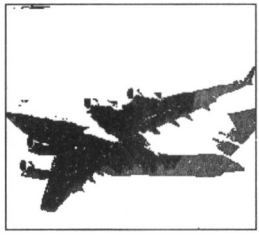

Рис. 8.14

Два последних параметра означают размер блока (охват соседних элементов) и величину постоянной, вычитаемой из полученного порогового значения (для ручной настройки пороговой обработки).

Большим преимуществом пороговой обработки является *избавление от шумов* (denoising) с сохранением только важных элементов. К примеру, пороговая обработка часто используется для изоляции букв на странице при работе с фотографиями печатного текста.

8.10. Удаление фона

Задача

Вам необходимо изолировать фон изображения.

Решение

Разметим прямоугольник вокруг нужного нам объекта на переднем плане и воспользуемся алгоритмом GrabCut (рис. 8.15):

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение и конвертировать в RGB
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Значения прямоугольника: начало координаты x, начало координаты y, ширина, высота
rectangle = (0, 56, 256, 150)
```

```
# Создать маску
mask = np.zeros(image_rgb.shape[:2], np.uint8)

# При помощи grabCut создать временный массив
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

# Запустить grabCut
cv2.grabCut(image_rgb, # Наше изображение
            mask,      # Маска
            rectangle, # Наш прямоугольник
            bgdModel,  # Временный массив для фона
            fgdModel,  # Временный массив для фона
            5,         # Количество итераций
            cv2.GC_INIT_WITH_RECT) # Используемый нашим прямоугольником режим

# Создать маску в том месте, где 0 означает точное или вероятное наличие фона,
# в противном случае возвращается 1
mask_2 = np.where((mask==2) | (mask==0), 0, 1).astype('uint8')

# Объединить изображение и новую маску для удаления фона
image_rgb_nobg = image_rgb * mask_2[:, :, np.newaxis]

# Вывести изображение
plt.imshow(image_rgb_nobg)
plt.axis("off")
plt.show()
```

Рис. 8.15

Обсуждение

Мы сразу можем заметить, что, хотя алгоритм GrabCut хорошо справляется с задачей, в левой части нашего изображения по-прежнему присутствуют элементы фона. Мы, конечно, можем вернуться и разметить эти участки вручную, однако на практике это будет набор из тысяч изображений, которые мы вряд ли сможем исправить по отдельности. В таком случае нам следует просто принять тот факт, что данные нашего изображения содержат некоторое количество фонового шума.

В рассмотренном примере мы выстраивали прямоугольник вокруг области, содержащей основные элементы. Таким образом, с точки зрения логики алгоритма GrabCut все, что находится за пределом прямоугольника, является фоном, и на этой информации основывается вычисление элементов фона внутри выделенной области. (Для понимания принципа работы алгоритма можно ознакомиться с объяснением Итая Блюменталя (Itay Blumenthal, **https://oreil.ly/DTGwb**).) После создания маски происходит разграничение областей точного/вероятного фона/основного сюжета (рис. 8.16):

```
# Вывести маску на экран
plt.imshow(mask, cmap='gray')
plt.axis("off")
plt.show()
```

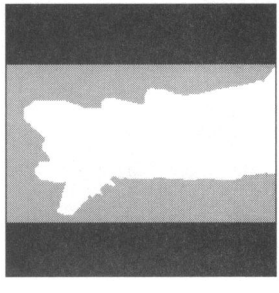

Рис. 8.16

Область черного цвета находится за пределами нашего прямоугольника и принимается за точный фон. Серая область определяется алгоритмом GrabCut как вероятный фон, а белая область — вероятный основной сюжет. Затем на основе этой маски создается вторая, которая объединяет области черного и серого цвета (рис. 8.17):

```
# Вывести маску на экран
plt.imshow(mask_2, cmap='gray')
plt.axis("off")
plt.show()
```

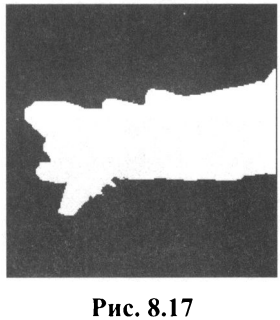

Рис. 8.17

В конце вторая маска применяется к изображению, тем самым остается только основной объект.

8.11. Определение границ

Задача

Вам нужно определить границы изображения.

Решение

Воспользуемся методом выделения границ, например детектором границ Кэнни (Canny edge detector), рис. 8.18:

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Вычислить медиану интенсивности
median_intensity = np.median(image_gray)

# Установить пороговые значения, равные единому стандартному отклонению выше
# и ниже медианы интенсивности
lower_threshold = int(max(0, (1.0 - 0.33) * median_intensity))
upper_threshold = int(min(255, (1.0 + 0.33) * median_intensity))

# Применить оператор Кэнни для выделения границ
image_canny = cv2.Canny(image_gray, lower_threshold, upper_threshold)

# Вывести изображение
plt.imshow(image_canny, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.18

Обсуждение

Определение границ является одним из основных разделов теории компьютерного зрения. Это связано с тем, что границы содержат большое количество информации. К примеру, на нашем изображении один фрагмент неба очень похож на другой и, скорее всего, не несет какой-либо уникальной или интересной информации. Напротив, части фона на границе неба и самолета могут сообщить намного больше информации (например, о форме объекта). Определение границ поможет нам избавиться от областей с несущественной информацией, изолировав наиболее информативные участки изображения.

Существует множество разных методов нахождения границ (оператор Собеля (Sobel filters), оператор Лапласа (Laplacian edge detector) и т. д.). В нашем решении мы использовали популярный оператор Кэнни. Детальное объяснение принципов работы этого оператора выходит за рамки нашей книги, однако необходимо упомянуть одну особенность. Для оператора Кэнни требуются два параметра, описывающие нижние и верхние пороговые значения градиента. Пикселы потенциальных границ между низкими и высокими пороговыми значениями рассматриваются как слабые границы, в то время как пикселы свыше максимумов являются сильными (определенными) границами. При использовании метода Кэнни в OpenCV обязательно указывать параметры нижних и верхних пороговых значений. В нашем примере нижнее и верхнее значения соответствуют стандартному отклонению ниже и выше медианы интенсивности пикселов изображения. Тем не менее мы зачастую можем добиться лучших результатов, если зададим низкие и высокие пороговые значения вручную и предварительно опробуем их на нескольких изображениях, прежде чем применять детектор границ Кэнни для всего набора.

Дополнительные материалы для чтения

- Canny Edge Detector ("Оператор Кэнни", Википедия; **https://oreil.ly/gG9xo**).
- Canny Edge Detection Auto Thresholding ("Автоматические пороговые значения оператора Кэнни"; **https://oreil.ly/YvjM5**).

8.12. Обнаружение углов

Задача

Вам необходимо обнаружить углы изображения.

Решение

Воспользуемся реализацией детектора углов Харриса (Harris corner detector) в OpenCV — функцией `cornerHarris` (рис. 8.19):

```
# Загрузить библиотеку
import cv2
```

```
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение
image_bgr = cv2.imread("images/plane_256x256.jpg")
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
image_gray = np.float32(image_gray)

# Задать параметры детектора углов
block_size = 2
aperture = 29
free_parameter = 0.04

# Определить углы
detector_responses = cv2.cornerHarris(image_gray,
                                      block_size,
                                      aperture,
                                      free_parameter)

# Увеличить индикаторы углов
detector_responses = cv2.dilate(detector_responses, None)

# Оставить полученные значения детектора только выше пороговых,
# выделить их белым цветом
threshold = 0.02
image_bgr[detector_responses >
          threshold *
          detector_responses.max()] = [255,255,255]

# Преобразовать в оттенки серого
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Вывести изображение
plt.imshow(image_gray, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.19

Обсуждение

Детектор углов Харриса является популярным методом поиска точек пересечения двух граней. Необходимость поиска углов обусловлена теми же факторами, что и определение границ, — углы содержат много информации. Ссылки на подробное описание принципа работы детектора углов Харриса можно найти в конце этого раздела. Коротко говоря, алгоритм находит окна (также называемые *окрестностями* (neighborhoods) или *областями* (patches)), небольшой сдвиг которых (моделируемое встряхивание окна) приводит к крупным изменениям внутреннего содержимого из пикселов. Функция `cornerHarris` содержит три основных параметра, которые мы можем использовать для управления выделением границ. Первый параметр, `block_size`, применяется для определения размера окрестности пиксела, используемого при выделении углов. Второй параметр, `aperture`, означает размер используемого ядра (не стоит волноваться, если это понятие вам не знакомо) оператора Собеля (Sobel kernel), и, наконец, есть третий произвольный параметр (free parameter), величина которого пропорциональна плавности углов.

В результате мы получим изображение в оттенках серого с выделением потенциальных углов (рис. 8.20):

```
# Вывести потенциальные углы
plt.imshow(detector_responses, cmap='gray')
plt.axis("off")
plt.show()
```

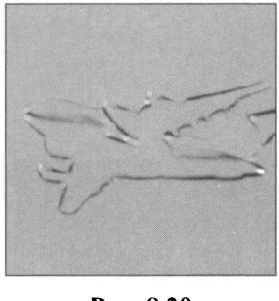

Рис. 8.20

Затем при помощи пороговой обработки мы оставляем только наиболее вероятные углы. В качестве альтернативного метода можно воспользоваться угловым детектором Ши-Томаси (Shi-Tomasi corner detector), который во многом схож с детектором углов Харриса (функция `goodFeaturesToTrack`) и определяет заданное количество сильных углов. Функция `goodFeaturesToTrack` имеет три основных параметра: количество нужных углов, минимальный уровень качества угла (от 0 до 1) и минимальное евклидово расстояние между углами:

```
# Загрузить изображения
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
```

```
# Указать количество нужных углов
corners_to_detect = 10
minimum_quality_score = 0.05
minimum_distance = 25

# Определить углы
corners = cv2.goodFeaturesToTrack(image_gray,
                                  corners_to_detect,
                                  minimum_quality_score,
                                  minimum_distance)
corners = np.int16(corners)

# Обвести каждый угол белым кругом
in corners:
    x, y = corner[0]
    cv2.circle(image_bgr, (x,y), 10, (255,255,255), -1)

# Преобразовать в оттенки серого
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Вывести изображение
plt.imshow(image_rgb, cmap='gray')
plt.axis("off")
plt.show()
```

Рис. 8.21

Дополнительные материалы для чтения

- Функция `cornerHarris` библиотеки OpenCV (**https://oreil.ly/vLMBj**).
- Функция `goodFeaturesToTrack` библиотеки OpenCV (**https://oreil.ly/Ra-x6**).

8.13. Создание признаков для машинного обучения

Задача

Вы хотите преобразовать изображение в наблюдение для машинного обучения.

Решение

Воспользуемся методом `flatten` библиотеки NumPy для преобразования многомерного массива данных изображения в вектор со значениями наблюдения:

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Изменить размер на 10x10 пикселов
image_10x10 = cv2.resize(image, (10, 10))

# Преобразовать данные изображения в одномерный вектор
image_10x10.flatten()
array([133, 130, 130, 129, 130, 129, 129, 128, 128, 127, 135, 131, 131,
       131, 130, 130, 129, 128, 128, 128, 134, 132, 131, 131, 130, 129,
       129, 128, 130, 133, 132, 158, 130, 133, 130,  46,  97,  26, 132,
       143, 141,  36,  54,  91,   9,   9,  49, 144, 179,  41, 142,  95,
        32,  36,  29,  43, 113, 141, 179, 187, 141, 124,  26,  25, 132,
       135, 151, 175, 174, 184, 143, 151,  38, 133, 134, 139, 174, 177,
       169, 174, 155, 141, 135, 137, 137, 152, 169, 168, 168, 179, 152,
       139, 136, 135, 137, 143, 159, 166, 171, 175], dtype=uint8)
```

Обсуждение

В сущности, изображения являются сеткой пикселов. Если это изображение в оттенках серого, каждый пиксел имеет одно значение (значение интенсивности пиксела равно 1 для белого цвета и 0 для черного цвета). К примеру, у нас есть картинка размером 10 × 10 пикселов (рис. 8.22):

```
plt.imshow(image_10x10, cmap="gray")
plt.axis("off")
plt.show()
```

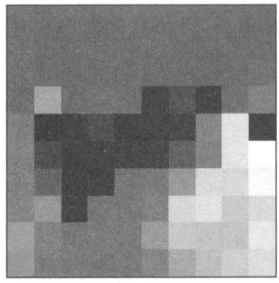

Рис. 8.22

В таком случае размерность данных также будет 10 × 10:

```
image_10x10.shape
```

```
(10, 10)
```

И если мы преобразуем массив в одномерный список, мы получим вектор из 100 элементов (10 умножим на 10):

```
image_10x10.flatten().shape
```

```
(100,)
```

В результате мы получаем данные признаков нашего изображения, которые можно добавить к векторам из других изображений, чтобы затем передать эти данные алгоритмам машинного обучения.

Если это изображение в цвете, каждый пиксел представляет множество значений (обычно три) в соответствии с числом каналов (красный (R), зеленый (G), синий (B) и т. д.), из которых получается нужный цвет пиксела. Таким образом, из изображения в цвете размером 10 × 10 мы получим 300 значений признаков для каждого наблюдения:

```
# Загрузить изображение в цвете
image_color = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Изменить размер на 10x10 пикселов
image_color_10x10 = cv2.resize(image_color, (10, 10))

# Преобразовать данные изображения в одномерный вектор и вывести размерность
image_color_10x10.flatten().shape
```

```
(300,)
```

Одна из главных сложностей обработки изображений и компьютерного зрения заключается в том, что, поскольку каждый пиксел в наборе изображений является отдельным признаком, увеличение количества изображений приводит к стремительному росту числа признаков:

```
# Загрузить изображение в оттенках серого
image_256x256_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)
```

```
# Преобразовать данные изображения в одномерный вектор и вывести размерность
image_256x256_gray.flatten().shape
```

```
(65536,)
```

И это число увеличивается еще больше, если изображение цветное:

```
# Загрузить изображение в цвете
image_256x256_color = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Преобразовать данные изображения в одномерный вектор и вывести размерность
image_256x256_color.flatten().shape
```

```
(196608,)
```

Как видно из вывода функции, даже небольшое изображение имеет почти 200 000 признаков, что может приводить к проблемам в процессе обучения наших моделей, поскольку количество признаков в некоторых случаях сильно превышает количество наблюдений.

Решение данной проблемы может заключаться в правильной работе с размерностью для сокращения количества признаков без потери существенных объемов информации. Мы разберем этот вопрос более подробно в отдельной главе.

8.14. Создание гистограмм цветов в виде признаков

Задача

Вы хотите создать набор признаков, представляющий все цвета изображения.

Решение

Вычислим гистограммы для каждого цветового канала:

```
# Загрузить библиотеку
import cv2
import numpy as np
from matplotlib import pyplot as plt

np.random.seed(0)

# Загрузить изображение
image_bgr = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Конвертировать в RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
```

```
# Создать список значений признаков
features = []

# Вычислить гистограмму для каждого цветового канала
colors = ("r","g","b")

# Для каждого канала: вычислить гистограмму и добавить ее в список значений признаков
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb], # Изображение
                             [i],         # Индекс канала
                             None,        # Без маски
                             [256],       # Размер гистограммы
                             [0,256])     # Диапазон
    features.extend(histogram)

# Создать вектор для значений признаков наблюдения
observation = np.array(features).flatten()

# Вывести значения наблюдения для первых пяти признаков
observation[0:5]
array([ 1008.,   217.,   184.,   165.,   116.], dtype=float32)
```

Обсуждение

В случае цветовой модели RGB каждый цвет получается из комбинации трех цветовых каналов (например, красный, зеленый, синий). В свою очередь, каждый канал принимает значение в диапазоне до 256 (целое число от 0 до 255). Например, левый верхний пиксел нашего изображения имеет следующие значения каналов:

```
# Вывести значения каналов RGB
image_rgb[0,0]
array([107, 163, 212], dtype=uint8)
```

Гистограмма отражает распределение значений данных. Ниже представлен простой пример:

```
# Подключить pandas
import pandas as pd
import matplotlib.pyplot as plt

# Создать набор произвольных данных
data = pd.Series([1, 1, 2, 2, 3, 3, 3, 4, 5])

# Вывести гистограмму на экран
data.hist(grid=False)
plt.show()
```

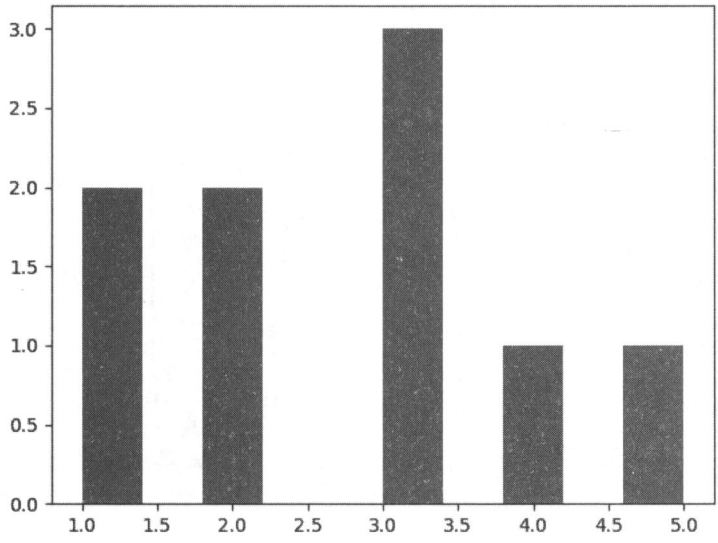

Рис. 8.23

В данном примере у нас есть набор данных, включающий два значения 1, два значения 2, три значения 3, одно значение 4 и одно значение 5. Каждый столбец означает количество вхождений значения (1, 2 и т. д.) в данных.

По аналогии мы можем применить этот метод к каждому цветовому каналу с диапазоном значений 256 (количество возможных значений канала) вместо пяти. Ось x означает величину возможного значения канала (до 256), а ось y — количество появлений данного канала в виде пикселов в изображении (приношу извинения владельцам печатных копий, в которых нет цветных изображений), рис. 8.24:

```
# Вычислить гистограмму для каждого цветового канала
colors = ("r","g","b")

# Для каждого канала: вычислить гистограмму и построить график
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb],   # Изображение
                             [i],       # Индекс канала
                             None,      # Без маски
                             [256],     # Размер гистограммы
                             [0,256])   # Диапазон
    plt.plot(histogram, color = channel)
    plt.xlim([0,256])

# Вывести график
plt.show()
```

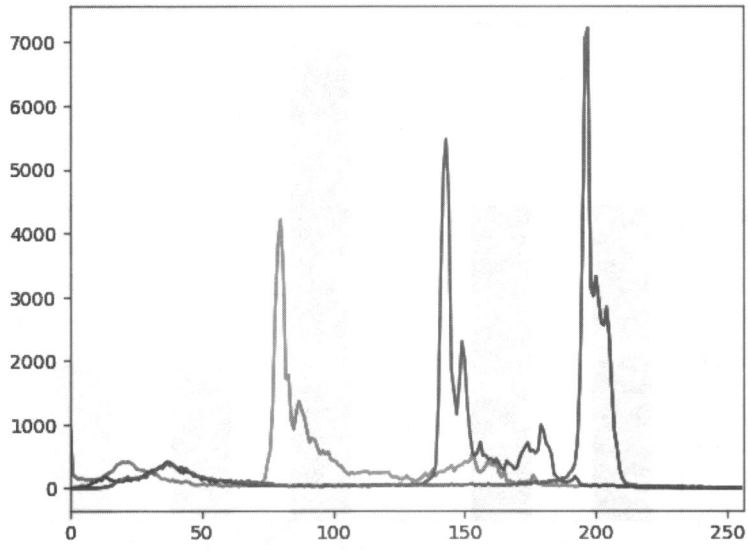

Рис. 8.24

Как мы можем видеть на гистограмме, немногие пикселы имеют синий цвет в диапазоне от 0 до примерно 180, в то время как большинство значений синего находится в примерном диапазоне от 190 до 210. Мы можем наблюдать значения распределения цветов для всех каналов. Однако гистограмма — это не просто визуализация; она содержит 256 признаков для каждого цветового канала, что дает в сумме 768 признаков для обозначения распределения цветов в изображении.

Дополнительные материалы для чтения

- Histogram ("Гистограмма", Википедия; **https://oreil.ly/nPbJT**).
- Histogram ("Гистограмма", документация pandas; **https://oreil.ly/h60M5**).
- Histogram ("Гистограмма", курс OpenCV; **https://oreil.ly/BuX1C**).

8.15. Использование предобученных эмбеддингов в качестве признаков

Задача

Вы хотите загрузить предобученные эмбеддинги (pretrained embeddings) из уже существующей в библиотеке PyTorch модели и использовать их в качестве входных данных в ваших моделях.

Решение

Воспользуемся torchvision.models для выбора модели и получения ее эмбеддинга для нашего изображения:

```
# Загрузить библиотеку
import cv2
import numpy as np
import torch
from torchvision import transforms
import torchvision.models as models

# Загрузить изображение
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)

# Преобразовать изображение в тип данных pytorch
convert_tensor = transforms.ToTensor()
pytorch_image = convert_tensor(np.array(image_rgb))

# Загрузить предобученную модель
model = models.resnet18(pretrained=True)

# Выбрать нужный нам слой модели для вывода
layer = model._modules.get('avgpool')

# Перевести модель в режим оценки (evaluation mode)
model.eval()

# Обернуть эмбеддинг в менеджер контекста with no_grad
with torch.no_grad():
    embedding = model(pytorch_image.unsqueeze(0))

print(embedding.shape)
torch.Size([1, 1000])
```

Обсуждение

В сфере машинного обучения *трансферное обучение* (transfer learning) обычно означает перенос информации, полученной при выполнении одного задания, на другое. Для того чтобы не начинать работу с нуля, мы можем воспользоваться образами,

полученными из большого количества предобученных моделей для классификации изображений (например, ResNet), тем самым значительно обогатив наши собственные модели. В качестве наглядного примера будет весьма удачным решением использовать веса ранее обученной модели распознавания кошек для обучения модели распознавания собак. Перенос информации из одной модели в другую позволяет избежать необходимости заново обучать новую модель и вместо этого воспользоваться накопленной ранее информацией.

Мы не будем подробно рассматривать внедрение трансферного обучения в компьютерное зрение; достаточно знать, что существует множество способов извлечения образов, полученных на основе эмбеддинга, помимо PyTorch. В другой популярной библиотеке глубокого обучения TensorFlow мы можем использовать `tensorflow_hub`:

```
# Загрузить библиотеку
import cv2
import tensorflow as tf
import tensorflow_hub as hub

# Загрузить изображение
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Преобразовать изображение в тип данных tensorflow
tf_image = tf.image.convert_image_dtype([image_rgb], tf.float32)

# Создать модель и при помощи модели inception V1 получить эмбеддинги
embedding_model = hub.KerasLayer(
    "https://tfhub.dev/google/imagenet/inception_v1/feature_vector/5"
)
embeddings = embedding_model(tf_image)

# Вывести на экран форму эмбеддинга
print(embeddings.shape)
(1, 1024)
```

Дополнительные материалы для чтения

- Курс PyTorch: Transfer Learning for Computer Vision ("Трансферное обучение для компьютерного зрения"; **https://oreil.ly/R8RTk**).
- TensorFlow Hub (**https://oreil.ly/iwHI6**).

8.16. Распознавание объектов при помощи OpenCV

Задача

Вам нужно найти объекты на изображении при помощи предобученных каскадных классификаторов (cascade classifiers) библиотеки OpenCV.

Решение

Загрузим и запустим один из классификаторов OpenCV на основе каскада Хаара (Haar cascade classifiers, **https://oreil.ly/XlXbm**). В данном случае мы будем использовать заранее обученную модель распознавания лиц для определения и выделения при помощи рамки вокруг лица на фотографии (рис. 8.25):

```
# Импортировать библиотеки
import cv2
from matplotlib import pyplot as plt

# при первом запуске:
# mkdir models && cd models
# wget https://tinyurl.com/mrc6jwhp
face_cascade = cv2.CascadeClassifier()
face_cascade.load(
    cv2.samples.findFile(
        "models/haarcascade_frontalface_default.xml"
    )
)

# Загрузить изображение
image_bgr = cv2.imread("images/kyle_pic.jpg", cv2.IMREAD_COLOR)
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Распознать лица и нарисовать прямоугольник
faces = face_cascade.detectMultiScale(image_rgb)
for (x,y,w,h) in faces:
    cv2.rectangle(image_rgb, (x, y),
                  (x + h, y + w),
                  (0, 255, 0), 5)

# Вывести изображение
plt.subplot(1, 1, 1)
plt.imshow(image_rgb)
plt.show()
```

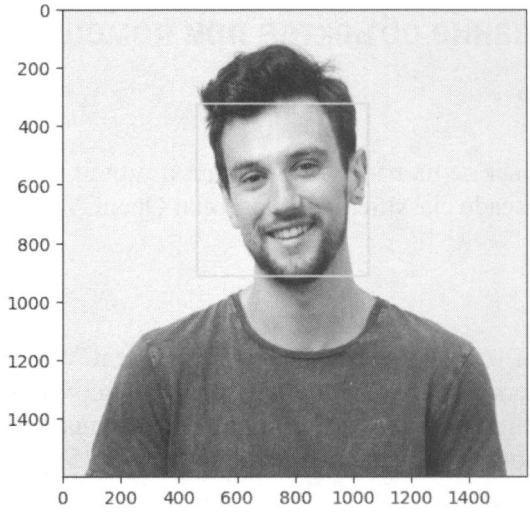

Рис. 8.25

Обсуждение

Каскадный классификатор Хаара представляет собой модели машинного обучения, которые запоминают набор признаков изображений (конкретно признаки Хаара) и затем используются для распознавания объектов на изображениях. Признаки представляют собой простые прямоугольники, которые выводятся из разницы сумм прямоугольных областей. Далее при помощи алгоритма градиентного бустинга (gradient boosting algorithm) извлекаются более важные признаки, и, наконец, при помощи каскадных классификаторов создаются более сильные модели.

Мы не будем подробно вдаваться в детали этого процесса; стоит лишь отметить, что предобученные модели можно легко получить, например, из репозитория OpenCV на GitHub (**https://oreil.ly/273DA**) в формате файлов XML и применить их для обработки нужных изображений без создания собственной модели. Это полезно при необходимости добавления простых двоичных признаков изображения, таких как `contains_face` (или любых других объектов), к вашим данным.

Дополнительные материалы для чтения

- Cascade Classifier ("Каскадный классификатор", курс OpenCV; **https://oreil.ly/dFhu6**).

8.17. Классификация изображений при помощи Pytorch

Задача

Вы хотите классифицировать изображения с использованием заранее обученных моделей библиотеки Pytorch.

Решение

Воспользуемся `torchvision.models` для выбора модели предобученного классификатора изображений и обработки нашего изображения:

```
# Загрузить библиотеки
import cv2
import json
import numpy as np
import torch
from torchvision import transforms
from torchvision.models import resnet18
import urllib.request

# Загрузить классы imagenet
with urllib.request.urlopen(
    "https://raw.githubusercontent.com/raghakot/keras-vis/master/resources/"
    ):
    imagenet_class_index = json.load(url)

# Инициализировать предобученную модель
model = resnet18(pretrained=True)

# Загрузить изображение
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Преобразовать изображение в тип данных pytorch
convert_tensor = transforms.ToTensor()
pytorch_image = convert_tensor(np.array(image_rgb))

# Перевести модель в режим оценки
model.eval()

# Создать прогноз
prediction = model(pytorch_image.unsqueeze(0))

# Получить индекс наиболее вероятного прогноза
_, index = torch.max(prediction, 1)

# Преобразовать результат в процентное значение
percentage = torch.nn.functional.softmax(prediction, dim=1)[0] * 100
```

```
# Вывести на экран название по индексу элемента с процентом вероятности
print(imagenet_class_index[str(index.tolist()[0])][1],
    percentage[index.tolist()[0]].item())
airship 6.0569939613342285
```

Обсуждение

Многие предобученные модели глубокого обучения для классификации изображений доступны через библиотеки PyTorch и TensorFlow. В данном примере мы используем ResNet18 — 18-слойную архитектуру глубокой нейронной сети (deep neural network), обученную на наборе данных ImageNet. В Pytorch также доступны более глубокие модели вроде ResNet101 и ResNet152, а также много других моделей для обработки изображений. При использовании набора ImageNet обученные модели могут предсказывать вероятности для всех классов, указанных в переменной `imagenet_class_index` из сниппета, загруженного ранее с GitHub.

Как и в случае распознавания лиц в OpenCV (*см. рецепт 8.16*), мы можем использовать предсказание классов объектов изображений в качестве признаков будущих моделей машинного обучения или полезных тегов метаданных изображений.

Дополнительные материалы для чтения

- Models and Pre-trained Weights ("Модели и предобученные веса", документация PyTorch; **https://oreil.ly/MhlxR**).

ГЛАВА 9

Снижение размерности через выделение признаков

Введение

При создании моделей машинного обучения мы, как правило, имеем дело с тысячами и сотнями тысяч признаков. Например, в *главе 8* мы преобразовали цветное изображение 256×256 в 196 608 признаков. С учетом того что каждый пиксел может принимать одно из 256 значений, полученные наблюдения могут иметь 256^{196608} различных конфигураций. Для многих алгоритмов машинного обучения работа с такими данными представляет определенную трудность ввиду чрезмерных затрат на сбор достаточного количества наблюдений. Это относится, в том числе, к табличным структурированным наборам данных, т. к. после их обработки мы снова получаем тысячи признаков.

К счастью, не все признаки равнозначны, поэтому при помощи *выделения признаков* (feature extraction) для снижения размерности (dimensionality reduction) можно преобразовать набор признаков $p_{\text{исх}}$ в $p_{\text{новый}}$, получив $p_{\text{исх}} > p_{\text{новый}}$ с сохранением большей части хранящейся внутри информации. Другими словами, мы сокращаем количество признаков без существенного влияния на возможность создания высококачественных предсказаний на основе наших данных. Далее в этой главе мы рассмотрим различные методы выделения признаков.

Недостатком методов выделения признаков является то, что получаемые признаки непригодны для понимания человеком. Они все так же хорошо подходят для обучения наших моделей, однако человек увидит лишь набор случайных чисел. Если же мы хотим сохранить понятность моделей для человека, нам лучше подойдет снижение размерности через *отбор признаков* (feature selection), описанный в *главе 10*. При отборе признаков мы удаляем несущественные на наш взгляд признаки и не меняем оставшиеся. В отличие от выделения признаков, мы не сможем сохранить всю информацию, однако оставшиеся признаки остаются нетронутыми, и мы сможем изучать их в процессе анализа.

9.1. Сокращение признаков при помощи главных компонент

Задача

Вы хотите сократить количество признаков с сохранением дисперсии (важной информации) данных.

Решение

Используем метод анализа главных компонент (PCA) при помощи функции PCA библиотеки scikit-learn:

```
# Загрузить библиотеку
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets

# Загрузить данные
digits = datasets.load_digits()

# Стандартизировать матрицу признаков
features = StandardScaler().fit_transform(digits.data)

# Создать PCA для сохранения 99% дисперсии
pca = PCA(n_components=0.99, whiten=True)

# Запустить PCA
features_pca = pca.fit_transform(features)

# Вывести результаты
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_pca.shape[1])
```

```
Original number of features: 64
Reduced number of features: 54
```

Обсуждение

Анализ главных компонент (PCA) является распространенным способом линейного уменьшения размерности данных. При его использовании происходит проецирова-

ние наблюдений на (желательно небольшой) набор главных компонентов матрицы признаков с наибольшей *дисперсией* данных, что на практике означает сохранение информации. Поскольку данный метод является неконтролируемым, в нем не используется информация целевого вектора, а упор делается на матрицу признаков.

Ссылки на математическое описание принципа работы PCA можно найти в конце этого раздела. Попробуем на простом примере понять, как работает PCA с точки зрения интуиции. На рис. 9.1 показаны данные с двумя признаками: x_1 и x_2. Как мы можем видеть, наблюдения вытягиваются в форме сигары — больше в ширину и меньше в высоту. Можно сказать, что дисперсия "ширины" значительно меньше "высоты". Вместо понятий "ширины" и "высоты" для первой главной компоненты используем термин "направление" с наибольшей дисперсией, а для второй главной компоненты — "направление" со второй по величине дисперсией (и т. д.).

Одна из стратегий сокращения числа признаков заключается в проецировании всех наблюдений двумерного пространства на одномерный вектор главной компоненты. Мы потеряем информацию второй главной компоненты, однако это может быть допустимым компромиссом в некоторых ситуациях. Так выглядит метод PCA.

Рис. 9.1. Главная и вторая по величине компоненты PCA

В библиотеке scikit-learn PCA представлен классом `PCA`. При помощи параметра `n_components` можно запустить одну из двух операций. Если значение аргумента больше 1, алгоритм `pca` вернет все признаки. В таком случае необходимо понять, как выбрать оптимальное количество признаков. С другой стороны, при значении `n_components` от 0 до 1 функция `pca` вернет минимальное количество признаков с наибольшей дисперсией. Обычно используются значения `0.95` и `0.99`, что соответствует сохранению 95% и 99% дисперсии изначальных признаков. Параметр `whiten=True` масштабирует каждую главную компоненту до нулевого среднего значения и единичной дисперсии. Другой параметр, `svd_solver="randomized"`, позволяет

применить стохастический алгоритм для зачастую намного более быстрого поиска первой главной компоненты.

В результате при помощи PCA у нас получилось уменьшить размерность на 10 признаков, сохранив 99% информации (дисперсии) матрицы признаков.

Дополнительные материалы для чтения

- PCA ("Метод главных компонент", документация scikit-learn; **https://oreil.ly/OT_gN**).
- Principal Component Analysis with Linear Algebra, Jeff Jauregui ("Метод главных компонент в линейной алгебре"; **https://oreil.ly/Uns61**).

9.2. Сокращение признаков у линейно неразделимых данных

Задача

Вы полагаете, что работаете с линейно неразделимыми данными, при этом хотите уменьшить размерность.

Решение

Применим расширенный метод главных компонент с использованием ядер для уменьшения нелинейной размерности:

```
# Загрузить библиотеку
from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles

# Создать линейную неразделимость
data features, _ = make_circles(n_samples=1000, random_state=1, noise=0.1, factor=0.1)

# Применим ядро PCA и ядро с радиальной базисной функцией (radius basis function, RBF)
kpca = KernelPCA(kernel="rbf", gamma=15, n_components=1)
features_kpca = kpca.fit_transform(features)

print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kpca.shape[1])
```

```
Original number of features: 2
Reduced number of features: 1
```

Обсуждение

Метод PCA позволяет уменьшить размерность матрицы признаков (т. е. количество признаков). В стандартном методе главных компонент для сокращения числа признаков используется линейная проекция (linear projection). Если данные *линейно разделимы* (т. е. мы можем прочертить прямую или гиперплоскость между разными классами), значит, PCA сработает корректно. Однако если данные линейно неразделимы (т. е. мы можем разделить классы только при помощи кривой границы решения), линейное преобразование также не сработает. В нашем примере мы использовали функцию `make_circles` библиотеки scikit-learn для создания искусственного набора данных с целевым вектором из двух классов и двух признаков. Функция `make_circles` выводит линейно неразделимые данные; один класс находится внутри другого класса (рис. 9.2).

Рис. 9.2. Проекция первой главной компоненты на линейно неразделимые данные

При использовании линейного метода главных компонент для уменьшения размерности данных два класса линейно проецируются на первую главную компоненту, что приводит к их сплетению (рис. 9.3).

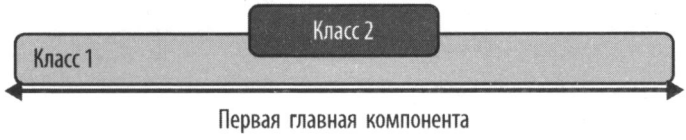

Рис. 9.3. Первая главная компонента линейно неразделимых данных без ядра PCA

Мы должны стремиться преобразовать данные таким образом, чтобы с уменьшением размерности добиться линейной разделимости. Как показано на рис. 9.4, это возможно при помощи ядра PCA.

Рис. 9.4. Первая главная компонента линейно неразделимых данных с ядром PCA

При помощи ядер мы можем осуществить проекцию линейно неразделимых данных на большее пространство, где они становятся линейно разделимыми; это называется *ядерным трюком* (kernel trick). Не стоит беспокоиться, если вы не понимаете принципов данного подхода; просто имейте в виду, что ядра — это еще один способ проецирования данных. В классе `kernelPCA` библиотеки scikit-learn представлено множество ядер, которые мы можем указать при помощи параметра `kernel`. Обычно в качестве радиальной базисной функции используется гауссово ядро (`rbf`), однако мы также можем использовать полиномиальное (polynomial) ядро (`poly`) и сигмоидальное ядро (`sigmoid`). Кроме того, мы можем указать линейную проекцию (`linear`), что дает такой же результат, как и стандартный PCA.

Недостатком ядра PCA является необходимость указывать количество параметров. К примеру, в *рецепте 9.1* мы указывали `n_components`, равный `0.99`, чтобы при помощи PCA сохранить 99% дисперсии компонентов. Эта опция отсутствует в ядре PCA. Вместо нее нам нужно задать количество компонентов (например, `n_components=1`). Более того, в ядрах есть другие гиперпараметры, которые мы должны настраивать; например, для радиальной базисной функции необходимо указать значение `gamma`.

Как же узнать, какое значение необходимо использовать? Методом проб и ошибок. В частности, мы можем обучить нашу модель несколько раз с разными ядрами или значениями параметров. Таким образом, мы сможем подобрать наиболее эффективную комбинацию и получать с ее помощью качественные прогнозы. Мы более подробно разберем эту популярную в машинном обучении стратегию в *главе 12*.

Дополнительные материалы для чтения

- Документация scikit-learn по ядрам PCA (**https://oreil.ly/SCAX-**).
- Kernel Tricks and Nonlinear Dimensionality Reduction via RBF Kernel PCA ("Ядерные трюки и уменьшение нелинейной размерности при помощи ядра RBF PCA"; **https://oreil.ly/ktm5Z**).

9.3. Сокращение признаков при помощи повышения разделимости классов

Задача

Вы хотите сократить количество используемых классификатором признаков путем повышения разделимости между классами.

Решение

Воспользуемся *линейным дискриминантным анализом* (linear discriminant analysis, LDA) для проецирования признаков на оси компонент для увеличения разделимости классов:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Загрузить набор данных с ирисами
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать и запустить LDA, с его помощью преобразовать признаки
lda = LinearDiscriminantAnalysis(n_components=1)
features_lda = lda.fit(features, target).transform(features)

# Вывести количество признаков
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_lda.shape[1])
```

```
Original number of features: 4
Reduced number of features: 1
```

При помощи функции `explained_variance_ratio_` мы можем узнать долю дисперсии, которую объясняет каждая компонента. В нашем решении одна компонента объясняет более 99% дисперсии:

```
lda.explained_variance_ratio_
```

```
array([0.9912126])
```

Обсуждение

Линейный дискриминантный анализ (LDA) является классификацией, которая, в том числе, может использоваться для уменьшения размерности. Принцип работы LDA схож с PCA, который заключается в проецировании пространства признаков на малоразмерное пространство. Однако в PCA нас интересовали только оси компонент, при помощи которых увеличивается дисперсия данных, тогда как в LDA нам также нужно увеличить различия между классами. На рис. 9.5 представлены данные, содержащие два целевых класса и два признака. Если мы спроецируем данные на ось y, то получим два сложно разделимых класса (т. е. с частичным наложением (overlap)); тем временем при проекции данных на ось x мы получим вектор признаков (т. е. мы уменьшим размерность на 1) с сохранением разделимости

классов. В реальности отношение между классами будет сложнее, а размерность — выше, однако суть концепции не меняется.

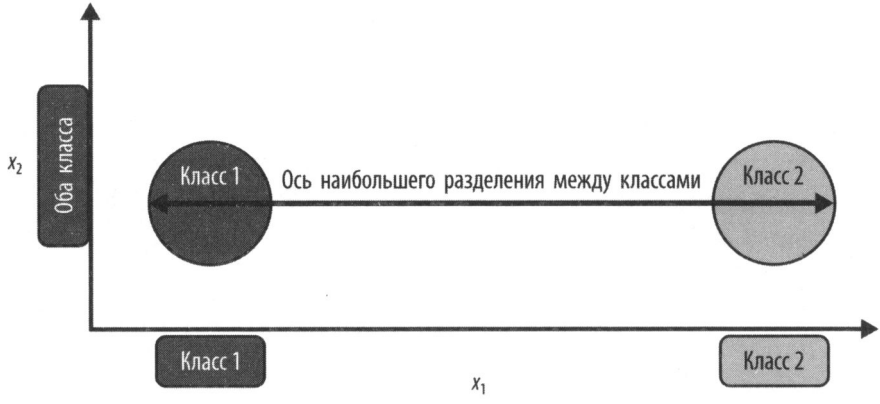

Рис. 9.5. Увеличения различия между классами при помощи LDA

Для использования LDA в библиотеке scikit-learn есть класс `LinearDiscriminantAnalysis`, содержащий параметр `n_components` для указания количества признаков, которое необходимо сохранить. Для того чтобы понять, какой аргумент нужно добавить в `n_components` (например, сколько параметров сохранить), мы можем воспользоваться данными `explained_variance_ratio_` о дисперсии каждого выводимого признака, представленными в виде сортированного массива. Например:

```
lda.explained_variance_ratio_
```

```
array([0.9912126])
```

В частности, мы можем запустить класс `LinearDiscriminantAnalysis` со значением `n_components`, равным `None`, для получения данных коэффициента дисперсии, объясняемой каждым признаком компоненты, а затем посчитать количество компонент, необходимых для преодоления порога объясненной дисперсии (обычно `0.95` или `0.99`):

```
# Создать и запустить LDA
lda = LinearDiscriminantAnalysis(n_components=None)
features_lda = lda.fit(features, target)

# Создать массив коэффициентов объясненной дисперсии
lda_var_ratios = lda.explained_variance_ratio_

# Создать функцию
def select_n_components(var_ratio, goal_var: float) -> int:
    # Указать изначальную объясненную дисперсию
    total_variance = 0.0

    # Указать изначальное количество признаков
    n_components = 0
```

```
    # Для объясненной дисперсии каждого признака
    for explained_variance in var_ratio:

        # Прибавить объясненную дисперсию к общему
        total_variance += explained_variance

        # Добавить одну компоненту
        n_components += 1

        # Если достигнут целевой уровень объяснения дисперсии
        if total_variance >= goal_var:
            # Прервать цикл
            break

    # Вывести количество компонент
    return n_components

# Запустить функцию
select_n_components(lda_var_ratios, 0.95)
```

Дополнительные материалы для чтения

- Comparison of LDA and PCA 2D Projection of Iris Dataset ("Сравнение 2D-проекции LDA и PCA на примере базы данных ирисов"; **https://oreil.ly/evGsx**).
- Linear Discriminant Analysis ("Линейный дискриминантный анализ"; **https://oreil.ly/uOB81**).

9.4. Сокращение признаков при помощи разложения матрицы

Задача

У вас есть матрица признаков с неотрицательными значениями, и вы хотите сократить размерность.

Решение

Воспользуемся *неотрицательным матричным разложением* (nonnegative matrix factorization, NMF) для уменьшения размерности матрицы признаков:

```
# Загрузить библиотеку
from sklearn.decomposition import NMF
from sklearn import datasets
```

```
# Загрузить данные
digits = datasets.load_digits()

# Загрузить матрицу признаков
features = digits.data

# Создать, настроить и применить NMF
nmf = NMF(n_components=10, random_state=4)
features_nmf = nmf.fit_transform(features)

# Вывести результаты
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_nmf.shape[1])
```

```
Original number of features: 64
Reduced number of features: 10
```

Обсуждение

Неотрицательное матричное разложение (NMF) является неконтролируемым методом уменьшения линейной размерности, в основе которого лежит разложение (т. е. разбитие на множество матриц, произведение которых примерно соответствуют изначальной матрице) матрицы признаков на несколько матриц, показывающих скрытую связь между наблюдениями и их признаками. Интуитивно NMF позволяет снизить размерность, поскольку при разложении два множителя (умножаемые матрицы) могут иметь значительно меньшую размерность, чем матрица произведений. Формально, если мы зададим нужное число признаков r, то получим следующую формулу NMF:

$$\mathbf{V} \approx \mathbf{WH},$$

где $\mathbf{V}$ — матрица признаков размером $n \times d$ (т. е. d признаков, n наблюдений); $\mathbf{W}$ — матрица размером $n \times r$; $\mathbf{H}$ — матрица размером $r \times d$. Изменяя значение r, мы можем настроить необходимую величину снижения размерности.

Главным условием использования неотрицательного матричного разложения (NMF), как следует из названия, является отсутствие отрицательных значений в матрице признаков. Кроме того, в отличие от PCA и других рассмотренных нами методов, NMF не возвращает данные объясненной дисперсии выходных признаков. Таким образом, для поиска оптимального значения `n_components` нам следует перебрать значения и определить наиболее подходящее для нашей готовой модели (*см. главу 12*).

Дополнительные материалы для чтения

- Non-negative matrix factorization ("Неотрицательное матричное разложение", Википедия; **https://oreil.ly/HJ_Qe**).

9.5. Сокращение признаков разреженных данных

Задача

Необходимо уменьшить размерность имеющейся разреженной матрицы признаков.

Решение

Используем *усеченное сингулярное разложение* (truncated singular value decomposition, TSVD):

```
# Загрузить библиотеки
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import TruncatedSVD
from scipy.sparse import csr_matrix
from sklearn import datasets
import numpy as np

# Загрузить данные
digits = datasets.load_digits()

# Стандартизировать матрицу признаков
features = StandardScaler().fit_transform(digits.data)

# Создать разреженную матрицу
features_sparse = csr_matrix(features)

# Создать усеченное SVD
tsvd = TruncatedSVD(n_components=10)

# Применить усеченное SVD к разреженной матрице
features_sparse_tsvd = tsvd.fit(features_sparse).transform(features_sparse)

# Вывести результаты
print("Original number of features:", features_sparse.shape[1])
print("Reduced number of features:", features_sparse_tsvd.shape[1])
```

```
Original number of features: 64
Reduced number of features: 10
```

Обсуждение

Усеченное сингулярное разложение (усеченное SVD, TSVD) похоже на анализ главных компонент (PCA), т. к. в PCA на одном из этапов часто используется неусеченное *сингулярное разложение* (nontruncated singular value decomposition). Имея набор d признаков, в результате сингулярного разложения (SVD) мы получим матрицы факторных коэффициентов размером $d \times d$, тогда как усеченное сингулярное разложение вернет факторы размером $n \times n$, где параметр n был указан заранее. На практике преимущество усеченной модели SVD перед PCA заключается в том, что она может работать с разреженными матрицами признаков.

У усеченного SVD также есть один недостаток: использование генератора случайных чисел приводит к перестановке знаков в результатах преобразования, создавая, другими словами, "неопределенность знаков". Для решения этой проблемы следует использовать метод `fit` один раз для каждого конвейера, а затем свободно использовать метод `transform`.

Как и в случае линейного дискриминантного анализа, нам необходимо указать количество признаков (компонентов), которое мы хотим получить. Для этого используем параметр `n_components`. Остается неясным, какое количество компонентов следует считать оптимальным. В качестве одного из решений можно представить `n_components` в качестве гиперпараметра и оптимизировать его в процессе выбора модели (т. е. выбрать такое значение `n_components`, при котором получается наилучшая модель). С другой стороны, поскольку результат усеченного SVD пропорционален дисперсии изначальной матрицы признаков, объясненной каждым компонентом, мы можем выбрать множество компонентов, необходимое для объяснения размера дисперсии (обычно это 95% и 99%). В нашем примере три первых компонента вывода объясняют приблизительно 30% изначальной дисперсии данных:

```
# Суммировать три первых компонента, объясняющих коэффициент размерности
tsvd.explained_variance_ratio_[0:3].sum()
```

```
0.3003938537287226
```

Мы можем автоматизировать этот процесс, создав функцию для запуска усеченного SVD со значением `n_components` на один меньше изначального количества признаков, а затем рассчитать число компонентов для объяснения необходимой величины от изначальной дисперсии:

```
# Создать и запустить усеченное SVD со значением на 1 меньше количества признаков
tsvd = TruncatedSVD(n_components=features_sparse.shape[1]-1)
features_tsvd = tsvd.fit(features)

# Проверить долю информации от общей дисперсии
tsvd_var_ratios = tsvd.explained_variance_ratio_

# Создать функцию
def select_n_components(var_ratio, goal_var):
```

```
    # Указать исходную объясненную дисперсию
    total_variance = 0.0

    # Указать исходное количество признаков
    n_components = 0

    # Для объясненной дисперсии каждого признака
    for explained_variance in var_ratio:

        # Прибавить объясненную дисперсию к общему
        total_variance += explained_variance

        # Добавить одну компоненту
        n_components += 1

        # Если достигнут целевой уровень объяснения дисперсии
        if total_variance >= goal_var:

            # Прервать цикл
            break

    # Вывести количество компонент
    return n_components

# Запустить функцию
select_n_components(tsvd_var_ratios, 0.95)
40
```

Дополнительные материалы для чтения

- Truncated ("Усеченное SVD", документация scikit-learn; **https://oreil.ly/nD1pF**).

ГЛАВА 10

Снижение размерности через отбор признаков

Введение

В *главе 9* мы узнали, как можно снизить размерность матрицы признаков путем создания новых функций, которые (в идеальном случае) сохраняют потенциал оригинальных данных для тренировки эффективных моделей при значительно меньшей размерности. Это называется *выделением признаков* (feature extraction). В этой главе мы разберем другой подход: отбор наиболее качественных и информативных признаков и отказ от менее важных. Это называется *отбором признаков* (feature selection).

Существует три категории методов отбора признаков: фильтры (filter), обертки (wrapper) и встроенные методы (embedded). *Фильтрующие методы* (filter methods) применяются для выбора наилучших признаков путем изучения их статистических свойств. Примером такого отбора могут быть методы, в которых мы явно задаем пороги для статистических данных или производим отбор необходимых признаков вручную. В *методах обертывания* (wrapper methods) поиск подмножества признаков, позволяющих создавать наиболее эффективные модели предсказаний, происходит через апробацию и отсеивание ошибок. Эти методы зачастую показывают наибольшую эффективность, поскольку наилучший результат получается в результате реальных экспериментов, а не наивных допущений. Наконец, во *встроенных методах*, или *методах вложений* (embedded methods), отбор подмножества наилучших признаков является частью или продолжением процесса обучения алгоритмов.

В идеальном случае нам следовало бы рассмотреть все три категории методов в одной главе. Однако ввиду того, что встроенные методы привязаны к конкретным алгоритмам обучения, нам будет сложно понять их работу в отрыве от более детального изучения непосредственно алгоритмов. Поэтому в данной главе мы рассмотрим только методы отбора признаков путем фильтрации и обертывания, а конкретные методы вложений будем рассматривать в последующих главах, посвященных углубленному изучению алгоритмов обучения.

10.1. Пороговый отбор числовых признаков дисперсии

Задача

Имеется множество числовых признаков, из которого нужно удалить значения с низкой дисперсией (т. е. вероятно содержащие малое количество информации).

Решение

Выберем набор признаков с дисперсией выше пороговых значений:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.feature_selection import VarianceThreshold

# Загрузить некоторые данные
iris = datasets.load_iris()

# Создать набор признаков и целевой набор
features = iris.data
target = iris.target

# Задать пороговые значения
thresholder = VarianceThreshold(threshold=.5)

# Создать матрицу признаков с высокой дисперсией
features_high_variance = thresholder.fit_transform(features)

# Вывести матрицу признаков с высокой дисперсией
features_high_variance[0:3]
array([[ 5.1,  1.4,  0.2],
       [ 4.9,  1.4,  0.2],
       [ 4.7,  1.3,  0.2]])
```

Обсуждение

Метод пороговой обработки дисперсии (variance thresholding, VT) является примером одного из базовых способов отбора признаков, который осуществляется посредством фильтрации. В его основе лежит идея о том, что признаки с низкой дисперсией, вероятно, менее информативны (и полезны) признаков с высокой дисперсией.

При использовании метода VT первым делом вычисляется дисперсия каждого признака:

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \mu\right)^2,$$

где $\mathbf{x}$ — вектор признаков; x_i — значение отдельного признака; μ — среднее значение признака. Затем происходит удаление всех признаков со значениями ниже пороговых. При использовании VT важно помнить две вещи. Во-первых, дисперсия не центрируется; другими словами, она измеряется в квадратах единиц измерения самого признака. Из этого следует, что метод VT не будет работать с набором признаков, если они содержат разные единицы измерения (например, один признак выражает год, а другой — сумму в долларах). Во-вторых, порог дисперсии устанавливается вручную, соответственно, пользователь сам определяет пригодность значения (или можно воспользоваться методом выбора модели, который описывается в *главе 12*). При помощи атрибута `variances_` мы можем увидеть значение дисперсии каждого признака:

```
# Вывести значения дисперсии
thresholder.fit(features).variances_
array([0.68112222, 0.18871289, 3.09550267, 0.57713289])
```

Наконец, если признаки были стандартизованы (до среднего нулевого значения и единицы измерения дисперсии), метод пороговой обработки дисперсии по очевидным причинам будет работать неправильно:

```
# Загрузить библиотеку
from sklearn.preprocessing import StandardScaler

# Стандартизировать матрицу признаков
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Вычислить дисперсию для каждого признака
selector = VarianceThreshold()
selector.fit(features_std).variances_
array([1., 1., 1., 1.])
```

10.2. Пороговый отбор дисперсии бинарных признаков

Задача

Имеется множество бинарных категориальных признаков, из которого нужно удалить значения с низкой дисперсией (т. е. вероятно содержащие малое количество информации).

Решение

Выберем набор признаков с дисперсией случайной переменной, распределенной по закону Бернулли, выше пороговых значений:

```
# Загрузить библиотеку
from sklearn.feature_selection import VarianceThreshold

# Создать матрицу признаков, содержащую:
# признак 0: 80% класс 0
# признак 1: 80% класс 1
# признак 2: 60% класс 0, 40% класс 1
features = [[0, 1, 0],
            [0, 1, 1],
            [0, 1, 0],
            [0, 1, 1],
            [1, 0, 0]]

# Осуществить пороговый отбор по дисперсии
thresholder = VarianceThreshold(threshold=(.75 * (1 - .75)))
thresholder.fit_transform(features)
array([[0],
       [1],
       [0],
       [1],
       [0]])
```

Обсуждение

Одной из стратегий отбора наиболее информативных и удаления менее информативных категориальных признаков, как и в случае числовых данных, является оценка показателей их дисперсии. При работе с бинарными признаками (случайными переменными распределения Бернулли) дисперсия вычисляется следующим образом:

$$\mathrm{Var}(x) = p(1 - p),$$

где p — доля наблюдений класса 1. Таким образом, задав значение p, мы можем избавиться от признаков, в которых большое количество наблюдений принадлежит одному классу.

10.3. Обработка признаков с высоким коэффициентом корреляции

Задача

У вас есть матрица признаков с предположительно высокой корреляцией некоторых признаков.

Решение

Проверим коэффициенты корреляции признаков при помощи матрицы корреляций. При наличии признаков с высокой корреляцией удалим один из них:

```
# Загрузить библиотеку
import pandas as pd
import numpy as np

# Создать матрицу с высокой корреляцией двух признаков
features = np.array([[1, 1, 1],
                     [2, 2, 0],
                     [3, 3, 1],
                     [4, 4, 0],
                     [5, 5, 1],
                     [6, 6, 0],
                     [7, 7, 1],
                     [8, 7, 0],
                     [9, 7, 1]])

# Преобразовать матрицу в DataFrame
dataframe = pd.DataFrame(features)

# Создать матрицу корреляций
corr_matrix = dataframe.corr().abs()

# Выбрать верхний треугольник матрицы корреляций
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
                          k=1).astype(bool))

# Определить индекс столбцов признаков со значением корреляции выше 0,95
to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]
```

```
# Удалить признаки
dataframe.drop(dataframe.columns[to_drop], axis=1).head(3)
   0  2
0  1  1
1  2  0
2  3  1
```

Обсуждение

Высокая степень корреляции признаков является одной из основных проблем машинного обучения. Два коррелирующих признака содержат очень схожую информацию, а значит, будет излишне использовать их вместе. При использовании простых моделей, например линейной регрессии, отказ от удаления таких признаков влияет на допущения линейной регрессии и может привести к раздутому значению коэффициента детерминации (R-squared). Решение весьма простое — удалить один коррелирующий признак из множества. Удаление таких признаков при помощи пороговых значений корреляции является примером фильтрации.

В нашем решении мы сначала создаем матрицу корреляций всех признаков:

```
# Матрица корреляций
dataframe.corr()
          0         1         2
0  1.000000  0.976103  0.000000
1  0.976103  1.000000 -0.034503
2  0.000000 -0.034503  1.000000
```

Затем мы используем верхний треугольник матрицы корреляций для определения пар признаков с высокой корреляцией:

```
# Верхний угол матрицы корреляций
upper
    0         1         2
0 NaN  0.976103  0.000000
1 NaN       NaN  0.034503
2 NaN       NaN       NaN
```

Наконец, мы удаляем один признак из каждой пары.

10.4. Удаление нерелевантных для классификации признаков

Задача

Имеется категориальный целевой вектор, из которого необходимо удалить неинформативные признаки.

Решение

Если признаки категориальные, вычислим хи-квадрат (χ^2) между каждым признаком и целевым вектором:

```
# Загрузить библиотеку
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_classif

# Загрузить данные
iris = load_iris()
features = iris.data
target = iris.target

# Преобразовать данные в числовые, чтобы получить категориальные данные
features = features.astype(int)

# Выбрать два признака с наибольшими показателями хи-квадрата
chi2_selector = SelectKBest(chi2, k=2)
features_kbest = chi2_selector.fit_transform(features, target)

# Вывести результаты
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

```
Original number of features: 4
Reduced number of features: 2
```

Если признаки количественные, используем дисперсионный анализ (ANOVA, Analysis Of VAriance) для получения F-значения между каждым признаком и целевым вектором:

```
# Выбрать два признака с наибольшими F-значениями
fvalue_selector = SelectKBest(f_classif, k=2)
features_kbest = fvalue_selector.fit_transform(features, target)

# Вывести результаты
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

```
Original number of features: 4
Reduced number of features: 2
```

Для того чтобы не указывать конкретное число признаков, воспользуемся объектом класса `SelectPercentile` для выбора *n* процента самых удачных признаков:

```
# Загрузить библиотеку
from sklearn.feature_selection import SelectPercentile
```

```
# Выбрать 75% признаков с наибольшими F-значениями
fvalue_selector = SelectPercentile(f_classif, percentile=75)
features_kbest = fvalue_selector.fit_transform(features, target)

# Вывести результаты
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
Original number of features: 4
Reduced number of features: 3
```

Обсуждение

При помощи статистического критерия хи-квадрат мы можем узнать, насколько независимы два категориальных вектора. Другими словами, эти критерии показывают разницу между видимыми наблюдениями каждого класса категориальных признаков и возможными состояниями при отсутствии зависимости (т. е. связи) от целевого вектора:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i},$$

где O_i — количество видимых наблюдений в классе i; E_i — количество вероятных наблюдений в классе i.

Критерий хи-квадрат — это одно число, которое показывает величину отличия наблюдаемых вычислений от возможных вычислений при отсутствии каких-либо связей. При подсчете этого статистического критерия получаемое нами значение демонстрирует степень независимости признака от целевого вектора. Если целевой показатель не зависит от значения признака, он не представляет для нас интереса, поскольку в нем нет пригодной для классификации информации. С другой стороны, если оба признака находятся в тесной зависимости, они, весьма вероятно, содержат информацию, полезную для обучения наших моделей.

При использовании хи-квадрата для отбора признаков мы рассчитываем его величину между каждым признаком и целевым вектором, затем выбираем признаки с наиболее подходящим значением хи-квадрата. В scikit-learn выбор признаков осуществляется при помощи класса `SelectKBest`. Количество информативных признаков, которые нужно сохранить, задается параметром `k`.

Следует отметить, что критерий хи-квадрат можно рассчитать только для двух категориальных векторов. По этой причине для отбора признаков при помощи хи-квадрата целевой вектор и признаки должны быть категориальными. При наличии количественных признаков мы также можем использовать критерий хи-квадрат, однако нам понадобится сначала преобразовать их в категориальные. Наконец, для использования статистического показателя хи-квадрат все значения должны быть неотрицательными.

В качестве альтернативы, имея числовой признак, мы можем использовать функцию `f_classif` для получения F-значения дисперсионного анализа с каждым признаком и целевым вектором. При группировке числовых признаков относительно целевого вектора F-статистика определяет присутствие существенных отличий средних значений каждой группы. Например, если у нас есть бинарный целевой вектор, пол человека и количественный признак — контрольные показатели (test scores), — при помощи F-значения мы сможем определить, различаются ли контрольные показатели мужчин и женщин. Если нет, то контрольный показатель никак не поможет предугадать пол, а значит, этот признак неинформативен.

10.5. Рекурсивное исключение признаков

Задача

Вы хотите автоматизировать процесс выбора лучших признаков.

Решение

В scikit-learn мы можем осуществить *рекурсивное исключение признаков* (recursive feature elimination, RFE) посредством кросс-валидации (cross-validation, CV), воспользовавшись классом `RFECV`. Другими словами, мы используем метод обертывания для отбора признаков и запускаем несколько итераций обучения модели, каждый раз удаляя признак, пока результативность (т. е. точность) модели не начнет падать. В итоге остаются наиболее полезные признаки:

```
# Загрузить библиотеку
import warnings
from sklearn.datasets import make_regression
from sklearn.feature_selection import RFECV
from sklearn import datasets, linear_model

# Отключить надоедливое, не очень нужное предупреждение
warnings.filterwarnings(action="ignore", module="scipy",
                        message="^internal gelsd")

# Создать матрицу признаков, целевой вектор и настоящие коэффициенты
features, target = make_regression(n_samples = 10000,
                                   n_features = 100,
                                   n_informative = 2,
                                   random_state = 1)

# Создать объект линейной регрессии
ols = linear_model.LinearRegression()
```

```
# Рекурсивно исключить признаки
rfecv = RFECV(estimator=ols, step=1, scoring="neg_mean_squared_error")
rfecv.fit(features, target)
rfecv.transform(features)
array([[ 0.00850799,  0.7031277 ,  1.52821875],
       [-1.07500204,  2.56148527, -0.44567768],
       [ 1.37940721, -1.77039484, -0.74675125],
       ...,
       [-0.80331656, -1.60648007,  0.52231601],
       [ 0.39508844, -1.34564911,  0.4228057 ],
       [-0.55383035,  0.82880112,  1.73232647]])
```

В результате рекурсивного исключения признаков мы получаем набор признаков, которые нужно оставить:

```
# Вывести количество лучших признаков
rfecv.n_features_
3
```

Мы также можем просмотреть, какие именно признаки нужно оставить:

```
# Вывести категории
rfecv.support_
array([False, False, False, False, False,  True, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False])
```

Кроме того, мы можем ранжировать признаки:

```
# Ранжировать признаки от лучших (1) к худшим
rfecv.ranking_
array([11, 92, 96, 87, 46,  1, 48, 23, 16,  2, 66, 83, 33, 27, 70, 75, 29,
       84, 54, 88, 37, 42, 85, 62, 74, 50, 80, 10, 38, 59, 79, 57, 44,  8,
       82, 45, 89, 69, 94,  1, 35, 47, 39,  1, 34, 72, 19,  4, 17, 91, 90,
       24, 32, 13, 49, 26, 12, 71, 68, 40,  1, 43, 63, 28, 73, 58, 21, 67,
        1, 95, 77, 93, 22, 52, 30, 60, 81, 14, 86, 18, 15, 41,  7, 53, 65,
       51, 64,  6,  9, 20,  5, 55, 56, 25, 36, 61, 78, 31,  3, 76])
```

Обсуждение

Это, весьма вероятно, самый сложный рецепт книги из уже рассмотренных нами ранее, объединяющий различные инструменты из предыдущих глав. Тем не менее процесс достаточно прост и интуитивен, поэтому мы можем изучить его уже сейчас. Идея рекурсивного исключения признаков (RFE) заключается в повторяющемся обучении модели с обновлением *весов*, или *коэффициентов*, на каждой итерации. На первом этапе обучения мы включаем все признаки. Затем мы находим признак с наименьшими параметрами (предполагается, что признаки масштабированы или стандартизованы), т. е. менее важный, и удаляем его из множества.

Возникает очевидный вопрос: сколько признаков нам нужно сохранить? Гипотетически, мы можем повторять цикл, пока не оставим всего один признак. Более разумным подходом будет использование новой концепции — *кросс-валидации* (CV). Здесь мы обозначим основную идею, а подробный разбор оставим для следующей главы.

Предположим, что данные содержат (1) целевой показатель, который нужно предсказать, и (2) матрицу признаков. Тогда, во-первых, нам нужно разделить их на две группы: обучающий набор и тестовый набор. Во-вторых, при помощи обучающего набора данных мы тренируем нашу модель. В-третьих, не используя целевые показатели тестового набора, мы применяем к его признакам нашу модель с целью предсказать значения этого тестового набора. Наконец, мы сравниваем полученные предсказания целевых значений с настоящими для оценки модели.

Мы можем применять кросс-валидацию для определения оптимального количества признаков, которые нужно сохранить при рекурсивном исключении. В частности, мы используем кросс-валидацию для оценки модели на каждой итерации. Если удаление признака позволило улучшить модель, мы запускаем следующий цикл. Однако, если после удаления признака результат кросс-валидации негативный, мы возвращаем признак и считаем это множество оптимальным.

В scikit-learn рекурсивное исключение признаков при помощи кросс-валидации представлено классом `RFECV` с важными параметрами. Параметр `estimator` определяет тип обучаемой модели (например, линейная регрессия), `step` указывает количество или пропорцию признаков, которые будут удалены на каждой итерации, а `scoring` является показателем качества для оценки нашей модели в процессе кросс-валидации.

Дополнительные материалы для чтения

- Recursive feature elimination with cross-validation ("Рекурсивное исключение признаков при помощи кросс-валидации", документация scikit-learn; **https://oreil.ly/aV-Fz**).

ГЛАВА 11
Оценка модели

Введение

В этой главе мы ознакомимся со стратегиями оценки качества моделей, создаваемых при помощи алгоритмов машинного обучения. Может показаться странным, что мы обсуждаем процесс оценивания моделей, предварительно не научившись создавать их, однако для этого есть причина. Польза от моделей определяется качеством их прогнозов, а значит, наша цель, в сущности, заключается не просто в создании модели (это легко), а в создании качественной модели (это трудно). Поэтому, прежде чем изучать мириады алгоритмов обучения, давайте разберем, как оценивать модели на их основе.

11.1. Модели на основе кросс-валидации

Задача

Вы хотите оценить, как хорошо ваша классификационная модель справляется с обобщением непредвиденных данных.

Решение

Создадим конвейер для обработки данных, обучения модели и последующей оценки при помощи кросс-валидации:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import KFold, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

# Загрузить набор данных digits
digits = datasets.load_digits()
```

```
# Создать матрицу признаков
features = digits.data

# Создать целевой вектор
target = digits.target

# Создать объект стандартизации
standardizer = StandardScaler()

# Создать объект логистической регрессии
logit = LogisticRegression()

# Создать конвейер для стандартизации и последующего запуска логистической регрессии
pipeline = make_pipeline(standardizer, logit)

# Создать переменную k-блочной кросс-валидации
kf = KFold(n_splits=5, shuffle=True, random_state=0)

# Запустить k-блочную кросс-валидацию
cv_results = cross_val_score(pipeline,   # Конвейер
                             features,   # Матрица признаков
                             target,     # Целевой вектор
                             cv=kf,      # Показатели качества
                             scoring="accuracy",   # Функция потерь
                             n_jobs=-1)  # Использовать все ядра CPU

# Вычислить среднее значение
cv_results.mean()
0.969958217270195
```

Обсуждение

На первый взгляд, оценка моделей машинного обучения с учителем (supervised learning) может показаться тривиальной задачей: нужно обучить модель и при помощи метрик качества (точность, среднеквадратическая ошибка и т. д.) оценить ее работу. Однако такой подход в корне ошибочен. Мы не достигнем желаемого результата, если будем обучать модель на наших данных, а затем оценивать ее эффективность применительно к обработке этих данных. Наша модель должна эффективно работать не только с уже обработанными данными, но и с ранее не известными ей данными (например, новый покупатель, новое преступление, новое изображение). В свою очередь, представленный здесь метод должен помочь в оценке возможности модели создавать прогнозы на основе незнакомых данных.

Одной из стратегий является извлечение среза данных для тестирования. Это называется *валидацией* (validation), или *удерживающей проверкой* (hold-out). При валидации наблюдения (признаки и целевые значения) разделяются на два набора, которые называют *обучающим* (или тренировочным) и *тестовым*. Тестовый набор мы откладываем в сторону и не используем до определенного момента. Далее мы обучаем нашу модель на тренировочном наборе, используя признаки и целевой вектор, чтобы с помощью этого набора в дальнейшем создавать наиболее репрезентативные предсказания. Наконец, при помощи тестового набора мы симулируем "незнакомые внешние" данные для оценки производительности модели. Следует отметить, что у метода валидации есть два слабых места. Во-первых, оценка производительности модели сильно зависит от того набора признаков, который мы поместили в тестовый набор. Во-вторых, в процессе обучения и последующей оценки модели используются не все доступные данные.

Более прогрессивной стратегией, позволяющей обойти эти недостатки, является *k-блочная кросс-валидация* (k-fold cross-validation, KFCV). При использовании метода KFCV мы разделяем данные на *k подмножеств*, или *блоков* (folds). Затем модель обучается $k - 1$ раз на блоках, объединенных в одно обучающее множество, а последний блок используется в качестве тестового набора. Мы повторяем эту операцию k раз, каждый раз используя новый блок в качестве набора для тестирования. Итоговым результатом измерения является среднее значение данных производительности модели на каждой итерации.

В нашем примере мы использовали пять блоков для k-блочной кросс-валидации и вернули результат в виде переменной `cv_results`:

```
# Вывести на экран все 5 блоков
cv_results
```

```
array([0.96111111, 0.96388889, 0.98050139, 0.97214485, 0.97214485])
```

При использовании метода KFCV важно учитывать три особенности. Первое: предполагается, что каждое наблюдение было создано независимо от другого (т. е. это независимые одинаково распределенные данные (independent and identically distributed, IID)). Если данные независимы и ра́вно распределенные, при создании блоков наблюдения их можно перемешать. Для этого в scikit-learn можно задать параметр `shuffle=True`.

Второе: при оценке классификатора предпочтительнее использовать блоки, в которых распределение наблюдений приблизительно пропорционально целевым классам (*стратифицированная k-блочная проверка* — stratified k-fold). К примеру, если наш целевой вектор содержит данные пола, где 80% наблюдений — мужчины, то 80% наблюдений каждого блока должны быть данными о мужчинах, а 20% — о женщинах. Для использования стратифицированной k-блочной проверки в scikit-learn необходимо заменить класс `KFold` классом `StratifiedKFold`.

Наконец, для использования валидации или кросс-валидации нам необходимо при помощи обучающего набора предварительно настроить и применить эти преобразования к обучающему и тестовому наборам. Например, в процессе настройки при помощи метода `fit` инструмента стандартизации `standardizer` мы высчитываем

среднее значение и дисперсию только для обучающей последовательности. Далее мы применяем преобразования (используя метод transform) к обучающему и тестовому наборам:

```
# Загрузить библиотеку
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Настроить стандартизацию для обучающего набора
standardizer.fit(features_train)

# Применить к обучающему и тестовому наборам для последующего использования
# при обучении моделей
features_train_std = standardizer.transform(features_train)
features_test_std = standardizer.transform(features_test)
```

Это обусловлено тем, что мы пока что не трогаем отложенный в сторону тестовый набор. Если же мы будем обучать оба предобработанных набора на наблюдениях обеих последовательностей, то какая-то часть информации тестового набора обязательно попадет в обучающий. Это правило применимо ко всем этапам предварительной обработки, таким как отбор признаков.

В scikit-learn мы можем объединить выполнение нескольких процессов кросс-валидации при помощи пакета pipeline. Сначала нам нужно создать конвейер для предобработки данных (например, standardizer) и обучения модели (логистическая регрессия, logit):

```
# Создать конвейер
pipeline = make_pipeline(standardizer, logit)
```

Затем при помощи этого конвейера необходимо запустить k-блочную кросс-валидацию, а далее процесс автоматизируется инструментами scikit-learn:

```
# Запустить k-блочную кросс-валидацию
cv_results = cross_val_score(pipeline,    # Конвейер
                             features,    # Матрица признаков
                             target,      # Целевой вектор
                             cv=kf,       # Показатели качества
                             scoring="accuracy",    # Функция потерь
                             n_jobs=-1)   # Использовать все ядра CPU
```

У функции cross_val_score есть три параметра, которые нам следует разобрать.

cv

Определяет выбор метода кросс-валидации. Хотя k-блочная кросс-валидация является самым распространенным методом, существуют и другие, такие как пе-

рекрестная проверка с исключением одного (leave-one-out cross-validation), где значение *k* равно количеству экземпляров в наборе данных.

`scoring`

Является показателем успеха. Мы вернемся к его изучению в других примерах этой главы.

`n_jobs=-1`

При помощи `n_jobs=-1` мы сообщаем scikit-learn, что нам нужны все доступные ядра компьютера. К примеру, если нам доступны четыре ядра (например, в ноутбуках), мы можем задействовать их все для ускорения обработки операций.

Примечание. При запуске некоторых примеров из данной книги может появиться уведомление "ConvergenceWarning: lbfgs failed to converge" (Предупреждение конвергенции: ошибка сходимости при использовании алгоритма оптимизации `lbfgs`). Мы постарались этого избежать при настройке наших примеров, однако в случае появления этого предупреждения пока что игнорируйте его. Мы займемся решением подобных ошибок далее в книге, при рассмотрении конкретных типов моделей.

Дополнительные материалы для чтения

- Why Every Statistician Should Know About Cross-Validation ("Почему кросс-валидация нужна каждому статистику"; **https://oreil.ly/vrGXy**).
- Cross-Validation Gone Wrong ("Кросс-валидация зашла не туда"; **https://oreil.ly/NE-B8**).

11.2. Создание базовой регрессионной модели

Задача

Вы хотите создать простую базовую регрессионную модель, чтобы сравнить с ней другие обучаемые вами модели.

Решение

Воспользуемся классом `DummyRegressor` библиотеки scikit-learn и создадим основу в виде простой модели:

```
# Загрузить библиотеку
from sklearn.datasets import load_wine
from sklearn.dummy import DummyRegressor
from sklearn.model_selection import train_test_split

# Загрузить данные
wine = load_wine()
```

```
# Создать набор признаков
features, target = wine.data, wine.target

# Разделить на тестовый и обучающий наборы
features_train, features_test, target_train, target_test = train_test_split(
    features, target, random_state=0)

# Создать фиктивный регрессор
dummy = DummyRegressor(strategy='mean')

# "Обучить" фиктивный регрессор
dummy.fit(features_train, target_train)

# Вычислить значение R-квадрат
dummy.score(features_test, target_test)
```

-0.0480213580840978

Для сравнения обучим модель и оценим производительность:

```
# Загрузить библиотеку
from sklearn.linear_model import LinearRegression

# Обучить простую модель линейной регрессии
ols = LinearRegression()
ols.fit(features_train, target_train)

# Вычислить значение R-квадрат
ols.score(features_test, target_test)
```

0.804353263176954

Обсуждение

При помощи класса `DummyRegressor` мы можем создавать простейшие модели, которые подходят в качестве основы для сравнения с другими нашими моделями. Это может пригодиться для симуляции "наивного" процесса предсказаний, представленного в продукте или системе. Для примера представим, что продукт запрограммирован на то, что все новые пользователи независимо от их признаков потратят 100 условных единиц в первый месяц. Если мы добавим это допущение в базовую модель, то сможем точно определить пользу от применения подхода машинного обучения, сравнив показатель фиктивной модели `score` с обученной моделью.

В классе `DummyRegressor` мы можем выбрать метод создания предсказаний при помощи параметра `strategy`, который включает среднее значение и медиану обучаемо-

го множества. Более того, если в качестве значения strategy указывается constant, фиктивный регрессор сможет предсказывать константу для каждого наблюдения:

```
# Создать фиктивный регрессор, который предсказывает значение 1
clf = DummyRegressor(strategy='constant', constant=1)
clf.fit(features_train, target_train)

# Оценить результат
clf.score(features_test, target_test)
-0.06299212598425186
```

Одно небольшое замечание касательно вычисления результата score. Метод score по умолчанию возвращает коэффициент детерминации (R-квадрат, R^2):

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)^2},$$

где y_i — настоящее значение целевого наблюдения; $\hat{y}_i$ — предсказанное значение; $\bar{y}$ — основное значение целевого вектора.

Чем ближе значение R^2 к 1, тем больше дисперсия целевого вектора, объясненного признаками.

11.3. Создание базовой модели классификации

Задача

Вы хотите создать простой базовый классификатор, чтобы сравнить его с обучаемой моделью.

Решение

Воспользуемся классом DummyClassifier библиотеки scikit-learn:

```
# Загрузить библиотеку
from sklearn.datasets import load_iris
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import train_test_split

# Загрузить данные
iris = load_iris()

# Создать целевой вектор и матрицу признаков
features, target = iris.data, iris.target
```

```
# Разделить на тестовый и обучающий наборы
features_train, features_test, target_train, target_test = train_test_split(
    features, target, random_state=0)

# Создать фиктивный классификатор
dummy = DummyClassifier(strategy='uniform', random_state=1)

# "Обучить" модель
dummy.fit(features_train, target_train)

# Вычислить значение точности
dummy.score(features_test, target_test)
0.42105263157894735
```

Для того чтобы увидеть улучшения, мы можем сравнить базовый и обученный классификаторы:

```
# Загрузить библиотеку
from sklearn.ensemble import RandomForestClassifier

# Создать классификатор
classifier = RandomForestClassifier()

# Обучить модель
classifier.fit(features_train, target_train)

# Вычислить значение точности
classifier.score(features_test, target_test)
0.9736842105263158
```

Обсуждение

Одним из основных способов измерения эффективности классификатора является сравнение его показателей со случайным угадыванием. В scikit-learn это можно легко сделать при помощи класса `DummyClassifier`. Количество вариантов генерирования значений задается параметром `strategy`. Можно выделить две удобные стратегии. Первая: при помощи параметра `stratified` мы можем сделать наши предсказания пропорциональными классам целевого вектора обучаемой модели (например, если 20% наблюдений обучаемых данных означают женщин, в 20% предсказаний `DummyClassifier` будет этот класс). Вторая: если использовать параметр `uniform`, то полученные предсказания будут случайно распределены между классами. Например, если 20% означают женщин, а 80% — мужчин, то при использовании `uniform` мы получим предсказания, где 50% — мужчины и 50% — женщины.

Дополнительные материалы для чтения

- DummyClassifier (документация scikit-learn; **https://oreil.ly/bwqQU**).

11.4. Оценка предсказаний бинарного классификатора

Задача

Необходимо оценить качество имеющейся обученной модели классификации.

Решение

Воспользуемся функцией `cross_val_score` библиотеки scikit-learn для кросс-валидации. В качестве показателя производительности зададим параметр `scoring`, который может принимать одно из значений, включающих правильность (accuracy), точность (precision), полноту (recall) и F1-меру. *Правильность* (accuracy) является одной из распространенных метрик. Она представляет собой пропорцию верно предсказанных наблюдений:

$$\text{правильность} = \frac{TP + TN}{TP + TN + FP + FN}.$$

TP (true positive, истиннноположительное предсказание).

Количество истиннноположительных данных. Это наблюдения, которые относятся к *положительному* классу (наличие заболевания, совершение покупки и т. д.) и были предсказаны корректно.

TN (true negative, истинноотрицательное предсказание).

Количество истинноотрицательных данных. Это наблюдения, которые относятся к *отрицательному* классу (отсутствие заболевания, отсутствие покупки и т. д.) и были предсказаны корректно.

FP (false positive, ложноположительное предсказание).

Количество ложноположительных предсказаний, также называемых *ошибкой первого рода*. Это наблюдения, которые были отнесены к *положительному* классу, тогда как на практике относятся к *отрицательному*.

FN (false negative, ложноотрицательное предсказание).

Количество ложноотрицательных предсказаний, также называемых *ошибкой второго рода*. Это наблюдения, которые были отнесены к *отрицательному* классу, тогда как на практике относятся к *положительному*.

Указав параметр `scoring="accuracy"`, мы можем измерить правильность (accuracy) для 3-блочной (число по умолчанию) кросс-валидации:

```
# Загрузить библиотеку
from sklearn.model_selection import cross_val_score
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification

# Сгенерировать матрицу признаков и целевой вектор
X, y = make_classification(n_samples = 10000,
                           n_features = 3,
                           n_informative = 3,
                           n_redundant = 0,
                           n_classes = 2,
                           random_state = 1)

# Создать переменную логистической регрессии
logit = LogisticRegression()

# Запустить кросс-валидацию с параметром правильности
cross_val_score(logit, X, y, scoring="accuracy")
array([0.9555, 0.95  , 0.9585, 0.9555, 0.956 ])
```

Удобство использования параметра правильности (accuracy) объясняется лаконичностью его значения: это пропорция верно предсказанных наблюдений. Однако, в реальности, данные обычно имеют несбалансированные классы (например, 99,9% наблюдений относятся к классу 1 и только 0,1% — к классу 2). При наличии несбалансированных классов мы можем наблюдать парадокс, когда модель обладает высокой правильностью при низкой точности предсказаний. Допустим, мы пытаемся предсказать наличие очень редкой формы рака, которая распространена среди 0,1% населения. После обучения модели мы получили показатель правильности 95%. Однако 99,9% людей не болеют этой формой рака; другими словами, если бы мы просто создали "модель", в которой никто не болеет, показатель правильности такой модели был бы на 4,9% выше, хотя она ничего не может *предсказать*. По этой причине стоит также использовать другие метрики: точность (precision), полноту (recall) и F1-меру.

Точность (precision) является долей наблюдений, отнесенных классификатором к положительным и являющихся истиннположительными. Это своего рода измерение шума в наших предсказаниях — другими словами, с какой вероятностью наши положительные предсказания правильны. Высокоточные модели являются "пессимистичными" и генерируют положительные прогнозы только при полной уверенности. Формально точность можно выразить через формулу:

$$\text{точность} = \frac{TP}{TP + FP}.$$

```
# Запустить кросс-валидацию с параметром точности
cross_val_score(logit, X, y, scoring="precision")
array([0.95963673, 0.94820717, 0.9635996 , 0.96149949, 0.96060606])
```

Полнота (recall) — это доля наблюдений, определяемых классификатором как положительные, от всех наблюдений истинноположительного класса. Этот параметр отражает способность модели определять наблюдения положительного класса. Модели с высоким показателем полноты являются "оптимистичными" и имеют низкую планку при генерации положительных предсказаний:

$$\text{полнота} = \frac{TP}{TP + FN}.$$

```
# Запустить кросс-валидацию с параметром полноты
cross_val_score(logit, X, y, scoring="recall")
array([0.951, 0.952, 0.953, 0.949, 0.951])
```

При первом знакомстве с понятиями точности и полноты может потребоваться некоторое время на их понимание. Оба параметры — точность и полнота — проигрывают в интуитивности параметру правильности (accuracy). Мы с большой вероятностью захотим найти некоторый баланс между точностью и полнотой; для этого у нас есть F1-мера. Она является *средним гармоническим* (средняя величина при поиске пропорций):

$$\text{F1-мера} = 2 \cdot \frac{\text{точность} \cdot \text{полнота}}{\text{точность} + \text{полнота}}.$$

Этот показатель означает корректность полученных положительных предсказаний — какое количество названных положительными наблюдений являются истинноположительными:

```
# Запустить кросс-валидацию с параметром F1
cross_val_score(logit, X, y, scoring="f1")
array([0.95529884, 0.9500998 , 0.95827049, 0.95520886, 0.95577889])
```

Обсуждение

Параметр правильности (accuracy) в качестве показателя качества имеет множество полезных свойств, в частности, интуитивность. В свою очередь, при использовании более продвинутых метрик необходимо искать баланс между точностью (precision) и полнотой (recall) — другими словами, компромисс между "оптимистичностью" и "пессимистичностью" модели. Такой баланс обеспечивает F1-мера благодаря равномерному распределению двух параметров.

Если у нас уже есть истинные значения y и спрогнозированные значения y, вместо использования функции `cross_val_score` мы можем напрямую рассчитать показатели правильности и полноты:

```
# Загрузить библиотеку
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Создать обучающий и тестовый наборы
X_train, X_test, y_train, y_test = train_test_split(X, y,
```

```
                                                    test_size=0.1,
                                                    random_state=1)

# Предсказать значения для обучаемого целевого вектора
y_hat = logit.fit(X_train, y_train).predict(X_test)

# Вычислить значение правильности
accuracy_score(y_test, y_hat)
0.947
```

Дополнительные материалы для чтения

- Accuracy paradox ("Парадокс правильности", Википедия; **https://oreil.ly/vjgZ-**).

11.5. Оценка порогов бинарного классификатора

Задача

Вам нужно произвести оценку порогов бинарного классификатора и различных вероятностей.

Решение

Воспользуемся *ROC-кривой* (receiver operating characteristic, ROC) ошибок для оценки качества бинарного классификатора. В scikit-learn для расчета истинно- и ложноположительных результатов для каждого порогового значения и создания графика (рис. 11.1) можно использовать функцию `roc_curve`:

```
# Загрузить библиотеки
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.model_selection import train_test_split

# Создать матрицу признаков и целевой вектор
features, target = make_classification(n_samples=10000,
                                       n_features=10,
                                       n_classes=2,
                                       n_informative=3,
                                       random_state=3)

# Разделить на тестовый и обучающий наборы
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)
```

```
# Создать объект классификации
logit = LogisticRegression()

# Обучить модель
logit.fit(features_train, target_train)

# Получить спрогнозированные возможные значения
target_probabilities = logit.predict_proba(features_test)[:,1]

# Создать коэффициенты истинно- и ложноположительных значений
false_positive_rate, true_positive_rate, threshold = roc_curve(
    target_test,
    target_probabilities
)

# Вывести график ROC-кривой
plt.title("Кривая ROC (receiver operating characteristic)")
plt.plot(false_positive_rate, true_positive_rate)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0] , c=".7")
plt.plot([1, 1] , c=".7")
plt.ylabel("Доля истинноположительных исходов")
plt.xlabel("Доля ложноположительных исходов")
plt.show()
```

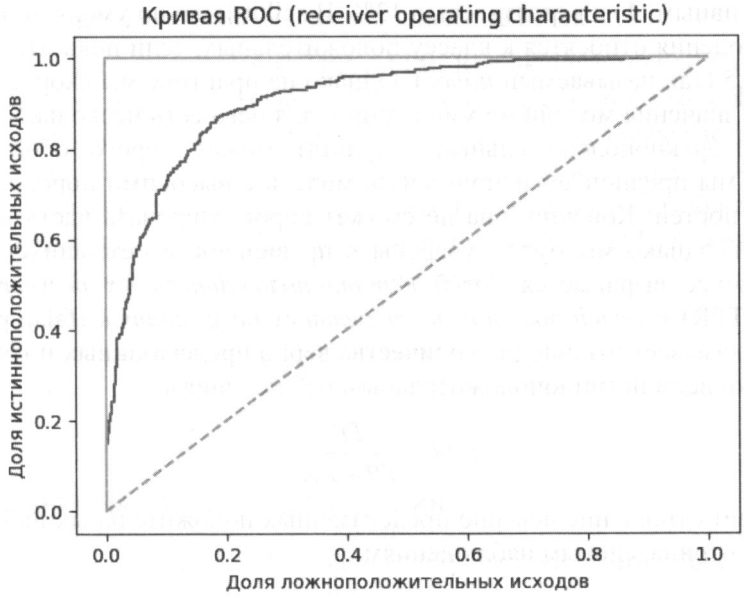

Рис. 11.1

Обсуждение

Метод ROC-кривой часто применяется для оценки качества бинарной классификации. Кривая отображает соотношение истинно- и ложноположительных значений при варьировании порога (т. е. вероятность отнесения прогнозируемого наблюдения к классу). При помощи графика мы можем наглядно оценить производительность модели. На рис. 11.1 с ROC-кривой классификатор, верно предсказывающий каждое наблюдение, изображен сплошной светло-серой линией, которая идет сразу строго вверх. Классификатор, предсказывающий случайные значения, изображен диагональной штриховой линией. Чем лучше модель, тем ближе она расположена к сплошной линии на графике.

До этого мы оценивали модели на основе прогнозируемых ими значений. Однако во многих алгоритмах эти предсказываемые значения строятся на оценках вероятностей. Другими словами, для каждого наблюдения существует явная вероятность отнесения к какому-либо классу. В нашем примере мы можем воспользоваться методом `predict_proba`, чтобы узнать прогнозируемые вероятности для первого наблюдения:

```
# Узнать прогнозируемые вероятности
logit.predict_proba(features_test)[0:1]
```

```
array([[0.86891533, 0.13108467]])
```

При помощи атрибута `classes_` мы можем просмотреть классы:

```
logit.classes_
```

```
array([0, 1])
```

Первое наблюдение в данном примере окажется негативным (0) с вероятностью ~87% и позитивным (1) с вероятностью 13%. В scikit-learn по умолчанию прогнозируемые наблюдения относятся к классу положительных, если показатель вероятности больше 0,5 (так называемый *порог*). Однако на практике мы скорее всего поменяем среднее значение модели по умолчанию, для чего есть несколько причин. Например, если ложноположительный результат может дорого обойтись нашей организации, мы предпочтем использовать модель с высокими пороговыми значениями вероятностей. Конечно, она не сможет спрогнозировать часть положительных значений, однако мы будем уверены в правильности оставшихся прогнозов. Этот компромисс выражается *долей истинноположительных результатов* (true positive rate, TPR) и *долей ложноположительных результатов* (false positive rate, FPR). TPR показывает отношение количества верно предсказанных положительных наблюдений ко всем истинноположительным наблюдениям:

$$TPR = \frac{TP}{TP + FN}.$$

FPR показывает отношение неверно предсказанных положительных наблюдений ко всем истинно отрицательным наблюдениям:

$$FPR = \frac{FP}{FP + TN}.$$

ROC-кривая отображает соотношение TPR и FPR для каждого порогового значения вероятностей. В нашем случае при величине порогового значения приблизительно 0,50 показатель TPR равен ~0,83, а FPR равен ~0,16:

```
print("Threshold:", threshold[124])
print("True Positive Rate:", true_positive_rate[124])
print("False Positive Rate:", false_positive_rate[124])
```

```
Threshold: 0.5008252732632008
True Positive Rate: 0.8346938775510204
False Positive Rate: 0.1607843137254902
```

Если же мы увеличим пороговое значение до ~80% (т. е. уровень уверенности модели при прогнозировании положительных наблюдений), значения TPR и FPR существенно снизятся:

```
print("Threshold:", threshold[49])
print("True Positive Rate:", true_positive_rate[49])
print("False Positive Rate:", false_positive_rate[49])
```

```
Threshold: 0.8058575028551827
True Positive Rate: 0.5653061224489796
False Positive Rate: 0.052941176470588235
```

В результате повышения требований к наблюдениям, которые потенциально могут быть причислены к положительному классу, модель не учитывает часть положительных наблюдений (снижение TPR) и уменьшает шум от отрицательных наблюдений, предсказанных как положительные (снижение FPR).

Помимо визуализации соотношения TPR и FPR, ROC-кривая может использоваться в качестве общей метрики для модели. Чем лучше модель, тем выше кривая и, соответственно, больше область под ней. По этой причине мы также часто вычисляем значение площади под ROC-кривой (area under the ROC curve, AUC ROC) для оценки общего качества модели со всем пороговыми значениями. Чем ближе значение площади к 1, тем лучше модель. Для вычисления площади под ROC-кривой в scikit-learn можно воспользоваться функцией `roc_auc_score`:

```
# Рассчитать площадь под ROC-кривой
roc_auc_score(target_test, target_probabilities)
```

```
0.9073389355742297
```

Дополнительные материалы для чтения

- ROC Curves in Python and R ("ROC-кривая в Python и R"; **https://oreil.ly/0qcpZ**).
- The Area Under an ROC Curve ("Площадь под ROC-кривой"; **https://oreil.ly/re7sT**).

11.6. Оценка прогнозов многоклассового классификатора

Задача

У вас есть модель, предсказывающая три класса и более, и вы хотите оценить ее результативность.

Решение

Используем кросс-валидацию со специальной метрикой оценки для работы с тремя классами и более:

```
# Загрузить библиотеку
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification

# Сгенерировать матрицу признаков и целевой вектор
features, target = make_classification(n_samples = 10000,
                                       n_features = 3,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 3,
                                       random_state = 1)

# Создать переменную логической регрессии
logit = LogisticRegression()

# Запустить кросс-валидацию с параметром правильности
cross_val_score(logit, features, target, scoring='accuracy')
array([0.841 , 0.829 , 0.8265, 0.8155, 0.82  ])
```

Обсуждение

При наличии сбалансированных классов (т. е. примерно схожего количества наблюдений для каждого класса целевого вектора), как и в случае бинарной классификации, параметр правильности (accuracy) подойдет в качестве простой и интерпретируемой метрики оценки. Правильность означает соотношение количества верных предположений к наблюдениям и хорошо подходит как для многоклассовой, так и для бинарной классификации. Если же наши классы не сбалансированы (частая ситуация), нам следует воспользоваться другими метриками оценки.

В scikit-learn достаточно встроенных метрик для оценки бинарной классификации. При этом многие из них могут быть расширены для использования в моделях, имеющих более двух классов. К ним относятся описанные ранее точность (precision), полнота (recall) и F1-мера. Хотя все они изначально создавались для работы с бинарной классификацией, мы можем применить их к многоклассовым данным, представив их в виде множества бинарных классов. В результате мы можем примерить метрики отдельно к каждому классу данных, а затем агрегировать результаты, усреднив их:

```
# Запустить кросс-валидацию с усредненным параметром F1-меры
cross_val_score(logit, features, target, scoring='f1_macro')
array([0.84061272, 0.82895312, 0.82625661, 0.81515121, 0.81992692])
```

В данном примере `macro` означает метод усреднения параметров оценки классов. Доступные опции включают `macro`, `weighted` и `micro`.

`macro` (макро).

Вычисляет усредненное значение показателей для каждого класса, взвешивая классы одинаково.

`weighted` (взвешенный).

Вычисляет усредненное значение показателей для каждого класса, взвешивая классы пропорционально их размеру относительно всех данных.

`micro` (микро).

Вычисляет усредненное значение показателей для каждой комбинации класса и наблюдений.

11.7. Визуализация показателей качества классификации

Задача

У вас есть тестовые данные из спрогнозированных и истинных классов, которые необходимо визуализировать для последующего сравнения качества моделей.

Решение

Воспользуемся *матрицей ошибок* (confusion matrix) для сравнения спрогнозированных и истинных классов:

```
# Загрузить библиотеки
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import pandas as pd

# Загрузить данные
iris = datasets.load_iris()

# Создать матрицу признаков
features = iris.data

# Создать целевой вектор
target = iris.target

# Создать список имен целевых классов
class_names = iris.target_names

# Создать обучающий и тестовый наборы
features_train, features_test, target_train, target_test = train_test_split(
    features, target, random_state=2)

# Создать переменную логистической регрессии
classifier = LogisticRegression()

# Обучить модель и создать прогнозы
target_predicted = classifier.fit(features_train,
    target_train).predict(features_test)

# Создать матрицу ошибок
matrix = confusion_matrix(target_test, target_predicted)

# Создать структуру DataFrame библиотеки pandas
dataframe = pd.DataFrame(matrix, index=class_names, columns=class_names)

# Создать тепловую карту
sns.heatmap(dataframe, annot=True, cbar=None, cmap="Blues")
plt.title("Матрица ошибок")
plt.tight_layout()
plt.ylabel("Истинный класс")
plt.xlabel("Предсказанный класс")
plt.show()
```

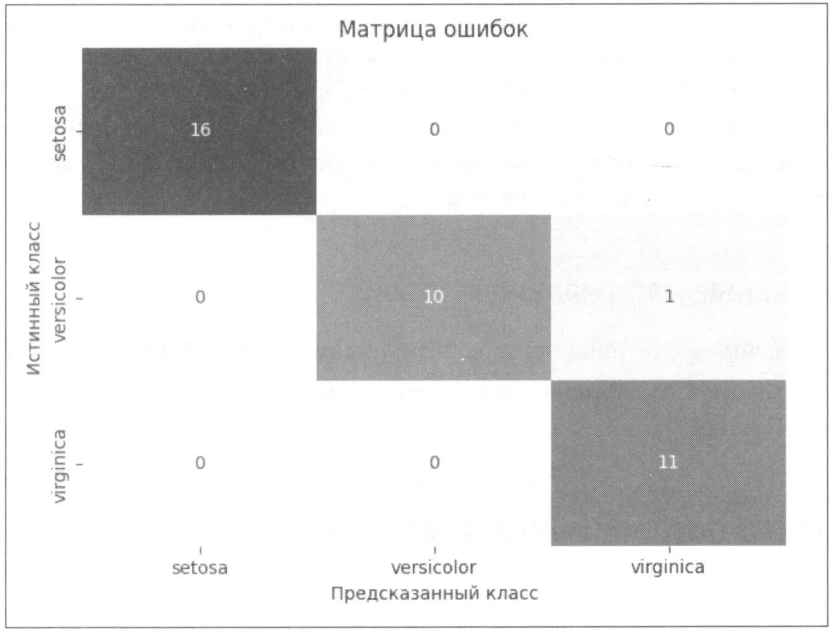

Рис. 11.2. Тепловая карта

Обсуждение

При помощи матриц ошибок можно легко и эффективно визуализировать показатели качества классификатора. Одним из главных преимуществ использования матриц ошибок является их интерпретируемость. Каждый столбец матрицы (часто изображаемой в виде тепловой карты) представляет прогнозируемый класс, а каждая строка — истинный класс. В результате элемент каждой клетки является одной из возможных комбинаций прогнозируемого и истинного классов. Для лучшего понимания объясним эту идею на примере. В нашем случае левая верхняя клетка означает количество спрогнозированных наблюдений, отнесенных к ирису щетинистому (определяется столбцом) и являющихся действительно ирисом щетинистым (определяется строкой). Это означает, что модель правильно прогнозирует результаты, относящиеся к ирису щетинистому (*Iris setosa*). Однако в случае с ирисом виргинским модель справляется хуже. Показатель правой нижней клетки означает, что модель успешно спрогнозировала 11 наблюдений, причислив их к ирису виргинскому (*Iris virginica*), однако (см. на 1 клетку выше) один элемент был ошибочно причислен не к тому классу, и относится к ирису разноцветному (*Iris versicolor*).

Следует отметить три момента касательно матриц ошибок. Во-первых, в идеальной модели значения будут распространяться по диагонали, а остальные клетки должны быть заполнены нулями. В плохой модели значения наблюдений будут распространены равномерно по всем клеткам. Во-вторых, благодаря матрице ошибок мы можем увидеть не только факт ошибки модели, но и ее причину. Другими словами,

мы можем обратить внимание на паттерны ошибочной классификации. Например, в нашей модели есть простая дифференциация по времени ириса виргинского и щетинистого и немного более сложная временная классификация для ириса виргинского и разноцветного. Наконец, матрицы ошибок могут работать с любым количеством классов (конечно, если в нашем целевом векторе содержится миллион классов, это затруднит понимание визуализации).

Дополнительные материалы для чтения

- Confusion matrix ("Матрица ошибок", Википедия; **https://oreil.ly/tDWPB**).
- Confusion Matrix ("Матрица ошибок", документация scikit-learn; **https://oreil.ly/fdsTg**).

11.8. Оценка регрессионных моделей

Задача

Вам нужно оценить возможности регрессионной модели.

Решение

Воспользуемся методом *среднеквадратической ошибки* (mean squared error, MSE):

```
# Загрузить библиотеку
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression

# Сгенерировать матрицу признаков и целевой вектор
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   n_informative = 3,
                                   n_targets = 1,
                                   noise = 50,
                                   coef = False,
                                   random_state = 1)

# Создать объект логической регрессии
ols = LinearRegression()

# Запустить кросс-валидацию линейной регрессии с (отрицательной) MSE
cross_val_score(ols, features, target, scoring='neg_mean_squared_error')
```

```
array([-1974.65337976, -2004.54137625, -3935.19355723, -1060.04361386,
       -1598.74104702])
```

Еще одной распространенной метрикой регрессии является коэффициент детерминации R^2:

```
# Запустить кросс-валидацию линейной регрессии с метрикой R-квадрат
cross_val_score(ols, features, target, scoring='r2')
array([0.8622399 , 0.85838075, 0.74723548, 0.91354743, 0.84469331])
```

Обсуждение

Среднеквадратическая ошибка (MSE) является наиболее распространенной метрикой оценки регрессионных моделей. Формально MSE выражается формулой:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2,$$

где n — количество наблюдений; y_i — истинное целевое значение наблюдения i, которое нужно спрогнозировать; $\hat{y}_i$ — прогнозируемое значение модели для y_i. MSE — это сумма квадратов расстояний между всеми прогнозируемыми и истинными значениями. Чем больше значение MSE, тем больше ошибка и, соответственно, хуже модель. С математической точки зрения возведение ошибок в квадрат имеет ряд преимуществ: ошибки будут иметь только положительные значения, а другая невостребованная особенность — несколько серьезных ошибок наказываются строже, чем множество маленьких, даже при равенстве абсолютных значений. В качестве примера представим две модели A и B с двумя наблюдениями в каждой:

- модель A имеет значения ошибок 0 и 10, таким образом, ее MSE равняется $0^2 + 10^2 = 100$;
- модель B имеет две ошибки со значением 5, таким образом, ее MSE равняется $5^2 + 5^2 = 50$.

Общее число ошибок моделей одинаковое и равно 10; тем не менее метод среднеквадратической ошибки показывает, что модель A (`MSE=100`) хуже модели B (`MSE=50`). На практике эта функция используется не часто (и остается теоретическим преимуществом), и MSE успешно применяется в качестве оценочной меры.

Важное замечание: по умолчанию в scikit-learn для разных аргументов параметра `scoring` лучшими считаются бóльшие значения. Однако это не распространяется на MSE, где бóльшие значения указывают на то, что модель хуже. По этой причине в scikit-learn при помощи аргумента `neg_mean_squared_error` можно использовать *отрицательную*, или *инвертированную* (negative), MSE.

Вместо показателя оценки регрессии мы можем использовать установленную по умолчанию метрику R^2 из *рецепта 11.2*, которая измеряет размер дисперсии целевого вектора, объясненный моделью.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2}{\sum_{i=1}^{n}\left(y_i - \bar{y}_i\right)^2},$$

где y_i — истинное целевое значение наблюдения i; $\hat{y}_i$ — спрогнозированное значение наблюдения i; $\bar{y}$ — средний показатель целевого вектора. Чем ближе значение R^2 к 1,0, тем лучше модель.

Дополнительные материалы для чтения

- Mean squared error ("Среднеквадратическая ошибка", Википедия; **https://oreil.ly/MWDlR**).
- Coefficient of determination ("Коэффициент детерминации", Википедия; **https://oreil.ly/lKKWk**).

11.9. Оценки моделей кластеризации

Задача

Вы применили алгоритм обучения без учителя для кластеризации данных. Теперь вы хотите оценить эффективность системы.

Решение

Воспользуемся *коэффициентами силуэта* (silhouette coefficients) для оценки качества кластеризации (имейте в виду, что это не оценка качества прогнозов):

```
# Загрузить библиотеки
import numpy as np
from sklearn.metrics import silhouette_score
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Создать матрицу признаков
features, _ = make_blobs(n_samples = 1000,
                         n_features = 10,
                         centers = 2,
                         cluster_std = 0.5,
                         shuffle = True,
                         random_state = 1)
```

```
# Кластеризовать данные при помощи метода k средних для прогнозирования классов
model = KMeans(n_clusters=2, random_state=1).fit(features)

# Получить спрогнозированные классы
target_predicted = model.labels_

# Оценить модель
silhouette_score(features, target_predicted)
0.8916265564072141
```

Обсуждение

Модель оценивания, обученная с учителем, сравнивает прогнозируемые значения (например, классы или количественные значения) с соответствующими истинными значениями целевого вектора. В свою очередь, методы кластеризации лучше всего подходят при отсутствии целевого вектора. Для многих метрик оценки качества кластеризации необходим целевой вектор, однако использование подходов с неконтролируемым обучением (например, кластеризации) при наличии целевого вектора излишне усложняет задачу.

Без целевого вектора мы не можем произвести сравнение прогнозируемых и истинных значений, но мы можем оценить свойства самих кластеров. Можно интуитивно предположить, что в "хороших" кластерах расстояния между наблюдениями одного кластера будут небольшими (т. е. это плотные кластеры (dense clusters)), в отличие от расстояния между разными кластерами (т. е. это обособленные кластеры (well-separated clusters)). В качестве одного значения для измерения обоих свойств выступают коэффициенты силуэта. Формально коэффициент силуэта наблюдения i можно представить в виде формулы:

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)},$$

где s_i — коэффициент силуэта наблюдения i; a_i — среднее расстояние между i и всеми наблюдениями этого класса; b_i — среднее расстояние между i и всеми наблюдениями ближайшего кластера другого класса. Функция `silhouette_score` возвращает среднее значение коэффициента силуэта всех наблюдений. Коэффициенты могут принимать значение от –1 до 1, где 1 означает плотные, обособленные кластеры.

Дополнительные материалы для чтения

- Документация по `silhouette_score` (**https://oreil.ly/gGjQj**).

11.10. Создание собственной оценочной метрики

Задача

Вы хотите произвести оценку модели, используя собственную метрику.

Решение

Создадим собственную функцию метрики и преобразуем ее в оценщик при помощи функции make_scorer из библиотеки scikit-learn:

```
# Загрузить библиотеку
from sklearn.metrics import make_scorer, r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.datasets import make_regression

# Создать матрицу признаков и целевой вектор
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   random_state = 1)

# Создать обучающий и тестовый наборы
features_train, features_test, target_train, target_test = train_test_split(
     features, target, test_size=0.10, random_state=1)

# Создать метрику
def custom_metric(target_test, target_predicted):
    # Рассчитать значение R-квадрат
    r2 = r2_score(target_test, target_predicted)
    # Получить значение R-квадрат
    return r2

# Создать оценщик и считать больший показатель лучшим
score = make_scorer(custom_metric, greater_is_better=True)

# Создать объект гребневой регрессии
classifier = Ridge()

# Обучить модель гребневой регрессии
model = classifier.fit(features_train, target_train)
```

```
# Применить собственный оценщик
score(model, features_test, target_test)
```

```
0.9997906102882058
```

Обсуждение

Хотя в scikit-learn есть большое количество встроенных метрик оценки качества модели, будет полезно задать собственные метрики. Для этого можно воспользоваться функцией make_scorer. Во-первых, нам нужно задать функцию, принимающую два аргумента — эталонный целевой вектор и спрогнозированные значения — и выводящую некоторое значение. Во-вторых, мы используем функцию make_scorer для создания оценщика, для которого нужно обязательно (при помощи параметра greater_is_better) определить предпочтительные значения — наименьшие или наибольшие.

В нашем решении собственная метрика (custom_metric) является простым примером, в который мы поместили встроенную метрику расчета значения R^2. На практике вместо функции custom_metric мы можем использовать любую метрику. Как можно заметить, наша собственная метрика расчета R^2 сравнивает полученные результаты со встроенным в scikit-learn методом r2_score:

```
# Спрогнозировать значения
target_predicted = model.predict(features_test)

# Рассчитать R-квадрат
r2_score(target_test, target_predicted)
```

```
0.9997906102882058
```

Дополнительные материалы для чтения

- Документация по make_scorer (**https://oreil.ly/-RqFY**).

11.11. Визуализация зависимости показателей от размера обучающего набора

Задача

Вам необходимо оценить степень влияния количества наблюдений вашего обучающего набора на показатели (правильность, F1-меру и т. д.).

Решение

Выведем график зависимости правильности (accuracy) от размера обучающей последовательности (рис. 11.3):

```
# Загрузить библиотеки
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve

# Загрузить данные
digits = load_digits()

# Создать матрицу признаков и целевой вектор
features, target = digits.data, digits.target

# Создать переменные обучающих и тестовых показателей кросс-валидации
# для обучающих наборов различных размеров
train_sizes, train_scores, test_scores = learning_curve(
    RandomForestClassifier(),  # Классификатор
    features,                  # Матрица признаков
    target,                    # Целевой вектор
    cv=10,                     # Количество блоков
    scoring='accuracy',        # Метрика качества
    n_jobs=-1,                 # Использовать все ядра CPU
    train_sizes=np.linspace(   # Обучающий набор
        0.01,
        1.0,
        50))                   # 50 размеров

# Задать средние значения и стандартные отклонения показателей обучающих наборов
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

# Задать средние значения и стандартные отклонения показателей тестовых наборов
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

# Построить прямые
plt.plot(train_sizes, train_mean, '--', color="#ab1ba8",  label="Обучающий набор")
plt.plot(train_sizes, test_mean, color="#0c2cc2", label="Кросс-валидация")
```

```
# Построить кривые
plt.fill_between(train_sizes, train_mean - train_std,
                 train_mean + train_std, color="#DDDDDD")
plt.fill_between(train_sizes, test_mean - test_std,
                 test_mean + test_std, color="#DDDDBB")

# Вывести график
plt.title("Кривая обучаемости")
plt.xlabel("Размер обучающего набора")
plt.ylabel("Показатель правильности")
plt.legend(loc="best")
plt.tight_layout()
plt.show()
```

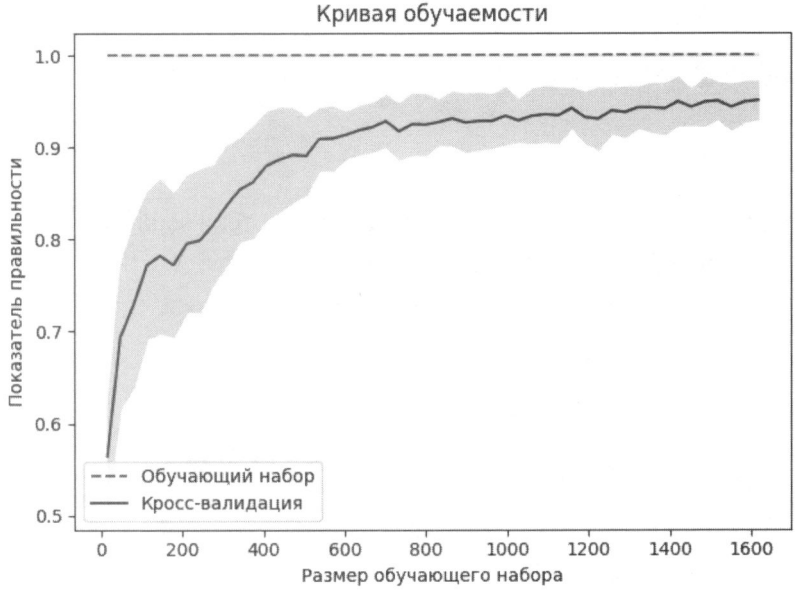

Рис. 11.3

Обсуждение

Мы можем использовать *кривые обучаемости* (learning curves) для визуализации качества (например, правильность, полнота) модели обучающего набора и кросс-валидации с постоянно увеличивающимся количеством наблюдений обучающего множества. С их помощью мы можем определить возможность улучшения нашей модели за счет сбора дополнительных данных.

В нашем примере мы выводим график правильности (accuracy) классификатора, состоящий из 50 обучающих наборов разного размера, включающий от 1% до 100% наблюдений. Увеличение показателя правильности моделей на основе кросс-валидации означает, что прибавление наблюдений (хотя на практике это не всегда возможно), вероятно, улучшит качество модели.

Дополнительные материалы для чтения

- Learning Curve ("Кривая обучаемости", документация scikit-learn; **https://oreil.ly/jAKwy**).

11.12. Создание текстового отчета с метриками качества

Задача

Вам нужно получить краткое описание качества классификатора.

Решение

Воспользуемся функцией `classification_report` библиотеки scikit-learn:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Загрузить данные
iris = datasets.load_iris()

# Создать матрицу признаков
features = iris.data

# Создать целевой вектор
target = iris.target

# Создать список имен целевого вектора
class_names = iris.target_names

# Создать обучающий и тестовый наборы
features_train, features_test, target_train, target_test = train_test_split(
    features, target, random_state=0)
```

```
# Создать переменную логистической регрессии
classifier = LogisticRegression()

# Обучить модель и сформировать прогноз
model = classifier.fit(features_train, target_train)
target_predicted = model.predict(features_test)

# Создать отчет о классификации
print(classification_report(target_test,
                            target_predicted,
                            target_names=class_names))
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        16
  versicolor       1.00      0.91      0.95        11
   virginica       0.92      1.00      0.96        11

    accuracy                           0.97        38
   macro avg       0.97      0.97      0.97        38
weighted avg       0.98      0.97      0.97        38
```

Обсуждение

Функция `classification_report` позволяет быстро просмотреть общие показатели качества, включая точность, полноту и F1-меру (*см. рецепт 11.4*). Столбец `support` отражает количество наблюдений каждого класса.

Дополнительные материалы для чтения

- Precision and recall ("Точность и полнота", Википедия; **https://oreil.ly/9mBSF**).

11.13. Визуализация влияния значений гиперпараметров

Задача

Вы хотите узнать, как изменение значений некоторых гиперпараметров влияет на качество модели.

Решение

Выведем график зависимости правильности (кривая валидации) от значения гиперпараметра (рис. 11.4):

```
# Загрузить библиотеки
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import validation_curve

# Загрузить данные
digits = load_digits()

# Создать матрицу признаков и целевой вектор
features, target = digits.data, digits.target

# Создать диапазон значений параметра
param_range = np.arange(1, 250, 2)

# Используя диапазон значений параметра, рассчитать правильность
# обучающего и тестового наборов
train_scores, test_scores = validation_curve(
    RandomForestClassifier(),   # Классификатор
    features,                   # Матрица признаков
    target,                     # Целевой вектор
    param_name="n_estimators",  # Оцениваемый гиперпараметр
    param_range=param_range,    # Диапазон значений гиперпараметра
    cv=3,                       # Количество блоков
    scoring="accuracy",         # Метрика качества
    n_jobs=-1)                  # Использовать все ядра CPU

# Вычислить среднее значение и стандартное отклонение показателей обучающего набора
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

# Вычислить среднее значение и стандартное отклонение показателей тестового набора
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

# Вывести график показателей правильности обучающего и тестового наборов
plt.plot(param_range, train_mean, label="Обучающий набор", '--', color="#ab1ba8")
plt.plot(param_range, test_mean, label="Кросс-валидация", color="dimgrey")
```

```
# Вывести график кривых правильности обучающего и тестового наборов
plt.fill_between(param_range, train_mean - train_std,
                 train_mean + train_std, color="gray")
plt.fill_between(param_range, test_mean - test_std,
                 test_mean + test_std, color="gainsboro")

# Вывести график
plt.title("Кривая валидации методом случайного леса")
plt.xlabel("Количество деревьев")
plt.ylabel("Показатель правильности")
plt.tight_layout()
plt.legend(loc="best")
plt.show()
```

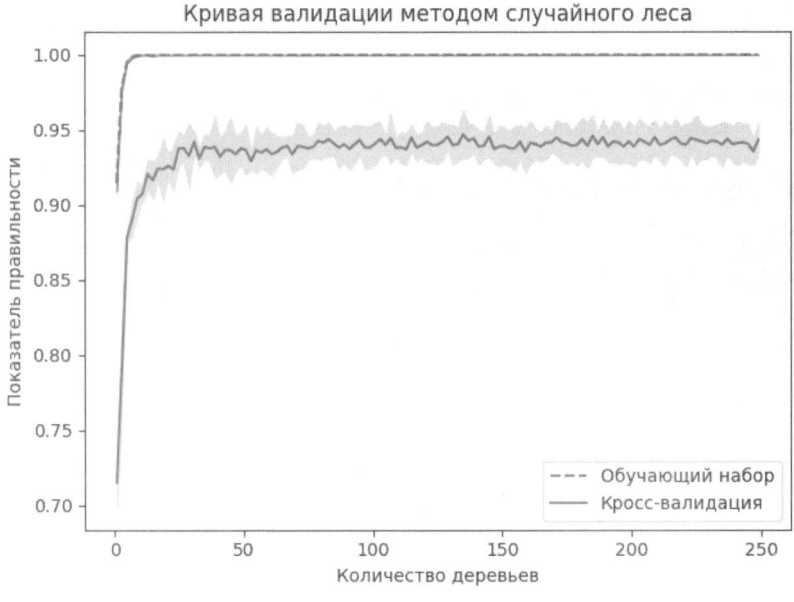

Рис. 11.4

Обсуждение

Большинство обучающих алгоритмов (в том числе рассматриваемых в этой книге) содержат гиперпараметры, которые необходимо задать перед началом обучения. К примеру, *алгоритм классификации случайного леса* (random forest classifier) создает лес деревьев решений, каждое из который определяет класс прогнозируемого наблюдения. Один гиперпараметр случайного леса означает количество деревьев. Значение гиперпараметра, как правило, определяется на этапе выбора модели (*см.*

главу 12). Тем не менее иногда бывает полезно визуализировать изменения качества модели при изменении значения гиперпараметра. В нашем примере мы выводим график изменения правильности (accuracy) классификации методом случайного леса для обучающего набора и модели кросс-валидации с постоянно увеличивающимся количеством деревьев. При небольшом количестве деревьев показатели обучающего набора и модели кросс-валидации низкие, что говорит о недостаточной отладке модели. Если же число деревьев увеличивается до 250, правильность обеих моделей выравнивается, что говорит об отсутствии практической значимости обучения большого леса.

В scikit-learn для расчета кривой валидации используется функция `validation_curve`, которая содержит три главных параметра.

`param_name`

Название изменяемого гиперпараметра.

`param_range`

Значение используемого гиперпараметра.

`scoring`

Оценочная метрика, применяемая к модели.

Дополнительные материалы для чтения

- Validation Curve ("Кривая валидации", документация scikit-learn; **https://oreil.ly/FH_kH**).

ГЛАВА 12

Выбор модели

Введение

В основе машинного обучения лежит процесс запоминания параметров с использованием обучающих алгоритмов, минимизирующих функцию потерь. Во многих алгоритмах машинного обучения (например, классификация методом опорного вектора и случайного леса) дополнительно представлены *гиперпараметры*, определяемые пользователем и регулирующие процесс запоминания параметров моделью. Как мы увидели ранее в этой книге, *параметрами* (иногда весами модели) называется то, что модель узнает в процессе обучения, тогда как гиперпараметры задаются нами (пользователями) вручную.

К примеру, случайные леса — это наборы деревьев решений (отсюда слово "лес"); однако количество этих деревьев не выводится алгоритмом и должно быть указано перед началом обучения. Это называется *оптимизацией гиперпараметров* (hyperparameter tuning или hyperparameter optimization), или *выбором статистической модели* (model selection). Кроме того, мы можем попробовать разные алгоритмы машинного обучения (например, попробовать классификацию методом опорного вектора и случайного леса, чтобы решить, в каком случае мы получаем более качественную модель).

Несмотря на вариативную терминологию по этой теме, под выбором модели в данной книге мы подразумеваем выбор лучших алгоритмов обучения и лучших гиперпараметров. Причина проста: представим, что у нас есть данные, которые мы хотим использовать в машинном обучении на основе классификации методом опорного вектора и случайного леса с 10 возможными значениями гиперпараметра в каждом. В результате нам придется выбирать наилучшую модель из 20 претендентов. В этой главе мы рассмотрим способы эффективного отбора наилучшей модели из имеющегося множества.

По ходу главы мы будем обращаться к конкретным гиперпараметрам, таким как C (обратная сила регуляризации). Не стоит волноваться, если какие-либо гиперпараметры вам не знакомы. Мы обсудим их более подробно в других главах. Воспринимайте гиперпараметры как настройки алгоритмов, которые необходимо указать перед началом обучения. Коротко говоря, поиск модели и подходящих гиперпараметров, позволяющих достичь наилучших показателей, заключается в экспериментировании, т. е. стоит попробовать различные комбинации и выбрать самую удачную.

12.1. Поиск лучших моделей методом полного перебора

Задача

Вы хотите выбрать лучшую модель, перебрав множество гиперпараметров.

Решение

Используем класс GridSearchCV библиотеки scikit-learn:

```
# Загрузить библиотеку
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать переменную логистической регрессии
logistic = linear_model.LogisticRegression(max_iter=500, solver='liblinear')

# Создать диапазон возможных значений гиперпараметра penalty
penalty = ['l1','l2']

# Создать диапазон возможных значений гиперпараметра регуляризации
C = np.logspace(0, 4, 10)

# Создать словарь возможных гиперпараметров
hyperparameters = dict(C=C, penalty=penalty)

# Создать поиск по сетке
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=0)

# Настроить поиск по сетке
best_model = gridsearch.fit(features, target)

# Вывести на экран лучшую модель
print(best_model.best_estimator_)
LogisticRegression(C=7.742636826811269, max_iter=500, penalty='l1',
                   solver='liblinear')
```

Обсуждение

Подход GridSearchCV (поиск по сетке) представляет собой метод грубой силы (brute-force) — перебор возможных сочетаний значений при выборе модели с помощью кросс-валидации. В частности, пользователь указывает множество возможных значений для одного или нескольких гиперпараметров и при помощи функции GridSearchCV обучает модель, пробуя каждое значение и/или их комбинацию. Лучшей считается модель с наиболее высокими показателями производительности.

В нашем примере мы использовали логистическую регрессию в качестве алгоритма обучения и два настраиваемых гиперпараметра: C и penalty (регуляризацию через штраф). Мы также указали два других параметра: решатель (solver) и максимальное количество итераций (max iterations). Не стоит волноваться, если эти термины вам не знакомы; мы рассмотрим их в последующих главах. Пока что нужно запомнить, что гиперпараметры C и penalty могут принимать диапазон значений, который нужно задать до начала обучения. Для C мы определяем 10 возможных значений:

```
np.logspace(0, 4, 10)
array([1.00000000e+00, 2.78255940e+00, 7.74263683e+00, 2.15443469e+01,
       5.99484250e+01, 1.66810054e+02, 4.64158883e+02, 1.29154967e+03,
       3.59381366e+03, 1.00000000e+04])
```

По аналогии определяем два значения для penalty: ['l1', 'l2']. Для каждого сочетания значений C и penalty мы обучаем модель и оцениваем ее при помощи k-блочной кросс-валидации. В нашем случае получилось 10 возможных значений C, 2 возможных значения penalty и 5 блоков (folds). В результате получилось $10 \times 2 \times 5 = 100$ возможных моделей, из которых нужно выбрать лучшую.

После завершения работы объекта класса GridSearchCV мы можем просмотреть значения гиперпараметров лучшей модели:

```
# Вывести лучшие гиперпараметры
print('Best Penalty:', best_model.best_estimator_.get_params()['penalty'])
print('Best C:', best_model.best_estimator_.get_params()['C'])
Best Penalty: l1
Best C: 7.742636826811269
```

После обнаружения лучших гиперпараметров объект GridSearchCV по умолчанию заново обучит модель, применив их ко всему набору данных (вместо исключения блока из кросс-валидации). Мы можем использовать эту модель для прогнозирования значений наравне с другими моделями scikit-learn:

```
# Спрогнозировать целевой вектор
best_model.predict(features)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
```

```
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Стоит выделить один параметр класса `GridSearchCV` — `verbose`. Хотя это и не обязательно, в процессе длительного поиска может быть полезно его периодически отслеживать. При помощи параметра `verbose` можно определить количество сообщений в процессе поиска, где `0` означает отсутствие оповещений, а значения от `1` до `3` — вывод дополнительных сообщений.

Дополнительные материалы для чтения

- Документация по `GridSearchCV` (**https://oreil.ly/XlMPG**).

12.2. Случайный поиск лучших моделей

Задача

Вы хотите осуществить поиск лучшей модели с меньшими вычислительными затратами, чем исчерпывающий поиск.

Решение

Воспользуемся классом `RandomizedSearchCV` библиотеки scikit-learn:

```
# Загрузить библиотеку
from scipy.stats import uniform
from sklearn import linear_model, datasets
from sklearn.model_selection import RandomizedSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать переменную логистической регрессии
logistic = linear_model.LogisticRegression(max_iter=500, solver='liblinear')

# Создать диапазон возможных значений гиперпараметра penalty
penalty = ['l1', 'l2']

# Создать диапазон возможных значений гиперпараметра регуляризации
C = uniform(loc=0, scale=4)
```

```
# Создать выбор гиперпараметров
hyperparameters = dict(C=C, penalty=penalty)

# Создать переменную случайного поиска
randomizedsearch = RandomizedSearchCV(
    logistic, hyperparameters, random_state=1, n_iter=100, cv=5, verbose=0,
    n_jobs=-1)

# Настроить случайный поиск
best_model = randomizedsearch.fit(features, target)

# Вывести на экран лучшую модель
print(best_model.best_estimator_)
LogisticRegression(C=1.668088018810296, max_iter=500, penalty='l1',
                   solver='liblinear')
```

Обсуждение

В *рецепте 12.1* мы использовали объект класса `GridSearchCV` с заданными пользователем гиперпараметрами для поиска лучшей модели на основе ее показателей. Однако вместо менее эффективного метода грубой силы `GridSearchCV` мы можем осуществить поиск среди заданного количества случайных комбинаций значений гиперпараметров на основе распределения пользователем (например, нормальное (normal), равномерное (uniform)). В scikit-learn случайный поиск реализуется при помощи класса `RandomizedSearchCV`.

При использовании `RandomizedSearchCV` мы указываем распределение, в рамках которого алгоритмы scikit-learn случайным образом без замены отбирают значения гиперпараметров. В качестве примера мы выберем 10 случайных значений, используя равномерное (uniform) распределение в диапазоне от 0 до 4:

```
# Обозначить равномерное распределение в диапазоне от 0 до 4, выбрать 10 значений
uniform(loc=0, scale=4).rvs(10)
array([3.95211699, 0.30693116, 2.88237794, 3.00392864, 0.43964702,
       1.46670526, 0.27841863, 2.56541664, 2.66475584, 0.79611958])
```

В качестве другого способа мы можем указать список значений, например, два значения гиперпараметра регуляризации `penalty` — `['l1', 'l2']`; в таком случае `RandomizedSearchCV` случайно выберет образец из списка с заменой.

Как и при использовании класса `GridSearchCV`, на выходе мы получаем значения гиперпараметров лучшей модели:

```
# Вывести лучшие гиперпараметры
print('Best Penalty:', best_model.best_estimator_.get_params()['penalty'])
print('Best C:', best_model.best_estimator_.get_params()['C'])
```

```
Best Penalty: l1
Best C: 1.668088018810296
```

Аналогично классу GridSearchCV, после окончания поиска класс RandomizedSearchCV настраивает новую модель, применяя полученные гиперпараметры ко всему набору данных. Мы можем использовать эту модель наравне с другими моделями scikit-learn, например для прогнозирования значений:

```
# Спрогнозировать целевой вектор
best_model.predict(features)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Количество вариантов сочетаний гиперпараметров (т. е. количество потенциально лучших обученных моделей) указывается при помощи параметра n_iter (количество итераций). Стоит отметить, что, хотя сам по себе класс RandomizedSearchCV не намного быстрее GridSearchCV, проверка меньшего числа комбинаций позволяет достичь схожего качества за меньшее время.

Дополнительные материалы для чтения

- Документация по RandomizedSearchCV (**https://oreil.ly/rpiSs**).
- Random Search for Hyper-Parameter Optimization ("Использование случайного поиска для оптимизации гиперпараметров"; **https://oreil.ly/iBcbo**).

12.3. Поиск лучшей модели среди множества алгоритмов обучения

Задача

Вам необходимо выбрать лучшую модель, исследуя множество алгоритмов обучения и соответствующих им гиперпараметров.

Решение

Создадим словарь из возможных алгоритмов машинного обучения и их гиперпараметров, которые затем используем в качестве области поиска класса GridSearchCV:

```
# Загрузить библиотеки
import numpy as np
```

```
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

# Указать случайное начальное значение
np.random.seed(0)

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать конвейер
pipe = Pipeline([("classifier", RandomForestClassifier())])

# Создать словарь из возможных алгоритмов и их гиперпараметров
search_space = [{"classifier": [LogisticRegression(max_iter=500,
       solver='liblinear')],
                 "classifier__penalty": ['l1', 'l2'],
                 "classifier__C": np.logspace(0, 4, 10)},
                {"classifier": [RandomForestClassifier()],
                 "classifier__n_estimators": [10, 100, 1000],
                 "classifier__max_features": [1, 2, 3]}]

# Создать поиск по сетке
gridsearch = GridSearchCV(pipe, search_space, cv=5, verbose=0)

# Настроить поиск по сетке
best_model = gridsearch.fit(features, target)

# Вывести на экран лучшую модель
print(best_model.best_estimator_)
Pipeline(steps=[('classifier',
                 LogisticRegression(C=7.742636826811269, max_iter=500,
                                    penalty='l1', solver='liblinear'))])
```

Обсуждение

В двух предыдущих задачах мы определяли лучшую модель на основе возможных значений гиперпараметра алгоритма обучения. Но что делать, если мы не знаем,

какой алгоритм лучше использовать? В scikit-learn мы можем включать алгоритмы обучения в область поиска. В данном примере мы остановились на двух алгоритмах: логистической регрессии и классификаторе случайного леса. У каждого из них свои гиперпараметры, и мы определяем их возможные значения при помощи `classifier__[имя_гиперпараметра]`. К примеру, в случае логистической регрессии мы задаем при помощи словарей возможные значения регуляризации пространства гиперпараметра `C` и возможные типы регуляризационных ограничений `penalty`:

```
{'classifier': [LogisticRegression(max_iter=500, solver='liblinear')],
 'classifier__penalty': ['l1', 'l2'],
 'classifier__C': np.logspace(0, 4, 10)}
```

Мы также можем создать схожий словарь параметров классификатора случайного леса:

```
{'classifier': [RandomForestClassifier()],
 'classifier__n_estimators': [10, 100, 1000],
 'classifier__max_features': [1, 2, 3]}
```

После окончания поиска мы можем просмотреть данные лучших алгоритмов обучения и гиперпараметров модели, используя атрибут `best_estimator_`:

```
# Вывести на экран лучшую модель
print(best_model.best_estimator_.get_params()["classifier"])
LogisticRegression(C=7.742636826811269, max_iter=500, penalty='l1',
                   solver='liblinear')
```

Как и в предыдущих случаях, после настройки поиска лучшей модели мы можем использовать полученную модель вместе с другими в scikit-learn:

```
# Спрогнозировать целевой вектор
best_model.predict(features)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

12.4. Поиск лучшей модели в процессе первичной обработки

Задача

Вы хотите добавить предварительную обработку к этапу выбора модели.

Решение

Создадим конвейер, в который добавим этап предобработки и все необходимые параметры:

```
# Загрузить библиотеки
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Указать случайное начальное значение
np.random.seed(0)

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект предобработки, включающий признаки функции StandardScaler и PCA
preprocess = FeatureUnion([("std", StandardScaler()), ("pca", PCA())])

# Создать конвейер
pipe = Pipeline([("preprocess", preprocess),
                 ("classifier", LogisticRegression(max_iter=1000,
                     solver='liblinear'))])

# Создать пространство возможных значений
search_space = [{"preprocess__pca__n_components": [1, 2, 3],
                 "classifier__penalty": ["l1", "l2"],
                 "classifier__C": np.logspace(0, 4, 10)}]

# Создать поиск по сетке
clf = GridSearchCV(pipe, search_space, cv=5, verbose=0, n_jobs=-1)

# Настроить поиск по сетке
best_model = clf.fit(features, target)

# Вывести на экран лучшую модель
print(best_model.best_estimator_)
```

```
Pipeline(steps=[('preprocess',
                 FeatureUnion(transformer_list=[('std', StandardScaler()),
                                                ('pca', PCA(n_components=1))])),
                ('classifier',
                 LogisticRegression(C=7.742636826811269, max_iter=1000,
                                    penalty='l1', solver='liblinear'))])
```

Обсуждение

Перед началом обучения модели нам часто приходится предварительно обрабатывать данные. Предобработка на этапе выбора модели требует от нас повышенного внимания к управлению данными. Во-первых, в классе `GridSearchCV` используется кросс-валидация для определения наиболее качественной модели. В случае кросс-валидации мы извлекаем и изолируем один блок (fold) — тестовый набор, — который выпадает из всех этапов предобработки (например, масштабирование или стандартизация). По этой причине мы не можем предварительно обрабатывать данные параллельно с объектом класса `GridSearchCV`. Вместо этого этапы предобработки нужно включить в алгоритм действий при выполнении `GridSearchCV`.

Эту на первый взгляд сложную задачу можно легко решить при помощи инструментов scikit-learn. Класс `FeatureUnion` позволяет правильно объединить несколько объектов-преобразователей. В нашем примере мы использовали `FeatureUnion` для объединения двух этапов предобработки: стандартизацию значений признаков (`StandardScaler`) и метод главных компонент (`PCA`). Так мы получили объект `preprocess`, включающий оба шага. Затем мы добавили предобработку в конвейер с нашим алгоритмом машинного обучения. В результате мы можем применить инструменты scikit-learn для точной настройки, преобразования и обучения моделей с разными комбинациями гиперпараметров.

Во-вторых, некоторые предобработанные модели имеют собственные параметры, которые обычно задает пользователь. Например, для снижения размерности при помощи PCA пользователь определяет количество главных компонент, используемых для преобразования набора признаков. В идеальной ситуации мы бы выбрали такое количество компонент, при котором модель показывает лучшую производительность на основе какой-либо оценочной метрики.

Механизмы scikit-learn позволяют упростить эту задачу. Значения возможных параметров компонент, включенных в область поиска, приравниваются к другим гиперпараметрам, среди которых осуществляется поиск. В нашем примере мы определили `features__pca__n_components': [1, 2, 3]` в области поиска, другими словами, мы хотим знать, можно ли получить лучшую модель при помощи одной, двух или трех главных компонент.

После завершения поиска модели мы можем просмотреть предобработанные значения лучшей модели. К примеру, мы можем вывести количество главных компонент:

```
# Вывести n лучших компонент
best_model.best_estimator_.get_params()['preprocess__pca__n_components']
```

```
1
```

12.5. Ускорение поиска модели через распараллеливание

Задача

Вам необходимо ускорить процесс поиска модели.

Решение

Указав параметр n_jobs=-1, мы можем задействовать все ядра системы, что позволит обучать несколько моделей одновременно:

```
# Загрузить библиотеки
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать переменную логистической регрессии
logistic = linear_model.LogisticRegression(max_iter=500, solver='liblinear')

# Создать диапазон возможных значений гиперпараметра регуляризации
penalty = ["l1", "l2"]

# Создать диапазон возможных значений C
C = np.logspace(0, 4, 1000)

# Создать словарь возможных гиперпараметров
hyperparameters = dict(C=C, penalty=penalty)

# Создать переменную случайного поиска
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=-1, verbose=1)

# Настроить случайный поиск
best_model = gridsearch.fit(features, target)

# Вывести на экран лучшую модель
print(best_model.best_estimator_)
```

```
Fitting 5 folds for each of 2000 candidates, totalling 10000 fits
# Настройка 5 блоков для 2000 возможных моделей, всего 10000
LogisticRegression(C=5.926151812475554, max_iter=500, penalty='l1',
                   solver='liblinear')
```

Обсуждение

В этой главе мы использовали примеры с небольшим количеством возможных моделей, чтобы не замедлять процесс выполнения кода. На практике у нас могут быть тысячи и десятки тысяч моделей для обучения. По этой причине на поиск наиболее качественной модели могут уйти многие часы.

Для того чтобы ускорить этот процесс, scikit-learn предоставляет возможность обучать несколько моделей одновременно. Коротко говоря, число одновременно обучаемых моделей в scikit-learn ограничено количеством ядер процессора. В современных ноутбуках есть как минимум четыре ядра; таким образом (предположим, что вы работаете на ноутбуке), вы можете одновременно обучать до четырех моделей. Это существенно ускорит процесс поиска подходящей модели. Количество параллельно обучаемых моделей определяет параметр `n_jobs`.

В нашем примере для `n_jobs` мы указали значение `-1`, что значит задействовать все ядра. По умолчанию параметр `n_jobs` имеет значение `1`, т. е. можно использовать только одно ядро. Мы можем наглядно убедиться в существенной разнице во времени выполнения, повторно запустив класс `GridSearchCV` из примера, но с параметром `n_jobs=1` (точное время выполнения зависит от конкретной системы):

```
# Создать переменную случайного поиска с использованием одного ядра
clf = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=1, verbose=1)

# Настроить случайный поиск
best_model = clf.fit(features, target)

# Вывести на экран лучшую модель
print(best_model.best_estimator_)
Fitting 5 folds for each of 2000 candidates, totalling 10000
# Настройка 5 блоков для 2000 возможных моделей, всего 10000 моделей)
fits LogisticRegression(C=5.926151812475554, max_iter=500, penalty='l1',
                        solver='liblinear')
```

12.6. Ускорение поиска модели с помощью алгоритмических методов

Задача

Вам нужно ускорить поиск модели, не используя дополнительные вычислительные мощности.

Решение

Если вы уже отобрали алгоритмы машинного обучения, можно воспользоваться алгоритмической оптимизацией (tuning) гиперпараметров при помощи кросс-валидации `LogisticRegressionCV` в scikit-learn:

```
# Загрузить библиотеку
from sklearn import linear_model, datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать переменную кросс-валидационной логистической регрессии
logit = linear_model.LogisticRegressionCV(Cs=100, max_iter=500,
        solver='liblinear')

# Обучить модель
logit.fit(features, target)

# Вывести модель на экран
print(logit)
LogisticRegressionCV(Cs=100, max_iter=500, solver='liblinear')
```

Обсуждение

Иногда процесс поиска лучших гиперпараметров можно значительно ускорить, если вместо методов грубой силы или случайного поиска воспользоваться свойствами алгоритма машинного обучения. Многие алгоритмы обучения scikit-learn (например, гребневая (ridge), лассо (lasso) и эластичная сеть (elastic net regression)) имеют встроенный алгоритмический метод на основе кросс-валидации. К примеру, класс `LogisticRegression` запускает классификацию со стандартной логической регрессией, тогда как класс `LogisticRegressionCV` запускает классификацию с более эффективной кросс-валидационной регрессией, при помощи которой можно определить оптимальное значение гиперпараметра `C`.

Метод scikit-learn `LogisticRegressionCV` включает параметр регуляции `C`. Если данные представлены списком (list), параметр `C` содержит несколько возможных значений гиперпараметра, из которых нужно выбрать одно. Если данные представлены в числовом формате (integer), параметр `C` создает список возможных значений. Вероят-

ные значения изображаются логарифмической последовательностью от 0,0001 до 1,0000 (диапазон оптимальных значений C).

Вместе с тем класс LogisticRegressionCV может выполнять поиск значений только для C, что является существенным недостатком. В *рецепте 12.1* область возможных значений гиперпараметра включала как C, так и другой гиперпараметр (регуляризация penalty). Такое ограничение присутствует во многих модельно-ориентированных подходах кросс-валидации scikit-learn.

Дополнительные материалы для чтения

- Документация по LogisticRegressionCV (**https://oreil.ly/uguJi**).
- Model specific cross-validation ("Модельно-ориентированная кросс-валидация", документация scikit-learn; **https://oreil.ly/6xfn6**).

12.7. Оценка качества после выбора модели

Задача

Вы хотите оценить качество выбранной модели.

Решение

Воспользуемся вложенной кросс-валидацией (nested cross-validation), чтобы избежать необъективной оценки:

```
# Загрузить библиотеки
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV, cross_val_score

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать переменную логистической регрессии
logistic = linear_model.LogisticRegression(max_iter=500, solver='liblinear')

# Задать диапазон из 20 возможных значений C
C = np.logspace(0, 4, 20)
```

```
# Создать выбор гиперпараметров
hyperparameters = dict(C=C)

# Создать поиск по сетке
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=-1, verbose=0)

# Запустить вложенную кросс-валидацию и получить среднее значение
cross_val_score(gridsearch, features, target).mean()
0.9733333333333334
```

Обсуждение

Идея вложенной кросс-валидации в процессе поиска модели многим может показаться сложной в начале освоения. Как вы помните, в случае k-блочной кросс-валидации мы обучали модель на $k - 1$ блоках данных, использовали эту модель для прогнозирования значений оставшегося блока и затем оценивали нашу модель, сравнивая спрогнозированные значения и истинными. Эта процедура повторялась k раз.

В случае способов поиска модели, описанных в этой главе (`GridSearchCV` и `RandomizedSearchCV`), мы использовали кросс-валидацию для оценки значений гиперпараметра, созданного наиболее качественной моделью. Однако здесь возникает достаточно серьезная проблема: поскольку мы используем данные для поиска наилучших значений гиперпараметра, мы не можем применить эти же данные для оценки качества модели. Как ее можно решить? Завернуть процесс кросс-валидации для поиска модели в другой процесс кросс-валидации! Во вложенной кросс-валидации "внутренний" процесс отвечает за выбор модели, а "внешний" — за объективную оценку качества модели. В нашем примере внутренней кросс-валидацией является объект класса `GridSearchCV`, который при помощи метода `cross_val_score` заворачивается во внешнюю кросс-валидацию.

Для того чтобы не запутаться, давайте рассмотрим простой пример. Сначала укажем параметр `verbose=1`, чтобы визуализировать процесс:

```
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=1)
```

Затем запустим `gridsearch.fit(features, target)` — внутренний процесс кросс-валидации для поиска наилучшей модели:

```
best_model = gridsearch.fit(features, target)
Fitting 5 folds for each of 20 candidates, totalling 100 fits
# Настройка 5 блоков для 20 возможных моделей, всего 100 моделей
```

Как мы можем понять из вывода, в результате внутренней кросс-валидации 20 вероятных моделей было обучено 5 раз, что в итоге дало 100 моделей. Теперь поместим `clf` внутрь нового процесса кросс-валидации (по умолчанию 5 блоков):

```
scores = cross_val_score(gridsearch, features, target)
Fitting 5 folds for each of 20 candidates, totalling 100 fits
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 20 candidates, totalling 100 fits
# Настройка 5 блоков для 20 возможных моделей, всего 100 моделей
```

Из вывода видно, что после кросс-валидация для поиска лучшей модели было обучено 20 моделей 5 раз, а после внешней 5-блочной кросс-валидации модель получила оценку, и в результате было создано 500 обученных моделей.

ГЛАВА 13
Линейная регрессия

Введение

Линейная регрессия является одним из наиболее простых алгоритмов машинного обучения с учителем. Если вам доводилось посещать вводный курс по статистике в колледже, то на завершающем этапе обучения, скорее всего, он был посвящен изучению линейной регрессии. Линейная регрессия и ее производные по-прежнему остаются одним из основных и очень удобных методов прогнозирования при работе с количественными значениями вектора (например, стоимость жилья, возраст). В этой главе мы рассмотрим разнообразные методы линейной регрессии (и некоторые производные), которые можно будет использовать для создания качественных моделей.

13.1. Настройка линейной связи

Задача

Вы хотите обучить модель, которая представляет линейную зависимость между признаком и целевым вектором.

Решение

Используем линейную регрессию (в библиотеке scikit-learn это класс `LinearRegression`):

```
# Загрузить библиотеку
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression

# Сгенерировать матрицу признаков и целевой вектор
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   n_informative = 2,
                                   n_targets = 1,
                                   noise = 0.2,
```

```
                                    coef = False,
                                    random_state = 1)

# Создать объект линейной регрессии
regression = LinearRegression()

# Настроить линейную регрессию
model = regression.fit(features, target)
```

Обсуждение

Линейная регрессия строится на гипотезе, что признак и вектор линейно зависимы. Другими словами, существует непрерывное влияние (коэффициент, вес или параметр) признаков на целевой вектор. Для простоты объяснения в нашем примере мы использовали лишь три признака для обучения модели. В результате наша линейная модель выглядит так:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \hat{\beta}_3 x_3 + \varepsilon,$$

где $\hat{y}$ — целевой показатель; x_i — данные одного признака; $\hat{\beta}_1, \hat{\beta}_2$ и $\hat{\beta}_3$ — коэффициенты, которые выводятся из настройки модели; ε — значение ошибки. После настройки модели мы можем просмотреть значение каждого параметра. Например, при помощи атрибута `intercept_` мы можем увидеть значение $\hat{\beta}_0$, также называемое *смещением* (bias), или *отрезком* (intercept):

```
# Вывести значение отрезка
model.intercept_
-0.009650118178816669
```

Значения $\hat{\beta}_1$ и $\hat{\beta}_2$ можно просмотреть при помощи атрибута `coef_`:

```
# Вывести значения коэффициентов признака
model.coef_
array([1.95531234e-02, 4.42087450e+01, 5.81494563e+01])
```

В нашем наборе данных целевое значение — это случайно сгенерированная непрерывная переменная:

```
# Первое значение целевого вектора
target[0]
-20.870747595269407
```

При помощи метода `predict` мы можем спрогнозировать выходные значения на основе входных признаков:

```
# Спрогнозировать целевое значение для первого наблюдения
model.predict(features)[0]
-20.861927709296808
```

Неплохо! Результат нашей модели имеет погрешность всего 0,01!

Важным преимуществом линейной регрессии является интерпретируемость, поскольку мы знаем, что коэффициенты модели получаются в результате изменения одного элемента целевого вектора. В нашем случае коэффициент первого признака равен примерно –0,02, т. е. целевое значение будет меняться при изменении каждого элемента первого признака.

При помощи функции `score` мы можем узнать, насколько эффективно наша модель обрабатывает данные:

```
# Вывести показатель эффективности обработки обучающих данных моделью
print(model.score(features, target))
```

```
0.9999901732607787
```

По умолчанию в scikit-learn показателем линейной регрессии является R^2 со значениями от 0,0 (худший) до 1,0 (лучший). Как видно из примера, наш показатель близок к эталонному значению 1,0. Однако стоит отметить, что мы оцениваем модель на основе обработанных ею данных (обучающий набор), хотя на практике мы бы использовали извлеченный тестовый набор. Тем не менее это очень хороший показатель для реальной модели.

13.2. Управление взаимосвязанными коэффициентами

Задача

У вас есть некоторое множество, в котором влияние одного признака на целевую переменную зависит от состояния другого признака.

Решение

При помощи класса `PolynomialFeatures` библиотеки scikit-learn создадим условие взаимодействия (interaction term) для поиска этой зависимости:

```
# Загрузить библиотеку
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.datasets import make_regression

# Сгенерировать матрицу признаков и целевой вектор
features, target = make_regression(n_samples = 100,
                                   n_features = 2,
                                   n_informative = 2,
                                   n_targets = 1,
                                   noise = 0.2,
                                   coef = False,
                                   random_state = 1)
```

```
# Создать условие взаимодействия
interaction = PolynomialFeatures(
    degree=3, include_bias=False, interaction_only=True)
features_interaction = interaction.fit_transform(features)

# Создать объект линейной регрессии
regression = LinearRegression()

# Настроить линейную регрессию
model = regression.fit(features_interaction, target)
```

Обсуждение

Иногда влияние признака на целевую переменную в разной степени может зависеть от другого признака. Возьмем пример с кофе, где нам даны два бинарных признака — наличие сахара (`sugar`) и факт размешивания (`stirred`), — и нужно предугадать сладость кофе. Простое добавление сахара в кофе (`sugar=1`, `stirred=0`) не сделает его более сладким (весь сахар останется на дне!), так же как и простое размешивание без сахара (`sugar=0`, `stirred=1`) не добавит сладости. Кофе станет сладким только при взаимодействии двух признаков (`sugar=1`, `stirred=1`) — добавления *и* размешивания сахара. Другими словами, признаки `sugar` и `stirred` взаимозависимы. В такой ситуации мы говорим об *эффекте взаимосвязи* (interaction effect) между признаками `sugar` и `stirred`.

Этот эффект можно выразить через добавление нового признака, значение которого выводится из взаимодействующих признаков:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \hat{\beta}_3 x_1 x_2 + \varepsilon,$$

где x_1 и x_2 — значения `sugar` и `stirred` соответственно; $x_1 x_2$ — их взаимосвязь.

В нашем решении мы использовали набор данных из двух признаков. Ниже представлены значения первого наблюдения каждого признака:

```
# Вывести значения признаков первого наблюдения
features[0]
array([0.0465673 , 0.80186103])
```

Для создания условия взаимодействия умножим значения каждого наблюдения:

```
# Загрузить библиотеку
import numpy as np

# Умножить значения первого и второго признаков для каждого наблюдения
interaction_term = np.multiply(features[:, 0], features[:, 1])
```

Далее мы можем просмотреть условие взаимодействия для первого наблюдения:

```
# Вывести условие взаимодействия для первого наблюдения
interaction_term[0]
```

```
0.037340501965846186
```

Мы часто прослеживаем взаимосвязь между двумя признаками, однако иногда эта связь может быть не столь очевидной. В таких случаях в scikit-learn можно использовать объект класса `PolynomialFeatures`, чтобы с его помощью создать условия взаимодействия (interaction terms) для всех вариантов соотнесения признаков. Затем мы можем воспользоваться методами поиска модели для определения комбинации признаков и условий взаимодействия, на основе которых получится наиболее качественная модель.

При использовании `PolynomialFeatures` для создания условий взаимодействия необходимо указать три важных параметра. Самым главным параметром является `interaction_only=True`, который сообщает объекту класса `PolynomialFeatures`, что нужно вернуть только условия взаимодействия (а не полиномиальные признаки, которые мы рассмотрим в *рецепте 13.3*). Класс `PolynomialFeatures` по умолчанию добавляет признак со значением `1`, называемый *смещением* (bias). Мы можем отменить это действие, указав `include_bias=False`. Наконец, параметр `degree` определяет наибольшее число признаков, из которых создаются условия взаимодействия (в том случае, если мы хотим создать условие взаимодействия из соотнесения трех признаков). Мы можем просмотреть вывод класса `PolynomialFeatures` из нашего примера и убедиться, что значения признака первого наблюдения и условие взаимодействия совпадают с нашими вычислениями:

```
# Вывести значения первого наблюдения
features_interaction[0]
```

```
array([0.0465673 , 0.80186103, 0.0373405 ])
```

13.3. Настройка нелинейной взаимосвязи

Задача

Вы хотите настроить модель с нелинейной зависимостью.

Решение

Создадим полиномиальную регрессию (polynomial regression), добавив полиномиальные признаки в модель линейной регрессии:

```
# Загрузить библиотеку
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.datasets import make_regression
```

```
# Сгенерировать матрицу признаков и целевой вектор
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   n_informative = 2,
                                   n_targets = 1,
                                   noise = 0.2,
                                   coef = False,
                                   random_state = 1)

# Создать полиномиальные признаки x^2 и x^3
polynomial = PolynomialFeatures(degree=3, include_bias=False)
features_polynomial = polynomial.fit_transform(features)

# Создать объект линейной регрессии
regression = LinearRegression()

# Настроить линейную регрессию
model = regression.fit(features_polynomial, target)
```

Обсуждение

До этого момента мы рассматривали модели только с линейной зависимостью. Примером линейной зависимости является количество этажей в здании и его высота. При линейной зависимости мы основываемся на допущении, что существует практически непрерывное влияние количества этажей на высоту здания, другими словами, здание с 20 этажами примерно в два раза выше здания с 10 этажами, которое примерно в два раза выше здания с 5 этажами. Тем не менее многие отношения не имеют прямой линейной зависимости.

Создаваемые нами модели часто выражают нелинейную связь: например, это соотношение количества академических часов и итоговых результатов тестов студента. Мы можем логически предположить, что существует большая разница в итоговых результатах студента, занимавшегося один час, и студента, не занимавшегося вообще. Однако разница между результатами двух студентов, обучавшихся 99 часов и 100 часов соответственно, будет намного менее заметной. Таким образом, влияние одного часа обучения на итоговый результат студента уменьшается при увеличении количества часов.

Полиномиальная регрессия является расширенной производной линейной регрессии, которая позволяет работать с нелинейной взаимосвязью. Мы можем получить полиномиальную регрессию, преобразовав линейную функцию из *рецепта 13.1*

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \varepsilon$$

в полиномиальную функцию путем добавления полиномиальных признаков:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_1^2 + \ldots + \hat{\beta}_d x_1^d + \varepsilon,$$

где d является степенью многочлена. Каким же образом мы можем применить линейную регрессию к нелинейной функции? Для этого мы не меняем способ настройки модели при помощи линейной регрессии, а добавляем полиномиальные признаки. Другими словами, мы не сообщаем линейной регрессии, что x^2 является квадратичным преобразованием x. Это лишь еще одна переменная.

Попробуем по порядку описать процесс. Для получения модели с нелинейными отношениями мы можем создать новые признаки путем возведения имеющегося признака x в степень: x^2, x^3 и т. д. Чем больше новых признаков мы добавим, тем более гибкой будет "прямая" нашей модели. Представим, что нам нужно создать многочлены до третьей степени. Для того чтобы упростить объяснение, мы сосредоточимся на одном наблюдении (первое наблюдение в нашем наборе данных) `x[0]`:

```
# Вывести первое наблюдение
features[0]
```

```
array([-0.61175641])
```

Для создания полиномиального признака нам нужно возвести первое наблюдение во вторую степень — x_1^2:

```
# Вывести первое наблюдение, возведенное во вторую степень x^2
features[0]**2
```

```
array([0.37424591])
```

Таким образом мы получили новый признак. Затем мы возведем значение первого наблюдения в третью степень — x_1^3:

```
# Вывести первое наблюдение, возведенное в третью степень x^3
features[0]**3
```

```
array([-0.22894734])
```

В результате, если мы добавим все три признака (x, x^2 и x^3) в нашу матрицу и запустим линейную регрессию, то получим полиномиальную регрессию:

```
# Вывести значения первого наблюдения x, x^2 и x^3
features_polynomial[0]
```

```
array([-0.61175641,  0.37424591, -0.22894734])
```

Класс `PolynomialFeatures` имеет два основных параметра. Во-первых, это параметр `degree`, который определяет наибольшую степень полиномиальных признаков. К примеру, указав `degree=3`, мы получим значения x^2 и x^3. Во-вторых, класс `PolynomialFeatures` по умолчанию включает признаки, состоящие из одного значения (так называемое смещение (bias)). Мы можем это изменить, указав `include_bias=False`.

13.4. Уменьшение дисперсии через регуляризацию

Задача

Вам нужно уменьшить дисперсию вашей модели с линейной регрессией.

Решение

Используем алгоритм обучения, например, гребневой регрессией или лассо-регрессией, который включает *штраф за уменьшение* (shrinkage penalty) или *регуляризацию*:

```
# Загрузить библиотеку
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_regression

# Сгенерировать матрицу признаков и целевой вектор
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   n_informative = 2,
                                   n_targets = 1,
                                   noise = 0.2,
                                   coef = False,
                                   random_state = 1)

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект гребневой регрессии, указав значение альфа
regression = Ridge(alpha=0.5)

# Настроить линейную регрессию
model = regression.fit(features_standardized, target)
```

Обсуждение

При использовании стандартной линейной регрессии цель обучения модели заключается в снижении суммы среднеквадратической ошибки между истинным (y_i) и спрогнозированным $(\hat{y}_i)$ целевыми значениями или остаточной суммы квадратов (residual sum of squares, RSS):

$$\text{RSS} = \sum_{i=1}^{n} \left(y_i - \hat{y}_i \right)^2.$$

В случае регрессии с регуляризацией происходит схожий процесс, однако он предполагает снижение остаточной суммы квадратов и некоторое ограничение конечного размера значений коэффициентов — так называемый *штраф за уменьшение* (shrinkage penalty), — чтобы предотвратить "сжатие" модели. Существует два основных метода регуляризации в линейной регрессии: гребневая и лассо-регрессия. Единственным формальным отличием является тип используемых штрафов за уменьшение. В *гребневой регрессии* штраф за уменьшение равен произведению оптимизируемого гиперпризнака и суммы квадратов всех коэффициентов:

$$\mathrm{RSS} + \alpha \sum_{j=1}^{p} \hat{\beta}_j^2,$$

где $\hat{\beta}_j$ — коэффициент j количества признаков p; α — гиперпараметр (мы рассмотрим его далее). В *лассо-регрессии* штраф за уменьшение равен произведению оптимизируемого гиперпризнака и суммы абсолютных значений всех коэффициентов:

$$\frac{1}{2n}\mathrm{RSS} + \alpha \sum_{j=1}^{p} \left|\hat{\beta}_j\right|,$$

где n — количество наблюдений. Что же лучше использовать? С точки зрения эмпирических наблюдений, гребневая регрессия выдает немного более качественные прогнозы (в *рецепте 13.5* мы рассмотрим причины), чем лассо, однако с лассо-регрессией повышается интерпретируемость моделей. Для того чтобы добиться баланса между функциями штрафов гребневой регрессии и лассо-регрессии, мы можем использовать *эластичную сеть* (elastic net), которая включает оба регуляризатора. Оба варианта регрессии позволяют нам создавать ограничения для больших и сложных моделей путем включения значений коэффициентов в функцию потерь, которую мы пытаемся уменьшить.

Мы можем изменять величину ограничения коэффициентов при помощи гиперпараметра α, где при большем значении получаются более простые модели. Идеальное значение α должно быть оптимизировано, как любой другой гиперпараметр. Настройка α в scikit-learn происходит через параметр `alpha`.

Для выбора идеального значения α в scikit-learn можно использовать класс `RidgeCV`:

```
# Загрузить библиотеку
from sklearn.linear_model import RidgeCV

# Создать объект гребневой регрессии, указав три значения альфа
regr_cv = RidgeCV(alphas=[0.1, 1.0, 10.0])

# Настроить линейную регрессию
model_cv = regr_cv.fit(features_standardized, target)

# Вывести на экран коэффициенты
model_cv.coef_
array([1.29223201e-02, 4.40972291e+01, 5.38979372e+01])
```

Мы также можем легко просмотреть значение лучшей модели:

```
# Вывести значения альфа
model_cv.alpha_
```

```
0.1
```

Одно последнее замечание: поскольку в линейной регрессии значения коэффициентов частично определяются размером признака, а в моделях с регуляризацией происходит сложение коэффициентов, мы должны убедиться, что признак был стандартизирован до начала обучения.

13.5. Уменьшение дисперсии через лассо-регрессию

Задача

Вы хотите упростить модель линейной регрессии путем сокращения количества признаков.

Решение

Воспользуемся лассо-регрессией:

```
# Загрузить библиотеку
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_regression

# Сгенерировать матрицу признаков и целевой вектор
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   n_informative = 2,
                                   n_targets = 1,
                                   noise = 0.2,
                                   coef = False,
                                   random_state = 1)

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект лассо-регрессии, указав значение альфа
regression = Lasso(alpha=0.5)

# Настроить линейную регрессию
model = regression.fit(features_standardized, target)
```

Обсуждение

Одной из особенностей лассо-регрессии является возможность сокращения коэффициентов модели до нуля, в результате чего можно существенно уменьшить количество признаков модели. Так, в нашем примере указано значение `alpha=0.5`, и многие коэффициенты равны 0, т. е. в модели не используются соответствующие им признаки:

```
# Вывести коэффициенты на экран
model.coef_
array([-0.        , 43.58618393, 53.39523724])
```

Однако при значительном увеличении значения α мы можем видеть, что ни один признак не используется:

```
# Создать объект лассо-регрессии с высоким значением альфа
regression_a10 = Lasso(alpha=10)
model_a10 = regression_a10.fit(features_standardized, target)
model_a10.coef_
array([-0.        , 32.92181899, 42.73086731])
```

При помощи этого механизма мы можем, например, создать матрицу из 100 признаков и затем настроить гиперпараметр лассо-регрессии таким образом, чтобы в модели были использованы только 10 (или любое другое число) наиболее важных. Это позволяет уменьшить дисперсию, улучшив интерпретируемость нашей модели (поскольку небольшое количество признаков проще объяснить).

ГЛАВА 14

Деревья и леса

Введение

Алгоритмы обучения на основе деревьев решений представляют большую группу непараметрических контролируемых методов обучения, которые широко используются для классификации и регрессии. Они построены на *деревьях решений*, в которых правила принятия решений (например, "Если кредит доверия человека больше 720...") формируют последовательную цепь. Результат представляет собой перевернутое дерево, на вершине которого находится главное решающее правило (decision rule), а все последующие ветвятся вниз. В дереве решений каждое решающее правило находится внутри узла (decision node), ветви которого ведут к новым узлам. Если ветвь не имеет решающего правила в конце, она называется *листом* (leaf).

Одной из причин столь широкой популярности моделей на основе деревьев является их интерпретируемость. В сущности, при необходимости создания интуитивно понятной модели мы можем буквально нарисовать полное дерево решений (*см. рецепт 14.3*). Базовая концепция деревьев породила множество расширенных вариаций, от случайного леса (random forest) до стекинга (stacking). В этой главе мы узнаем, как обучать, управлять, настраивать, визуализировать и оценивать различные модели на основе деревьев.

14.1. Обучение классификатора на основе деревьев решений

Задача

Вам необходимо обучить классификатор при помощи дерева принятия решений.

Решение

Используем класс `DecisionTreeClassifier` библиотеки scikit-learn:

```
# Загрузить библиотеку
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
```

```
# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации методом деревьев решений
decisiontree = DecisionTreeClassifier(random_state=0)

# Обучить модель
model = decisiontree.fit(features, target)
```

Обсуждение

При обучении с использованием деревьев решений мы стремимся найти такое решающее правило, которое минимизирует примесь (impurity) узла. В классе `DecisionTreeClassifier` по умолчанию используется коэффициент Джини (примесь Джини — Gini impurity), однако существует множество других способов измерения качества:

$$G(t) = 1 - \sum_{i=1}^{c} p_i^2,$$

где $G(t)$ — значение коэффициента Джини узла t; p_i — доля наблюдений класса c в узле t. Этот процесс поиска решающих правил путем разделения множества для его очистки рекурсивно повторяется до полной очистки листьев узлов (например, когда они содержат значения одного класса) или достижения некоторого условия остановки алгоритма.

Принцип работы с классом `DecisionTreeClassifier` в scikit-learn не отличается от других методов обучения — мы обучаем модель при помощи метода `fit`, после чего ее можно использовать для прогнозирования классов наблюдений:

```
# Создать новое наблюдение
observation = [[ 5,  4,  3,  2]]

# Спрогнозировать класс наблюдения
model.predict(observation)
array([1])
```

Мы также можем просмотреть распределение вероятностей классов наблюдения:

```
# Вывести распределение вероятностей для трех классов
model.predict_proba(observation)
array([[0., 1., 0.]])
```

Наконец, при помощи параметра criterion мы можем изменить текущий способ измерения примеси:

```
# Создать объект классификации методом деревьев решений с использованием энтропии
decisiontree_entropy = DecisionTreeClassifier(
    criterion='entropy', random_state=0)

# Обучить модель
model_entropy = decisiontree_entropy.fit(features, target)
```

Дополнительные материалы для чтения

- Decision Tree Learning, Princeton ("Обучение методом деревьев решений", глава 3 учебника по курсу "Взаимодействие с данными" принстонского университета; **https://oreil.ly/lCPBG**).

14.2. Обучение регрессора с помощью деревьев решений

Задача

Вам нужно обучить модель регрессии при помощи дерева решений.

Решение

Воспользуемся классом DecisionTreeRegressor библиотеки scikit-learn:

```
# Загрузить библиотеку
from sklearn.tree import DecisionTreeRegressor
from sklearn import datasets

# Загрузить данные с двумя признаками
diabetes = datasets.load_diabetes()
features = diabetes.data
target = diabetes.target

# Создать объект регрессии с помощью деревьев решений
decisiontree = DecisionTreeRegressor(random_state=0)
```

```
# Обучить модель
model = decisiontree.fit(features, target)
```

Обсуждение

Принцип регрессии на основе деревьев решений схож с классификацией на основе деревьев, однако вместо уменьшения коэффициента Джини или энтропии при дроблении на части по умолчанию измеряется снижение среднеквадратической ошибки (mean squared error, MSE):

$$\text{MSE} = \frac{1}{n_i}\sum_{i=1}^{n}\left(y_i - \bar{y}_i\right)^2,$$

где y_i — истинное значение целевого показателя; $\bar{y}_i$ — среднее значение. Для использования регрессией на основе деревьев решений в scikit-learn есть класс `DecisionTreeRegressor`. После обучения модели на основе дерева решений мы можем использовать ее для предсказания наблюдений целевого вектора:

```
# Создать новое наблюдение
observation = [features[0]]

# Спрогнозировать значение наблюдения
model.predict(observation)
array([151.])
```

Как и в `DecisionTreeClassifier`, мы можем выбрать желаемый способ измерения качества при помощи параметра `criterion`. К примеру, мы можем использовать деревья, в которых при дроблении на части происходит сокращение величины средней абсолютной ошибки (mean absolute error, MAE):

```
# Создать объект классификации на основе деревьев при помощи MAE
decisiontree_mae = DecisionTreeRegressor(criterion="absolute_error",
   random_state=0)

# Обучить модель
model_mae = decisiontree_mae.fit(features, target)
```

Дополнительные материалы для чтения

- Decision Tree Regression ("Регрессия методом деревьев решений", документация scikit-learn; **https://oreil.ly/EGkU_**).

14.3. Визуализация модели деревьев решений

Задача

Вам нужно визуализировать модель, созданную с использованием алгоритма обучения на основе деревьев решений.

Решение

Преобразуем модель в формат DOT и затем визуализируем ее (рис. 14.1):

```
# Загрузить библиотеки
import pydotplus
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
from IPython.display import Image
from sklearn import tree

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации с помощью деревьев решений
decisiontree = DecisionTreeClassifier(random_state=0)

# Обучить модель
model = decisiontree.fit(features, target)
# Создать данные в формате DOT
dot_data = tree.export_graphviz(decisiontree,
                                out_file=None,
                                feature_names=iris.feature_names,
                                class_names=iris.target_names)

# Нарисовать граф
graph = pydotplus.graph_from_dot_data(dot_data)

# Вывести граф на экран
Image(graph.create_png())
```

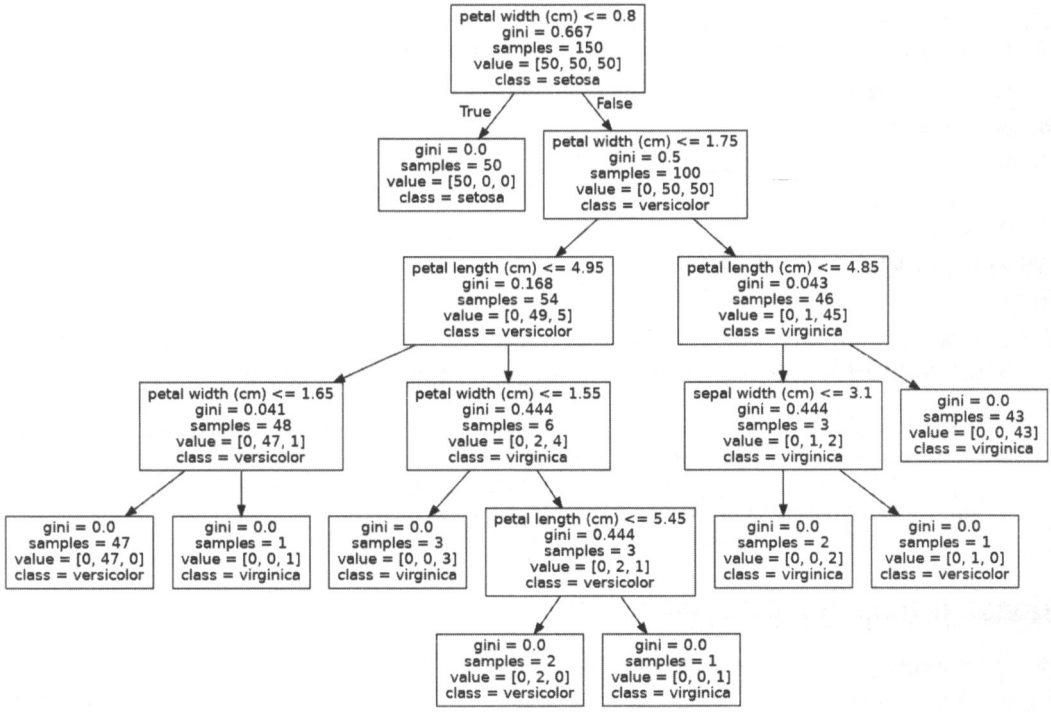

Рис. 14.1

Обсуждение

Весомым преимуществом классификаторов на основе деревьев является возможность визуализировать обученную модель, что позволяет создавать одни из самых легко интерпретируемых моделей машинного обучения. В нашем примере мы преобразовали формат обученной модели в DOT (graph description language — язык описания графов) и создали на ее основе граф.

Как мы можем наблюдать, решающее правило в корневом узле включает информацию о ширине лепестка (petal widths); если она меньше или равна 0,8 см, то мы двигаемся ниже по левой ветви, а если нет — по правой. Мы также можем видеть значение коэффициента Джини (0,667), количество наблюдений (150), количество наблюдений каждого класса ([50, 50, 50]) и класс, к которому текущее наблюдение будет отнесено на данном этапе (*setosa* — ирис щетинистый). Из этого следует, что на уровне этого узла алгоритму достаточно данных одного решающего правила (`petal width (cm) <= 0.8`) для безошибочного предсказания всех наблюдений класса *setosa*. Более того, добавление еще одного решающего правила для того же признака (`petal width (cm) <= 1.75`) позволяет дереву решений верно классифицировать 144 из 150 наблюдений. Это делает petal width (ширина лепестка) крайне важным признаком!

При необходимости визуализации дерево решений можно экспортировать в формат PDF или изображение PNG для использования в других приложениях и отчетах:

```
# Создать PDF-файл
graph.write_pdf("iris.pdf")
```

```
True
```

```
# Создать изображение PNG
graph.write_png("iris.png")
```

```
True
```

При помощи данного решения мы визуализировали классификатор на основе деревьев, при этом мы так же легко можем применить его для визуализации регрессора.

> **Примечание.** Пользователи macOS могут установить утилиту Graphviz для запуска представленного выше кода. Это можно сделать через менеджер пакетов Homebrew, используя команду `brew install graphviz`. Инструкции по установке пакетов можно найти на сайте Homebrew.

Дополнительные материалы для чтения

- Homebrew (**https://oreil.ly/GgeNI**).

14.4. Обучение классификатора на основе случайного леса

Задача

Вы хотите обучить классификатор при помощи леса случайных деревьев решений.

Решение

Воспользуемся классом `RandomForestClassifier` библиотеки scikit-learn для обучения модели классификации методом случайного леса.

```
# Загрузить библиотеку
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации методом случайного леса
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)
```

```
# Обучить модель
model = randomforest.fit(features, target)
```

Обсуждение

При использовании деревьев решений мы часто сталкиваемся с проблемой переобучения (overfitting), когда модель слишком сильно привязывается к образцам обучающего набора, в некоторых случаях запоминая огромное количество примеров. Это способствовало широкому распространению метода ансамблевого обучения, называемого *методом случайного леса* (random forest). Алгоритм случайного леса включает обучение множества деревьев решений, при этом наблюдения формируются методом бутстрэпа (т. е. генерируется соразмерная простая подвыборка случайных признаков из обучающей выборки), а для каждого узла выбирается определенный набор признаков. Каждое случайное дерево решений этого леса (отсюда название) определяет класс предсказываемого объекта; решение выносится на основании большинства "голосов".

При сравнении этого решения с *рецептом 14.1* мы видим, что классы `RandomForestClassifier` и `DecisionTreeClassifier` библиотеки scikit-learn работают схожим образом:

```
# Создать новое наблюдение
observation = [[ 5,  4,  3,  2]]

# Предсказать класс наблюдения
model.predict(observation)
array([1])
```

В классе `RandomForestClassifier` также представлено множество параметров `DecisionTreeClassifier`. К примеру, мы можем изменить способ измерения качества частей (split):

```
# Создать объект классификации методом случайного леса с использованием энтропии
randomforest_entropy = RandomForestClassifier(
    criterion="entropy", random_state=0)

# Обучить модель
model_entropy = randomforest_entropy.fit(features, target)
```

Вместе с тем класс `RandomForestClassifier` имеет ряд уникальных и очень важных параметров, поскольку представляет не отдельное дерево решений, а собранное в лес множество. Во-первых, это параметр `max_features`, который определяет максимальное количество признаков, используемых каждым узлом, и принимает различные аргументы, включая целые числа (количество признаков), числа с плавающей запятой (соотношение признаков) и `sqrt` (square root of the number of features) — квадратный корень числа функций. Параметр `max_features` по умолчанию имеет значение `auto`, которое функционально схоже с `sqrt`. Во-вторых, это параметр `bootstrap`, при помощи которого мы можем настроить генерацию подвыборки случайных признаков из обучающей выборки для дерева, с заменой (по умолчанию)

или без замены. В-третьих, это параметр `n_estimators`, который задает количество деревьев решений, включаемых в лес. Наконец, это параметр `n_jobs=-1`, который не является уникальным для классификаторов на основе случайного леса, однако позволяет получить прирост за счет использования всех ядер процессора для одновременного обучения нескольких моделей.

Дополнительные материалы для чтения

- Random Forests, Berkeley Statistics ("Метод случайного леса"; **https://oreil.ly/h-LQL**).

14.5. Обучение регрессора методом случайного леса

Задача

Вы хотите обучить модель регрессии при помощи леса случайных деревьев решений.

Решение

Воспользуемся классом `RandomForestRegressor` библиотеки scikit-learn для обучения модели регрессии методом случайного леса:

```
# Загрузить библиотеку
from sklearn.ensemble import RandomForestRegressor
from sklearn import datasets

# Загрузить данные с двумя признаками
diabetes = datasets.load_diabetes()
features = diabetes.data
target = diabetes.target

# Создать объект регрессии методом деревьев решений
randomforest = RandomForestRegressor(random_state=0, n_jobs=-1)

# Обучить модель
model = randomforest.fit(features, target)
```

Обсуждение

Помимо классификатора на основе деревьев решений мы можем схожим образом создать регрессор, в котором каждое дерево использует бутстрэп-подвыборку наблюдений, а в каждом узле используется определенный набор признаков. Как и в случае `RandomForestClassifier`, нужно выделить несколько важных параметров.

`max_features`

Определяет максимальное количество признаков, используемых каждым узлом. По умолчанию имеет p признаков, где p — общее количество признаков.

`bootstrap`

Определяет наличие или отсутствие замены при генерации подвыборки. По умолчанию `True`.

`n_estimators`

Определяет количество используемых деревьев решений. По умолчанию 10.

Дополнительные материалы для чтения

- Документация по `RandomForestRegressor` (**https://oreil.ly/ksa9Z**).

14.6. Оценка моделей случайного леса методом out-of-bag-ошибки

Задача

Вам необходимо оценить модель случайного леса, избегая кросс-валидации.

Решение

Рассчитаем показатель out-of-bag нашей модели:

```
# Загрузить библиотеку
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации методом случайного леса
randomforest = RandomForestClassifier(
    random_state=0, n_estimators=1000, oob_score=True, n_jobs=-1)

# Обучить модель
model = randomforest.fit(features, target)

# Вывести значение out-of-bag-ошибки
randomforest.oob_score_
```

```
0.9533333333333334
```

Обсуждение

При обучении методом случайного леса каждое дерево решений использует бутстрэп-подвыборку наблюдений. Это значит, что для каждого дерева существует отдельное подмножество наблюдений, которые не используются для обучения этого дерева. Они называются наблюдениями вне пакета (выборки), или OOB (out-of-bag) наблюдениями. Мы можем использовать такие наблюдения в качестве тестового набора для оценки качества нашей модели случайного леса.

Обучающий алгоритм сравнивает истинное значение каждого наблюдения с прогнозом, полученным от подмножества деревьев, при обучении которых это наблюдение не использовалось. Затем рассчитывается итоговый показатель, который выступает мерой качества модели случайного леса. Оценка показателя вне пакета (OOB) используется вместо кросс-валидации.

В scikit-learn расчет показателя вне пакета (OOB) происходит при помощи параметра `oob_score=True` в объекте случайного леса (т. е. `RandomForestClassifier`). Атрибут `oob_score_` возвращает значение показателя.

14.7. Определение ключевых признаков в моделях случайного леса

Задача

Вам нужно знать, какие признаки вашей модели случайного леса являются наиболее важными.

Решение

Воспользуемся атрибутом модели `eature_importances_` для вычисления и визуализации показателей значимости каждого признака:

```
# Загрузить библиотеки
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации методом случайного леса
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)
```

```
# Обучить модель
model = randomforest.fit(features, target)

# Вычислить значимость признака
importances = model.feature_importances_

# Отсортировать признаки по убыванию значимости
indices = np.argsort(importances)[::-1]

# Перегруппировать названия признаков в соответствии с их значимостью
names = [iris.feature_names[i] for i in indices]

# Создать диаграмму
plt.figure()

# Создать заголовок диаграммы
plt.title("Значимость признака")

# Добавить столбцы
plt.bar(range(features.shape[1]), importances[indices])

# Добавить названия признаков на оси x
plt.xticks(range(features.shape[1]), names, rotation=90)

# Вывести диаграмму
plt.show()
```

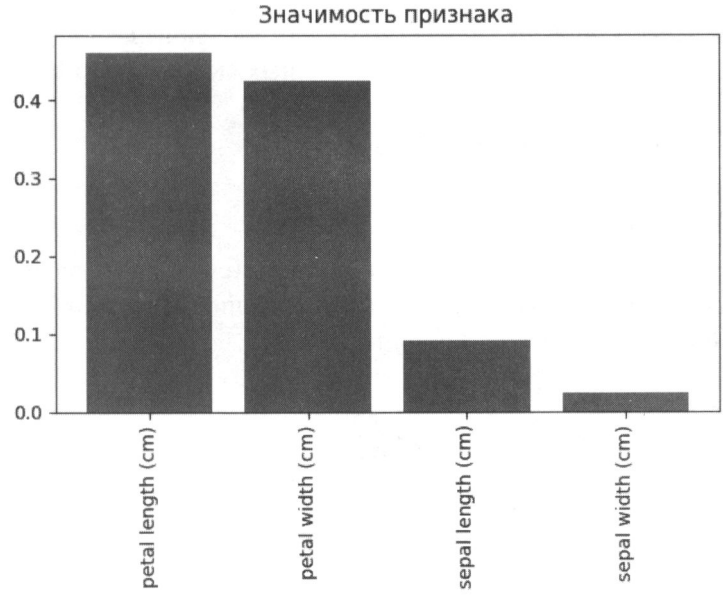

Рис. 14.2

Обсуждение

Одним из главных преимуществ деревьев решений является интерпретируемость. В частности, мы можем визуализировать модель целиком (*см. рецепт 14.3*). С другой стороны, модель случайного леса может включать десятки, сотни и даже тысячи деревьев решений. В такой ситуация простота и интуитивность визуализации модели случайного леса теряют всякий смысл. Однако существует другой способ: мы можем сравнить (и визуализировать) относительную значимость каждого признака.

В *рецепте 14.3* мы визуализировали модель классификации методом деревьев решений и увидели, что решающие правила с признаком ширины лепестка в основе могут верно классифицировать большое количество наблюдений. Интуитивно мы можем заключить, что значение ширины лепестка имеет бóльшую значимость для нашего классификатора. Формально наибольшую значимость имеют признаки, части которых при дроблении имеют наибольший показатель снижения примеси (например, значение коэффициента Джини или энтропия классификатора и дисперсия регрессора).

При оценке значимости признаков необходимо учитывать две вещи. Во-первых, при использовании библиотеки scikit-learn мы должны разделять номинальные категориальные признаки на множество бинарных признаков. Это приводит к распределению важности между всеми бинарными признаками и возможному снижению их значимости, даже если номинальный категориальный признак содержал важную информацию. Во-вторых, если два признака обладают высокой связностью, один будет признан более значимым, а важность другого, соответственно, сильно снизится, что скажется на интерпретируемости.

В scikit-learn мы можем проверить значимость каждого признака классификации и регрессии методом деревьев решений и случайных лесов при помощи атрибута `feature_importances_`:

```
# Вывести на экран значения значимости признака
model.feature_importances_
```

```
array([0.09090795, 0.02453104, 0.46044474, 0.42411627])
```

Чем выше значение показателя, тем важнее признак (сумма всех значений должна равняться 1). Использование графика на основе значений позволяет сделать модели на основе случайного леса более интерпретируемыми.

14.8. Выбор ключевых признаков в моделях случайного леса

Задача

Вам нужно произвести отбор признаков модели на основе случайного леса.

Решение

Определим наиболее значимые признаки и на их основе заново обучим модель:

```
# Загрузить библиотеку
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
from sklearn.feature_selection import SelectFromModel

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации методом случайного леса
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)

# Создать объект отбора признаков с показателем значимости выше
# или равным пороговому значению
selector = SelectFromModel(randomforest, threshold=0.3)

# При помощи объекта отбора создать новую матрицу признаков
features_important = selector.fit_transform(features, target)

# При помощи отобранных признаков обучить модель случайного леса
model = randomforest.fit(features_important, target)
```

Обсуждение

Бывает так, что при работе с некоторыми моделями нам нужно сократить количество признаков. Например, нам необходимо уменьшить дисперсию модели или улучшить интерпретируемость за счет использования только наиболее важных признаков.

В scikit-learn мы можем сократить число признаков модели в два этапа. Сначала происходит обучение модели случайного леса с использованием всех признаков. Затем мы определяем наиболее значимые признаки этой модели. Далее мы создаем новую матрицу, в которую включаем только самые важные признаки. В нашем примере мы воспользовались классом `SelectFromModel` для создания новой матрицы признаков со значениями, равными или выше заданных порогов. Наконец, мы создали новую модель с использованием только этих признаков.

Нам следует учесть две особенности такого подхода. Во-первых, при унитарном кодировании номинальных категориальных признаков значимость признака размывается из-за распределения между бинарными признаками. Во-вторых, при тесной

корреляции признаков показатель повышенной значимости распространяется только на один из них.

Дополнительные материалы для чтения

- Variable Selection Using Random Forests, Robin Genuer, Jean-Michel Poggi, and Christine Tuleau-Malot ("Отбор переменных методом случайного леса"; **https://oreil.ly/y9k2U**).

14.9. Управление несбалансированными классами

Задача

У вас есть целевой вектор с несбалансированными классами, и вы хотите обучить модель методом случайного леса.

Решение

Обучим модель на основе дерева решений или случайного леса с использованием параметра `class_weight="balanced"`:

```
# Загрузить библиотеки
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Нарушить баланс, удалив 40 первых наблюдений класса
features = features[40:,:]
target = target[40:]

# Создать целевой вектор, где класс равен 0, остальное равно 1
target = np.where((target == 0), 0, 1)

# Создать объект классификации методом случайного леса
randomforest = RandomForestClassifier(
    random_state=0, n_jobs=-1, class_weight="balanced")

# Обучить модель
model = randomforest.fit(features, target)
```

Обсуждение

Мы часто сталкиваемся с нарушением баланса классов при использовании реальных данных в машинном обучении. Присутствие несбалансированных классов может снизить качество нашей модели. Мы более подробно разберем алгоритмы работы с такими классами на этапе предобработки в *рецепте 17.5*. Многие алгоритмы машинного обучения в scikit-learn имеют встроенные методы исправления дисбаланса классов. Например, можем использовать `RandomForestClassifier` с параметром `class_weight`. При наличии словаря с названиями классов и нужными весами (например, `{"male": 0.2, "female": 0.8}`) `RandomForestClassifier` взвешивает классы равнозначно. В качестве более удобной альтернативы можно использовать аргумент `balanced`, который автоматически взвешивает классы пропорционально частоте их появления в наборе данных:

$$w_j = \frac{n}{kn_j},$$

где w_j — вес класса j; n — количество наблюдений; n_j — количество наблюдений класса j; k — общее количество классов. Например, в нашем решении есть 2 класса (k), 110 наблюдений (n) и 10 и 100 наблюдений каждого класса соответственно (n_j). При использовании параметра `class_weight="balanced"` меньший класс будет иметь больший вес:

```
# Измерить вес меньшего класса
110/(2*10)
```

```
5.5
```

тогда как больший класс имеет меньший вес:

```
# Измерить вес большего класса
110/(2*100)
```

```
0.55
```

14.10. Контроль размера дерева

Задача

Вы хотите вручную определить структуру и размер дерева решений.

Решение

Используем алгоритмы машинного обучения на основе деревьев в scikit-learn, задав параметры структуры дерева:

```
# Загрузить библиотеку
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
```

```
# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации с помощью деревьев решений
decisiontree = DecisionTreeClassifier(random_state=0,
                                      max_depth=None,
                                      min_samples_split=2,
                                      min_samples_leaf=1,
                                      min_weight_fraction_leaf=0,
                                      max_leaf_nodes=None,
                                      min_impurity_decrease=0)

# Обучить модель
model = decisiontree.fit(features, target)
```

Обсуждение

Алгоритмы машинного обучения на основе деревьев в scikit-learn включают инструменты управления размером деревьев решений. Они регулируются следующими параметрами.

`max_depth`

Максимальная глубина дерева. При значении `None` дерево будет увеличиваться до очистки всех листьев. При целочисленном значении дерево "подрезается" в соответствии с ним.

`min_samples_split`

Минимальное количество наблюдений узла до его дробления. Целочисленное значение в качестве аргумента означает простой показатель, тогда как число с плавающей запятой означает долю от всех наблюдений.

`min_samples_leaf`

Минимальное количество наблюдений для формирования листа. Принимает аргументы вроде `min_samples_split`.

`max_leaf_nodes`

Минимальное количество листьев.

`min_impurity_split`

Минимальный допустимый показатель снижения примеси перед дроблением.

Несмотря на важность всех параметров, нам скорее всего понадобятся только `max_depth` и `min_impurity_split`, поскольку небольшие деревья (так называемые *пни* (stumps)) являются простыми моделями с низкой дисперсией.

14.11. Улучшение качества через бустинг

Задача

Вам недостаточно деревьев решений и случайного леса, и необходимо создать более эффективную модель.

Решение

Для обучения модели при помощи бустинга можно использовать классы `AdaBoostClassifier` или `AdaBoostRegressor`:

```
# Загрузить библиотеку
from sklearn.ensemble import AdaBoostClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации методом деревьев с алгоритмом AdaBoost
adaboost = AdaBoostClassifier(random_state=0)

# Обучить модель
model = adaboost.fit(features, target)
```

Обсуждение

При использовании метода случайного леса предсказание целевого вектора осуществляется ансамблем (группой) деревьев случайных решений. В качестве альтернативного, зачастую более эффективного метода можно использовать так называемый *бустинг* (boosting). Одной из разновидностей бустинга является алгоритм AdaBoost, который предполагает цикличное обучение слабых моделей (часто это небольшие деревья решений, иногда называемые пнями), где при каждой последующей итерации приоритет получают признаки, неверно классифицированные на предыдущем этапе. При использовании AdaBoost, в частности, необходимо:

1. Присвоить каждому наблюдению x_i изначальное значение веса $w_i = 1/n$, где n является общим количеством наблюдений в наборе данных.
2. Обучить "слабую" модель на основе данных.

3. Учесть, что для каждого наблюдения:
   - если слабая модель правильно предсказывает x_i, значение w_i уменьшается;
   - если слабая модель неправильно предсказывает x_i, значение w_i повышается.
4. Обучить новую слабую модель, в которой предпочтение отдается наблюдениям с большим значением w_i.
5. Повторить шаги 4 и 5 до получения идеальных предсказаний или обучения имеющихся слабых моделей.

В результате получается агрегированная модель, в которой отдельные слабые модели обрабатывают сложные (с точки зрения прогнозирования) наблюдения. В scikit-learn мы можем использовать алгоритм AdaBoost при помощи классов `AdaBoostClassifier` или `AdaBoostRegressor`. К наиболее важным параметрам относятся `base_estimator`, `n_estimators`, `learning_rate` и `loss`.

`base_estimator`

Параметр представляет собой обучающий алгоритм для тренировки слабых моделей. В качестве значения по умолчанию указано дерево решений — наиболее распространенный метод обучения с AdaBoost.

`n_estimators`

Параметр указывает на количество моделей для цикличного обучения.

`learning_rate`

Параметр означает вклад каждой модели в веса; по умолчанию имеет значение `1`. Снижение коэффициента скорости обучения (learning rate) означает, что увеличение или уменьшение весов при коррекции будет происходить с меньшим шагом, в результате чего модель начинает обучаться медленнее (в некоторых случаях повышается эффективность модели).

`loss`

Параметр представлен только в классе `AdaBoostRegressor` и добавляет возможность использования функции потерь при обновлении весов. По умолчанию выбрана линейная функция потерь, однако ее можно заменить на `square` или `exponential`.

Дополнительные материалы для чтения

- Robert E. Schapire "Explaining AdaBoost" (Роберт Э. Шапир "Объяснение функции AdaBoost"; **https://oreil.ly/5E1v4**).

14.12. Обучение модели на основе алгоритма XGBoost

Задача

Вам необходимо обучить модель на основе деревьев с улучшенным прогнозированием.

Решение

Воспользуемся поддерживаемой Python библиотекой xgboost:

```
# Загрузить библиотеки
import xgboost as xgb
from sklearn import datasets, preprocessing
from sklearn.metrics import classification_report
from numpy import argmax

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать набор данных
xgb_train = xgb.DMatrix(features, label=target)

# Определить параметры
param = {
    'objective': 'multi:softprob',
    'num_class': 3
}

# Обучить модель
gbm = xgb.train(param, xgb_train)

# Получить прогнозы
predictions = argmax(gbm.predict(xgb_train), axis=1)

# Создать отчет о классификации
print(classification_report(target, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      0.96      0.98        50
           2       0.96      1.00      0.98        50

    accuracy                           0.99       150
   macro avg       0.99      0.99      0.99       150
weighted avg       0.99      0.99      0.99       150
```

Обсуждение

XGBoost (Extreme Gradient Boosting) — это популярная реализация алгоритма градиентного бустинга, используемая в машинном обучении. Хотя это не всегда модель на основе деревьев, она часто применяется при создании ансамблей деревьев решений. Данный алгоритм получил широкое распространение благодаря отличным результатам, продемонстрированным на соревнованиях по машинному обучению от веб-ресурса Kaggle, и остается надежным решением, превосходящим по производительности решения на основе обычных случайных лесов или градиентного бустинга.

Считается, что алгоритм XGBoost требует больших вычислительных ресурсов, однако за последние несколько лет была проделана большая работа по оптимизации вычислений (например, поддержка графических ускорителей), что позволило существенно ускорить процессы итераций; в результате XGBoost часто используют, когда нужна статистическая эффективность.

Дополнительные материалы для чтения

- Документация XGBoost (**https://oreil.ly/cAuGX**).

14.13. Улучшение производительности при помощи LightGBM

Задача

Вам нужно обучить модель на основе градиентного бустинга с оптимизацией вычислений.

Решение

Используем библиотеку машинного обучения на основе градиентного бустинга lightgbm:

```
# Загрузить библиотеки
import lightgbm as lgb
from sklearn import datasets, preprocessing
from sklearn.metrics import classification_report
from numpy import argmax

# Загрузить данные
iris = datasets.load_iris()
```

```
features = iris.data
target = iris.target

# Создать набор данных
lgb_train = lgb.Dataset(features, target)

# Определить параметры
params = {
    'objective': 'multiclass',
    'num_class': 3,
    'verbose': -1,
}

# Обучить модель
gbm = lgb.train(params, lgb_train)

# Получить прогнозы
predictions = argmax(gbm.predict(features), axis=1)

# Создать отчет о классификации
print(classification_report(target, predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        50
           2       1.00      1.00      1.00        50

    accuracy                           1.00       150
   macro avg       1.00      1.00      1.00       150
weighted avg       1.00      1.00      1.00       150
```

Обсуждение

Библиотека lightgbm используется в машинном обучении с градиентным бустингом. Она имеет оптимизацию времени обучения, логических заключений, а также поддерживает использование графических ускорителей. Благодаря высокой производительности данная библиотека широко используется в промышленной разработке и масштабных системах. Несмотря на простоту большинства моделей scikit-learn, некоторые библиотеки, например lightgbm, могут оказаться более эффективными при работе с большими объемами данных или наличии жестких временны́х ограничений на тренировку и выпуск.

Дополнительные материалы для чтения

- Документация LightGBM (**https://oreil.ly/XDcpG**).
- Документация CatBoost (еще одна оптимизированная библиотека для GBM; **https://oreil.ly/4Bb8g**).

ГЛАВА 15

Метод *k* ближайших соседей

Введение

Классификация методом *k* ближайших соседей (k-nearest neighbors, KNN) является одним из самых простых и распространенных инструментов контролируемого машинного обучения (supervised machine learning). Метод *k* ближайших соседей относится к так называемому ленивому обучению, в котором отсутствует этап обучения модели для прогнозирования. Вместо этого наблюдение наследует наиболее многочисленный класс среди *k* ближайших наблюдений.

К примеру, если наблюдение с неизвестным классом окружают наблюдения с классом 1, то оно также будет отнесено к классу 1. В этой главе мы научимся использовать инструменты scikit-learn для создания и осуществления классификации методом *k* ближайших соседей.

15.1. Определение ближайших соседей для наблюдения

Задача

Вам необходимо найти *k* ближайших наблюдений (соседей) целевого наблюдения.

Решение

Воспользуемся классом `NearestNeighbors` библиотеки scikit-learn:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
```

```
# Создать объект стандартизации
standardizer = StandardScaler()

# Стандартизировать признаки
features_standardized = standardizer.fit_transform(features)

# Указать число ближайших соседей, равное 2
nearest_neighbors = NearestNeighbors(n_neighbors=2).fit(features_standardized)

# Создать наблюдение
new_observation = [ 1,  1,  1,  1]

# Узнать удаленность и индексы ближайших соседей наблюдения
distances, indices = nearest_neighbors.kneighbors([new_observation])

# Вывести на экран ближайших соседей
features_standardized[indices]
array([[[1.03800476, 0.55861082, 1.10378283, 1.18556721],
        [0.79566902, 0.32841405, 0.76275827, 1.05393502]]])
```

Обсуждение

В данном примере мы использовали набор данных ирисов. Мы создали наблюдение `new_observation` с некоторыми значениями и затем нашли два ближайших наблюдения. Переменная `indices` содержит информацию о расположении ближайших наблюдений в наборе данных, значения которых можно отобразить при помощи `X[indices]`. Расстояние можно интуитивно представить как меру схожести; в таком случае двумя ближайшими наблюдениями будут цветы, наиболее похожие на созданный нами вариант.

Как же мы измеряем расстояние? В scikit-learn есть большой спектр различных метрик измерения расстояния d, включая евклидову метрику (Euclidean):

$$d_{Euclidean} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

и манхэттенское расстояние (Manhattan distance):

$$d_{Manhattan} = \sum_{i=1}^{n}|x_i - y_i|.$$

По умолчанию в классе `NearestNeighbors` используется расстояние Минковского (Minkowski distance):

$$d_{Minkowski} = \left(\sum_{i=1}^{n}|x_i - y_i|^p\right)^{1/p},$$

где x_i и y_i — два наблюдения, между которыми мы рассчитываем расстояние. Метрика Минковского включает гиперпараметр p, где $p = 1$ является манхэттенским расстоянием и $p = 2$ является евклидовым расстоянием и т. д. В scikit-learn по умолчанию $p = 2$.

Мы можем при помощи параметра `metric` указать нужную метрику измерения расстояния:

```
# Найти два ближайших соседних наблюдения через евклидово расстояние
nearestneighbors_euclidean = NearestNeighbors(
    n_neighbors=2, metric='euclidean').fit(features_standardized)
```

Созданная нами переменная `distance` содержит информацию о расстоянии до каждого из двух ближайших соседей:

```
# Вывести значения расстояния
distances
array([[0.49140089, 0.74294782]])
```

Кроме того, мы можем использовать `kneighbors_graph` для создания матрицы, в которой будут показаны все ближайшие соседи каждого наблюдения:

```
# Найти троих ближайших соседей для каждого наблюдения
# через евклидово расстояние (включая наблюдение)
nearestneighbors_euclidean = NearestNeighbors(
    n_neighbors=3, metric="euclidean").fit(features_standardized)

# Список списков с отображением трех ближайших соседей для каждого наблюдения
# (включая само наблюдение)
nearest_neighbors_with_self = nearestneighbors_euclidean.kneighbors_graph(
    features_standardized).toarray()

# Удалить единицы, помечающие наблюдение, как соседние себе
for i, x in enumerate(nearest_neighbors_with_self):
    x[i] = 0

# Вывести двоих ближайших соседей для первого наблюдения
nearest_neighbors_with_self[0]
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

В процессе поиска ближайших соседей или использования алгоритмов обучения на основе расстояния необходимо преобразовать признаки, чтобы они принадлежали одной единице измерения. Это связано с тем, что метрики расстояния воспринимают признаки как принадлежащие одной категории, соответственно, совмещение признаков в долларах и процентах приведет к ошибочным вычислениям расстояния второго признака. Для того чтобы предотвратить возможное появление этой ошибки в нашем решении, мы стандартизировали признаки при помощи класса `StandardScaler`.

15.2. Создание классификатора на основе *k* ближайших соседей

Задача

Вам нужно предсказать класс неизвестного наблюдения на основе класса соседних элементов.

Решение

Если набор данных не очень большой, можно использовать класс `KNeighborsClassifier`:

```
# Загрузить библиотеку
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Создать объект стандартизации
standardizer = StandardScaler()

# Стандартизировать признаки
X_std = standardizer.fit_transform(X)

# Обучить классификатор KNN, используя 5 соседних элементов
knn = KNeighborsClassifier(n_neighbors=5, n_jobs=-1).fit(X_std, y)

# Создать два наблюдения
new_observations = [[ 0.75,  0.75,  0.75,  0.75],
                    [ 1,  1,  1,  1]]
```

```
# Предсказать класс двух наблюдений
knn.predict(new_observations)
array([1, 2])
```

Обсуждение

Предположим, что у нас есть наблюдение x_u с неизвестным классом. Сначала алгоритм k ближайших соседей (KNN) определяет k ближайших элементов (иногда называемых *окрестностью* x_u) в соответствии с заданной метрикой расстояния (например, евклидова расстояния), затем k наблюдения "голосуют" за представляемый ими класс, и в результате объект x_u присваивается наиболее многочисленному из предполагаемых классов. Формально вероятность присвоения наблюдению x_u некоторого класса j можно выразить формулой:

$$\frac{1}{k}\sum_{i \in v} I(y_i = j),$$

где v — k-е наблюдение из ближайших к x_u; y_i — класс наблюдения i; I — индикатор (т. е. 1 — истина, 0 — наоборот). Для того чтобы узнать эту вероятность, мы можем использовать метод `predict_proba` из scikit-learn:

```
# Узнать вероятность принадлежности каждого наблюдения одному из трех классов
knn.predict_proba(new_observations)
array([[0. , 0.6, 0.4],
       [0. , 0. , 1. ]])
```

Наблюдение присваивается классу с наибольшим значением вероятности. На основе вывода в примере выше первое наблюдение должно быть отнесено к классу 1 (Pr = 0.6), а второе — к классу 2 (Pr = 1), в чем мы можем легко убедиться:

```
knn.predict(new_observations)
array([1, 2])
```

Класс `KNeighborsClassifier` содержит несколько важных параметров, которые следует упомянуть. Во-первых, параметр `metric` определяет используемую метрику расстояния. Во-вторых, параметр `n_jobs` означает количество используемых ядер процессора. Поскольку для создания прогнозов необходимо рассчитывать удаленность от точки до каждого элемента данных, рекомендуется использовать несколько ядер. В-третьих, параметр `algorithm` означает метод, используемый для определения ближайших соседей. Алгоритмы могут сильно различаться, однако нам во многих случаях не придется выбирать их вручную, поскольку `KNeighborsClassifier` по умолчанию самостоятельно подбирает наиболее оптимальный алгоритм. В-четвертых, по умолчанию `KNeighborsClassifier` учитывает класс каждого наблюдения, как мы описали выше, однако если мы укажем значение `distance` для параметра `weights`, то сможем отдать предпочтение ближайшим наблюдениям, снизив значимость более удаленных. Это кажется логичным, поскольку схожие наблюдения могут сообщить нам больше о классе целевого наблюдения.

Наконец, нам нужно стандартизировать признаки перед использованием KNN-классификатора, т. к. при расчете расстояния предполагается, что все признаки имеют одинаковую единицу измерения.

15.3. Определение оптимального количества ближайших соседей

Задача

Вы хотите подобрать значение k для классификатора на основе метода k ближайших соседей.

Решение

Воспользуемся методами выбора модели, например классом `GridSearchCV`:

```
# Загрузить библиотеку
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект стандартизации
standardizer = StandardScaler()

# Создать классификатор KNN
knn = KNeighborsClassifier(n_neighbors=5, n_jobs=-1)

# Создать конвейер
pipe = Pipeline([("standardizer", standardizer), ("knn", knn)])

# Создать пространство возможных значений
search_space = [{"knn__n_neighbors": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}]

# Организовать поиск по сетке
classifier = GridSearchCV(
    pipe, search_space, cv=5, verbose=0).fit(features_standardized, target)
```

Обсуждение

Величина значения k непосредственно влияет на классификатор KNN. В процессе машинного обучения мы стремимся найти баланс между смещением (bias) и дисперсией (variance). Иногда для этого можно использовать значение k. Если $k = n$, где n является количеством наблюдений, то мы получаем большое смещение при низкой дисперсии. Если $k = 1$, мы получаем небольшое смещение при высокой дисперсии. Для того чтобы получить наиболее качественную модель, нам нужно подобрать такое значение k, которое уравновесит смещение и дисперсию. В примере выше мы при помощи объекта класса `GridSearchCV` создали 5-блочную кросс-валидацию и применили ее к классификатору KNN с разными значениями k. В результате мы получили значение k, которое позволяет создать наиболее качественную модель:

```
# Лучшее значение области ближайших соседей (k)
classifier.best_estimator_.get_params()["knn__n_neighbors"]
```

```
6
```

15.4. Создание классификатора методом ближайших соседей в заданном радиусе

Задача

Дано наблюдение неизвестного класса, для которого нужно спрогнозировать класс на основе наблюдений в пределах некоторого расстояния.

Решение

Используем класс `RadiusNeighborsClassifier`:

```
# Загрузить библиотеку
from sklearn.neighbors import RadiusNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект стандартизации
standardizer = StandardScaler()

# Стандартизировать признаки
features_standardized = standardizer.fit_transform(features)
```

```
# Обучить классификатор на основе удаленности соседей
rnn = RadiusNeighborsClassifier(
    radius=.5, n_jobs=-1).fit(features_standardized, target)

# Создать два наблюдения
new_observations = [[ 1,  1,  1,  1]]

# Предсказать класс двух наблюдений
rnn.predict(new_observations)
array([2])
```

Обсуждение

При использовании классификации KNN прогнозирование класса наблюдения основывается на классах его k соседей. В качестве другого, менее распространенного способа можно использовать классификацию *методом ближайших соседей в заданном радиусе* (radius-based nearest neighbor, RNN), при которой класс наблюдения предугадывается на основе классов всех наблюдений в заданном радиусе r.

В scikit-learn класс `RadiusNeighborsClassifier`, за исключением двух параметров, идентичен `KNeighborsClassifier`. При помощи первого параметра, `radius`, в `RadiusNeighborsClassifier` мы указываем радиус области, в пределах которой все наблюдения считаются соседними. Данное значение рекомендуется рассматривать как другие гиперпараметры и настраивать на этапе выбора модели, если только у вас не возникает необходимость указать какое-либо значение. При помощи второго параметра, `outlier_label`, мы можем задать метку, которая будет присваиваться наблюдению, не имеющему соседей в указанном радиусе, что может помочь при поиске выбросов.

15.5. Поиск приближенных ближайших соседей

Задача

Вы хотите применить метод ближайших соседей к большим данным, сохранив низкую задержку.

Решение

Используем библиотеку Faiss компании Facebook[1] для поиска на основе *метода приближенных ближайших соседей* (approximate nearest neighbors, ANN):

[1] Деятельность этой социальной сети запрещена на территории Российской Федерации. — *Прим. ред.*

```
# Загрузить библиотеки
import faiss
import numpy as np
from sklearn import datasets
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Создать объект стандартизации
standardizer = StandardScaler()

# Стандартизировать признаки
features_standardized = standardizer.fit_transform(features)

# Настроить параметры faiss
n_features = features_standardized.shape[1]
nlist = 3
k = 2

# Создать индекс IVF
quantizer = faiss.IndexFlatIP(n_features)
index = faiss.IndexIVFFlat(quantizer, n_features, nlist)

# Обучить индекс и добавить векторы признаков
index.train(features_standardized)
index.add(features_standardized)

# Создать наблюдение
new_observation = np.array([[ 1,  1,  1,  1]])

# Осуществить поиск двух ближайших соседей внутри индекса
distances, indices = index.search(new_observation, k)

# Вывести векторы признаков двух ближайших соседей
np.array([list(features_standardized[i]) for i in indices[0]])
array([[1.03800476, 0.55861082, 1.10378283, 1.18556721],
       [0.79566902, 0.32841405, 0.76275827, 1.05393502]])
```

Обсуждение

Метод *k* ближайших соседей (KNN) является прекрасным инструментом поиска наилучших совпадений наблюдений при работе с небольшим набором данных. Правда, по мере роста объемов данных увеличивается время расчета удаленности наблюдения от остальных точек в представленном наборе. В больших системах на основе машинного обучения, например поисковых или рекомендательных системах, как правило, используется специальная форма измерения схожести векторов для поиска похожих наблюдений. Однако при работе с масштабными системами в реальном времени с необходимостью выдавать готовый результат менее чем за 100 миллисекунд метод *k* ближайших соседей невозможно использовать.

Метод приближенных ближайших соседей (ANN) позволяет обойти это ограничение, в некоторой степени жертвуя качеством поиска точных ближайших соседей в пользу быстродействия. Стоит отметить, что хотя при поиске методом ANN элементы первых 10 ближайших соседей или их порядок иногда могут не совпадать с результатами поиска методом *k* ближайших соседей, мы получаем их намного быстрее.

В нашем примере мы используем подход приближенных ближайших соседей, называемый инвертированным индексом файлов (inverted file index, IVF). Этот подход предполагает применение кластеризации для сокращения области поиска при использовании метода ближайших соседей. Наша область поиска делится на несколько разных участков (или кластеров) по принципу диаграммы Вороного. В процессе поиска ближайших соседей мы обращаемся к ограниченному множеству кластеров для обнаружения похожих наблюдений, а не сравниваем все точки нашего набора данных.

Воспользуемся примером с простым набором данных для визуализации процесса создания диаграммы Вороного. Возьмем график рассеяния, содержащий случайные данные в двух измерениях (рис. 15.1).

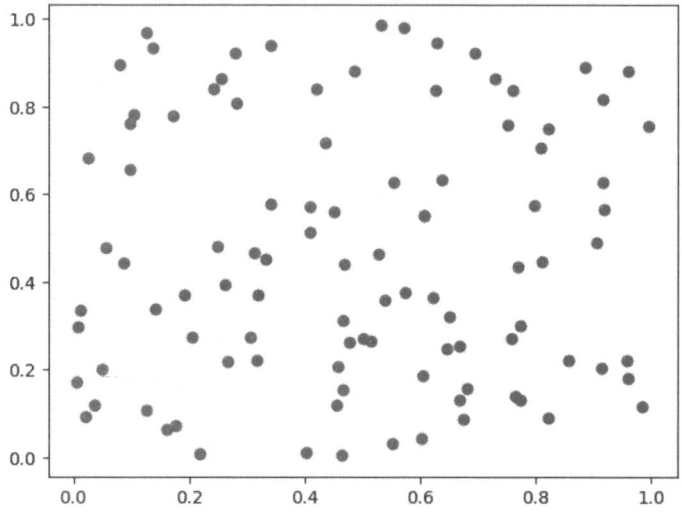

Рис. 15.1. График рассеяния случайно сгенерированных двумерных данных

Как показано на рис. 15.2, при помощи диаграммы Вороного мы можем создать некоторое множество областей, в каждую из которых попадает только небольшая часть от всех наблюдений, составляющих область нашего поиска.

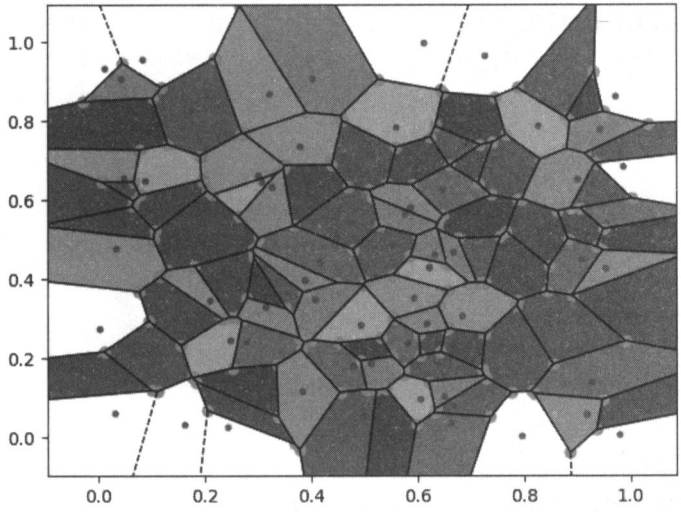

Рис. 15.2. Случайно сгенерированные двумерные данные, разделенные на множество различных областей

Меняя значение параметра `nlist` в библиотеке Faiss, мы можем регулировать количество необходимых кластеров. Другой параметр, `nprobe`, позволяет указать при создании запросов количество необходимых нам кластеров, среди которых нужно выполнить поиск соседних элементов целевого наблюдения. Увеличение значений `nlist` и `nprobe` позволяет повысить качество получаемых результатов поиска соседей, при этом растут нагрузка на вычислительные мощности и время обработки индексов IVF. Снижение значений каждого параметра приводит к обратному эффекту — ваш код будет выполняться быстрее, однако увеличится вероятность получения менее качественных результатов.

Как вы могли заметить, получившиеся результаты рассмотренного примера совпадают с результатами первого примера в этой главе. Поскольку мы работаем с небольшим количеством данных и используем всего три кластера, результаты классификации методом приближенных ближайших соседей вряд ли будут сильно отличаться от результатов классификации методом *k* ближайших соседей.

Дополнительные материалы для чтения

- Nearest Neighbor Indexes for Similarity Search (different ANN index types) ("Метод индексов ближайших соседей для поиска совпадений (различные типы индексов приближенных ближайших соседей)"; **https://oreil.ly/DVqgn**).

15.6. Оценка метода приближенных ближайших соседей

Задача

Вы хотите сравнить эффективность метода приближенных ближайших соседей и метода *k* ближайших соседей.

Решение

Рассчитаем значение полноты (recall) @k ближайших соседних элементов при использовании метода приближенных ближайших соседей относительно *k* ближайших соседей:

```
# Загрузить библиотеки
import faiss
import numpy as np
from sklearn import datasets
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler

# Количество ближайших соседей
k = 10

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Создать объект стандартизации
standardizer = StandardScaler()

# Стандартизировать признаки
features_standardized = standardizer.fit_transform(features)

# Создать переменную алгоритма k ближайших соседей с 10 ближайшими элементами
nearest_neighbors = NearestNeighbors(n_neighbors=k).fit(features_standardized)

# Настроить параметры faiss
n_features = features_standardized.shape[1] nlist = 3

# Создать индекс IVF
quantizer = faiss.IndexFlatIP(n_features)
index = faiss.IndexIVFFlat(quantizer, n_features, nlist)
```

```
# Обучить индекс и добавить векторы признаков
index.train(features_standardized)
index.add(features_standardized)
index.nprobe = 1

# Создать наблюдение
new_observation = np.array([[ 1,  1,  1,  1]])

# Рассчитать расстояние и индексы ближайших соседей наблюдения
knn_distances, knn_indices = nearest_neighbors.kneighbors(new_observation)

# Выполнить поиск внутри индексов двух ближайших соседей
ivf_distances, ivf_indices = index.search(new_observation, k)

# Найти пересечения
recalled_items = set(list(knn_indices[0])) & set(list(ivf_indices[0]))

# Вывести значение полноты
print(f"Recall @k={k}: {len(recalled_items)/k * 100}%")
Recall @k=10: 100.0%
```

Обсуждение

Полнота @k — это, в сущности, количество элементов, возвращаемых алгоритмом ANN при определенном значении k ближайших соседей, которые также появляются в аналогичных ближайших соседях в точке k, разделенных на k. В нашем примере для 10 ближайших соседей полнота составляет 100%, другими словами, при $k = 10$ алгоритмы ANN и k ближайших соседей возвращают одни индексы (хотя иногда в разном порядке).

Полнота является распространенной метрикой оценки алгоритма приближенных ближайших соседей относительно ближайших соседних элементов.

Дополнительные материалы для чтения

- Google's note on ANN for its Vertex Matching Engine Service (краткая статья от Google об использовании метода приближенных ближайших соседей в поисковой системе Vertex AI Search; **https://oreil.ly/-COc9**).

ГЛАВА 16

Логистическая регрессия

Введение

Несмотря на наличие слова "регрессия" в названии, *логистическая регрессия* (logistic regression) в действительности является популярным методом управляемой классификации. Логистическая регрессия (и производные вроде полиномиальной логистической регрессии) — это простой и понятный подход предсказания вероятного класса наблюдения. В этой главе мы рассмотрим способы обучения различных классификаторов при помощи логистической регрессии в scikit-learn.

16.1. Обучение бинарного классификатора

Задача

Вам нужно обучить простую модель классификации.

Решение

Воспользуемся классом `LogisticRegression` из библиотеки scikit-learn для обучения модели логистической регрессии:

```
# Загрузить библиотеку
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные с двумя классами
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)
```

```
# Создать объект логистической регрессии
logistic_regression = LogisticRegression(random_state=0)

# Обучить модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Несмотря на слово "регрессия" в названии, логистическая регрессия является популярным бинарным или двоичным классификатором (т. е. целевой вектор может принимать только два значения). В логистической регрессии линейная модель (например, $\beta_0 + \beta_1 x$) включена в логистическую (или сигмоидальную) функцию $\frac{1}{1+e^{-z}}$, что можно выразить формулой:

$$P(y_i = 1 \mid X) = \frac{1}{1+e^{-(\beta_0+\beta_1 x)}},$$

где $P(y_i = 1 \mid X)$ — вероятность, с которой значение y_i целевого наблюдения i присваивается классу 1; X — обучающие данные; β_0 и β_1 — изучаемые параметры; e — число Эйлера. Задача логистической функции заключается в приведении вывода функции к значению в диапазоне от 0 до 1, чтобы его можно было интерпретировать как вероятность. Если $P(y_i = 1 \mid X)$ больше 0,5, объект присваивается классу 1; в противном случае он присваивается классу 0.

В scikit-learn мы можем обучить модель логистической регрессии при помощи класса `LogisticRegression`. После обучения мы можем использовать модель для предсказания класса нового наблюдения:

```
# Создать новое наблюдение
new_observation = [[.5, .5, .5, .5]]

# Спрогнозировать класс
model.predict(new_observation)
array([1])
```

В данном примере наблюдение имеет класс 1. Мы также можем узнать вероятность присвоения наблюдения каждому классу:

```
# Узнать прогнозируемые вероятности
model.predict_proba(new_observation)
array([[0.17738424, 0.82261576]])
```

Вероятность отнесения нашего наблюдения к классу 0 составляет 17,7%, а к классу 1 — 82,2%.

16.2. Обучение многоклассового классификатора

Задача

Вам нужно обучить модель классификации, имея более двух классов.

Решение

Для обучения логистической регрессии в scikit-learn мы можем использовать класс `LogisticRegression` с методом one-vs-rest (один против всех) или полиномиальными методами:

```
# Загрузить библиотеку
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект логистической регрессии методом one-vs-rest
object logistic_regression = LogisticRegression(random_state=0, multi_class="ovr")

# Обучить модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Сама по себе логистическая регрессия — это алгоритм бинарной классификации, который не может работать с векторами, если количество классов превышает два. Однако есть пара расширений, которые позволяют преодолеть этот барьер. Во-первых, это логистическая регрессия методом *one-vs-rest* (one-vs-rest logistic regression, OvR), при которой для каждого класса создается отдельная модель, определяющая факт принадлежности наблюдения к этому классу (таким образом, мы получаем бинарную классификацию). Каждый случай классификации (например, принадлежность к классу 0) обрабатывается независимо от остальных.

Другим способом является использование *полиномиальной логистической регрессии* (multinominal logistic regression, MLR), в которой логистическая функция из *рецепта 16.1* заменяется многопеременной логистической функцией (softmax function):

$$P(y_i = 1 \mid X) = \frac{e^{\beta_k x_i}}{\sum_{j=1}^{K} e^{\beta_j x_i}},$$

где $P(y_i = 1 \mid X)$ — вероятность, с которой значение y_i целевого наблюдения i присваивается классу k; K — общее количество классов. Одним из преимуществ полиномиальной логистической регрессии является более качественное (т. е. лучше откалиброванное) прогнозирование вероятностей при помощи метода `predict_proba`.

При использовании `LogisticRegression` мы можем выбрать нужный нам способ; по умолчанию это логистическая регрессия методом one-vs-rest (ovr). Для переключения на полиномиальную логистическую регрессию нужно добавить аргумент со значением `multinomial`.

16.3. Уменьшение дисперсии за счет регуляризации

Задача

Вам необходимо уменьшить дисперсию модели логистической регрессии.

Решение

Настроим гиперпараметр показателя регуляризации C:

```
# Загрузить библиотеку
from sklearn.linear_model import LogisticRegressionCV
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект регрессии на основе дерева решений
logistic_regression = LogisticRegressionCV(
    penalty='l2', Cs=10, random_state=0, n_jobs=-1)
```

```
# Обучить модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Регуляризация используется в качестве метода ограничения сложных моделей для снижения их дисперсии. В частности, мы добавляем штрафы в функцию потерь, которую стремимся минимизировать, — это, как правило, L1- и L2-регуляризация. В случае L1:

$$\alpha\sum_{j=1}^{p}\left|\hat{\beta}_j\right|,$$

где $\hat{\beta}_j$ — значение параметров признака j из p обучаемых признаков; α — гиперпараметр, означающий величину регуляризации. В случае L2:

$$\alpha\sum_{j=1}^{p}\hat{\beta}_j^2.$$

С увеличением значений α возрастает величина штрафа для более высоких значений параметров (т. е. более сложных моделей). Вместо α в scikit-learn используется популярное обозначение C, где C является обратной величиной регуляризации: $C = 1/\alpha$. Для снижения дисперсии при использовании логистической регрессии мы можем рассматривать C как гиперпараметр и при помощи его настройки найти такое значение C, при котором получается наиболее качественная модель. Для настройки C в scikit-learn можно использовать класс `LogisticRegressionCV`. Параметр `Cs` в `LogisticRegressionCV` может принимать диапазон значений C для поиска (если список чисел с плавающей запятой представлен в виде аргумента) или, при использовании целых чисел, создавать список возможных значений, полученных из логарифмической шкалы в диапазоне от –10 000 до 10 000.

К сожалению, при использовании `LogisticRegressionCV` мы не можем осуществить поиск по различным штрафным членам. Для этого нам нужно обратиться к менее эффективным методам выбора моделей из *главы 12*.

16.4. Обучение классификатора на очень больших данных

Задача

Вам нужно обучить простую модель классификации на очень большом наборе данных.

Решение

Используем решатель на основе *стохастического усредненного градиентного спуска* (stochastic average gradient, SAG) для обучения модели логистической регрессии в scikit-learn при помощи класса `LogisticRegression`:

```
# Загрузить библиотеку
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект логистической регрессии
logistic_regression = LogisticRegression(random_state=0, solver="sag")

# Обучить модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Класс `LogisticRegression` библиотеки scikit-learn включает множество методов обучения моделей логистической регрессии, называемых *решателями* (solvers). В большинстве случаев scikit-learn самостоятельно выбирает походящий решатель или предупреждает о невозможности каких-либо операций. Однако нам это подходит не всегда.

Мы не будем подробно разбирать данный аспект здесь (более подробная информация представлена на слайдах Марка Шмидта (Mark Schmidt) по ссылке в *разд. "Дополнительные материалы для чтения" далее*); нам достаточно знать, что стохастический усредненный градиентный спуск позволяет обучить модель намного быстрее других решателей при работе с очень большими объемами данных. Поскольку система весьма чувствительна к масштабированию признаков, их крайне важно стандартизировать. При помощи аргумента `solver="sag"` мы можем добавить решатель в наш алгоритм обучения.

Дополнительные материалы для чтения

- Mark Schmidt "Minimizing Finite Sums with the Stochastic Average Gradient Algorithm" (Марк Шмидт "Минимизация конечных сумм при помощи алгоритма стохастического градиентного спуска"; **https://oreil.ly/K5rEG**).

16.5. Управление несбалансированными классами

Задача

Вам необходимо обучить простую модель классификации.

Решение

При помощи класса `LogisticRegression` библиотеки scikit-learn обучим модель логистической регрессии:

```
# Загрузить библиотеки
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Нарушить баланс, удалив 40 первых наблюдений класса
features = features[40:,:]
target = target[40:]

# Создать целевой вектор, где класс равен 0, остальное равно 1
target = np.where((target == 0), 0, 1)

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект регрессии методом дерева решений
logistic_regression = LogisticRegression(random_state=0, class_weight="balanced")

# Обучить модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

В классе `LogisticRegression`, как и многих других алгоритмах scikit-learn, есть встроенные методы управления несбалансированными классами. Если в наших данных встречаются крайне несбалансированные классы, от которых мы не избавились на этапе предобработки, мы можем использовать параметр `class_weight` для измерения весов классов, чтобы таким образом уравновесить их. В частности, при наличии аргумента `balanced` классы будут автоматически взвешены обратно пропорционально их частоте:

$$w_j = \frac{n}{kn_j},$$

где w_j — вес класса j; n — общее количество наблюдений; n_j — количество наблюдений класса j; k — общее число классов.

ГЛАВА 17

Метод опорных векторов

Введение

Для того чтобы лучше понять метод опорных векторов (support vector machines, SVM), нам сначала нужно разобраться с понятием *гиперплоскости* (hyperplanes). Формально гиперплоскость является подпространством с размерностью $n - 1$ пространства n. На первый взгляд это кажется сложным, хотя на практике все весьма просто. К примеру, если мы захотим разделить двумерное пространство, мы воспользуемся одномерной гиперплоскостью (т. е. линией). Если мы захотим разделить трехмерное пространство, то воспользуемся двумерной плоскостью (например, листом бумаги или одеялом). Гиперплоскость лишь обобщает эту идею для переноса на пространства *размерности n*.

При классификации данных *методом опорных векторов* алгоритм старается определить гиперплоскость, максимизирующую зазор (margin) между классами обучаемых данных. В случае двумерного множества данных с двумя классами мы можем представить гиперплоскость в виде широкой прямой полосы (т. е. линии разделения и зазоров), разделяющей два класса.

В этой главе мы научимся обучать системы на основе метода опорных векторов для различных ситуаций и узнаем, как можно использовать данный метод для решения некоторых общих проблем.

17.1. Обучение линейного классификатора

Задача

Вам нужно обучить модель для классификации наблюдений.

Решение

Используем *классификатор на основе опорного вектора* (support vector classifier, SVC) для поиска гиперплоскости, способной максимизировать зазоры между классами.

```
# Загрузить библиотеки
from sklearn.svm import LinearSVC
```

```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Загрузить данные с двумя классами и двумя признаками
iris = datasets.load_iris()
features = iris.data[:100,:2]
target = iris.target[:100]

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать классификатор на основе опорного вектора
svc = LinearSVC(C=1.0)

# Обучить модель
model = svc.fit(features_standardized, target)
```

Обсуждение

В scikit-learn простой классификатор SVC реализуется при помощи класса `LinearSVC`. Для того чтобы было проще понять принцип работы классификатора SVC, попробуем изобразить данные и гиперплоскость. Хотя SVC прекрасно справляется с многомерными данными, для примера мы взяли всего два признака и подмножество наблюдений, чтобы наши данные содержали только два класса. Это позволит нам визуализировать модель. Как было указано ранее, классификатор SVC осуществляет поиск гиперплоскости (при работе с двухмерными данными это линия) с максимальным зазором между классами. Используя приведенный ниже пример кода, мы можем визуализировать два класса данных в двумерном пространстве и прочертить гиперплоскость (рис. 17.1):

```
# Загрузить библиотеку
from matplotlib import pyplot as plt

# Сформировать график из точек и выделить классы разными цветами
color = ["black" if c == 0 else "lightgrey" for c in target]
plt.scatter(features_standardized[:,0], features_standardized[:,1], c=color)

# Создать гиперплоскость
w = svc.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-2.5, 2.5)
yy = a * xx - (svc.intercept_[0]) / w[1]
```

```
# Начертить гиперплоскость на графике
plt.plot(xx, yy)
plt.axis("off")
plt.show()
```

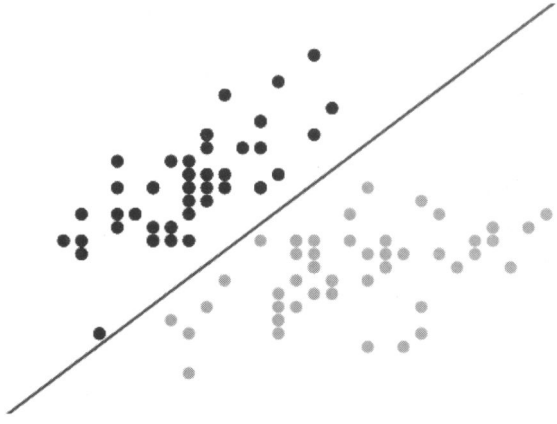

Рис. 17.1

На нашем графике все наблюдения класса 0 имеют черный цвет, а наблюдения класса 1 — светло-серый. Гиперплоскость является границей решения, которая определяет, к какому классу будут относиться новые наблюдения. В частности, любое наблюдение выше линии будет присвоено классу 0, а ниже линии — классу 1. Для того чтобы это проверить, мы создадим новое наблюдение в левом верхнем углу графика, которое предположительно должно быть отнесено к классу 0:

```
# Создать новое наблюдение
new_observation = [[ -2,  3]]

# Спрогнозировать класс нового наблюдения
svc.predict(new_observation)
array([0])
```

Следует отметить несколько вещей касательно классификации SVC. Во-первых, в нашем примере мы используем только бинарные данные (т. е. с двумя классами) с целью их визуализации, однако SVC также может работать с многоклассовыми данными. Во-вторых, как видно на графике, гиперплоскость по определению является линейной (т. е. не кривой). В нашем случае это было не так важно, поскольку данные можно было разделить на два класса линейно, прочертив ровную прямую гиперплоскость. К сожалению, на практике такое встречается редко.

Обычно мы не можем так идеально разделить два класса. В таких ситуациях необходимо найти баланс между максимизацией зазора гиперплоскости и минимизацией неверной классификации. В SVC за минимизацию ошибок отвечает гиперпараметр *C*. Он является параметром модели обучения SVC и вводит штрафы на неверную классификацию точек данных. При небольшом значении *C* классификатор допускает неверную классификацию (большое смещение при низкой дисперсии).

Если значение *C* высокое, классификатор вводит штрафы за неправильную классификацию данных, в результате чего появляются изгибы, позволяющие избежать ошибок при классификации точек данных (маленькое смещение при высокой дисперсии).

В scikit-learn *C* определяется параметром `C`; по умолчанию `C=1.0`. Нам следует рассматривать *C* как гиперпараметр алгоритма обучения, который можно настроить при помощи способов выбора модели из *главы 12*.

17.2. Управление линейно неразделимыми классами через ядра свертки

Задача

Вам необходимо обучить классификатор на основе опорного вектора, при этом у вас есть линейно неразделимые классы.

Решение

Обучим расширенную модель на основе опорного вектора при помощи функций ядра свертки для создания нелинейных границ решений:

```
# Загрузить библиотеки
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Установить целое начальное значение
np.random.seed(0)

# Создать два признака
features = np.random.randn(200, 2)

# Использовать логический вентиль XOR (вам не обязательно знать, что это)
# для создания линейно неразделимых классов
target_xor = np.logical_xor(features[:, 0] > 0, features[:, 1] > 0)
target = np.where(target_xor, 0, 1)

# Создать модель на основе опорного вектора с ядром радиальной базисной функции
svc = SVC(kernel="rbf", random_state=0, gamma=1, C=1)

# Обучить классификатор
model = svc.fit(features, target)
```

Обсуждение

Мы не будем подробно останавливаться на описании метода опорных векторов. Тем не менее небольшое объяснение может помочь лучше понять модели на основе SVM и ядер свертки. Классификацию методом опорного вектора можно выразить формулой:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x_i, x_{i'}),$$

где β_0 — смещение (bias); S — множество всех наблюдений опорных векторов; α — параметр модели для запоминания; $(x_i, x_{i'})$ — сочетание двух наблюдений x_i и $x_{i'}$ опорных векторов. K является ядерной функцией, которая сравнивает схожесть элементов x_i и $x_{i'}$. Не стоит волноваться, если вы ничего не знаете о ядерной функции. Для решения текущей задачи нужно лишь помнить, что: во-первых, K определяет тип гиперплоскости, используемой для разделения классов; во-вторых, при помощи различных ядер мы можем создавать разные гиперплоскости. К примеру, если нам нужна простая линейная гиперплоскость, как в *рецепте 17.1*, мы можем использовать линейное ядро:

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j},$$

где p является количеством признаков. Если же нам нужна нелинейная граница решений, следует воспользоваться полиномиальным ядром:

$$K(x_i, x_i') = \left(r + \gamma \sum_{j=1}^{p} x_{ij} x_{ij}' \right)^d,$$

где d является степенью функции полиномиального ядра. Мы также можем использовать другое распространенное решение с поддержкой опорных векторов — *ядро радиальной базисной функции*:

$$K(x_i, x_i') = \exp\left(-\gamma \sum_{j=1}^{p} \left(x_{ij} x_{ij}' \right)^2 \right),$$

где γ является гиперпараметром, который должен быть больше 0. Из объяснения выше следует основная идея, заключающаяся в том, что при наличии линейно неразделимых данных мы можем заменить линейное ядро другим для создания нелинейной гиперплоскости границы решения.

Для простоты понимания рассмотрим простой пример работы ядер. Данная функция, основанная на алгоритме Себастьяна Рашки (Sebastian Raschka), выводит график наблюдений и границ гиперплоскости в двумерном пространстве (рис. 17.2).

Вам не обязательно знать принцип работы функции; я добавил ее в пример кода, чтобы вы самостоятельно смогли опробовать ваши идеи на практике:

```
# Создать график из наблюдений и границ решений гиперплоскости
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier):
    cmap = ListedColormap(("red", "blue"))
    xx1, xx2 = np.meshgrid(np.arange(-3, 3, 0.02), np.arange(-3, 3, 0.02))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.1, cmap=cmap)
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker="+", label=cl)
```

Данные в нашем примере содержат два признака (т. е. двумерны) и целевой вектор с классом для каждого наблюдения. Нам важно, чтобы классы были *линейно неразделимыми*. Другими словами, мы не можем прочертить прямую линию, чтобы разделить их. Сначала давайте создадим классификатор на основе опорных векторов с линейным ядром:

```
# Создать классификатор на основе опорных векторов с линейным ядром
svc_linear = SVC(kernel="linear", random_state=0, C=1)

# Обучить модель
svc_linear.fit(features, target)
SVC(C=1, kernel='linear', random_state=0)
```

Далее, поскольку у нас всего два признака, мы работаем в двумерном пространстве и можем визуализировать наблюдения, их классы и линейную плоскость нашей модели:

```
# Вывести график наблюдений и гиперплоскости
plot_decision_regions(features, target, classifier=svc_linear)
plt.axis("off")
plt.show()
```

Из графика видно, что наша линейная гиперплоскость весьма плохо справляется с разделением классов. Теперь попробуем заменить линейное ядро на ядро радиальной базисной функции и обучим на его основе новую модель:

```
# Создать модель на основе опорных векторов с ядром радиальной базисной функции
svc = SVC(kernel="rbf", random_state=0, gamma=1, C=1)

# Обучить классификатор
model = svc.fit(features, target)
```

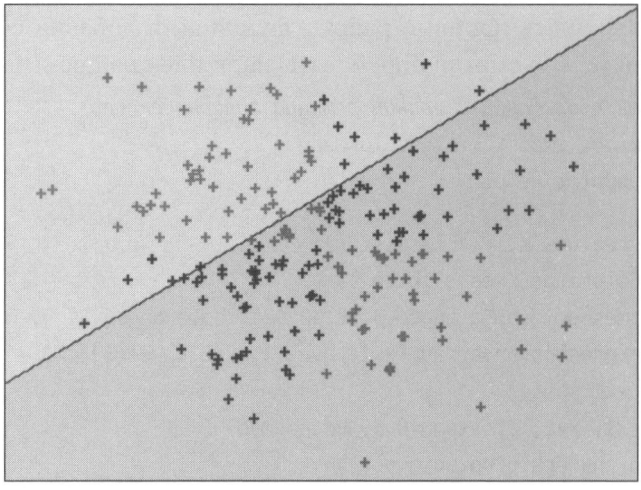

Рис. 17.2

Потом визуализируем наблюдения и гиперплоскость (рис. 17.3):

```
# Вывести график наблюдений и гиперплоскости
plot_decision_regions(features, target, classifier=svc)
plt.axis("off")
plt.show()
```

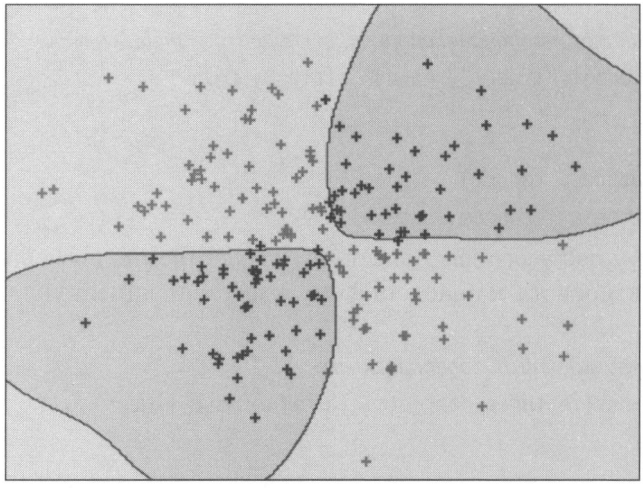

Рис. 17.3

Ядро радиальной базисной функции позволяет нам создавать более эффективную границу решения, которая превосходит линейное ядро в качестве разделения двух классов. Все вышесказанное является основным стимулом использования ядер в методе опорных векторов.

В scikit-learn выбор ядра осуществляется путем изменения значения параметра `kernel`. После выбора ядра нужно указать соответствующие параметры ядра, на-

пример значение d (параметр degree) в полиномиальных ядрах и значение γ (параметр gamma) в ядрах радиальной базисной функции. Нам также понадобится задать параметр штрафов C. При обучении модели лучшим решением будет представить эти параметры в качестве гиперпараметров и попробовать различные способы выбора модели для комбинирования их значений с целью создания наиболее эффективной модели.

17.3. Создание прогнозируемых вероятностей

Задача

Вам нужно получить информацию о прогнозируемых вероятностях класса наблюдения.

Решение

Воспользуемся классом SVC библиотеки scikit-learn, укажем probability=True, обучим модель и затем применим метод predict_proba для вывода откалиброванных вероятностей:

```
# Загрузить библиотеки
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать классификатор на основе опорного вектора
svc = SVC(kernel="linear", probability=True, random_state=0)

# Обучить модель
model = svc.fit(features_standardized, target)

# Обучить новое наблюдение
new_observation = [[.4, .4, .4, .4]]
```

```
# Вывести прогнозируемые вероятности
model.predict_proba(new_observation)
array([[0.00541761, 0.97348825, 0.02109414]])
```

Обсуждение

Многие из изученных нами ранее алгоритмов используют оценку вероятностей для прогнозирования классов. Например, в методе *k* ближайших соседей классы *k* соседей наблюдения, в сущности, являются множеством голосов, из которых выводится вероятность присвоения наблюдения определенному классу. Предпочтение отдается классу с наибольшим показателем вероятности. Классификатор на основе метода опорных векторов (SVC) базируется на использовании гиперплоскости для создания отдельных областей, что само по себе не создает оценку вероятности принадлежности наблюдения к определенному классу. Однако мы можем использовать калибровку вероятностей принадлежности к классу, с некоторыми пояснениями. В SVC с двумя классами можно использовать *калибровку Платта* (Platt scaling), при которой сначала обучается модель SVC, а потом обучается логистическая регрессия на основе кросс-валидации для соотнесения вывода SVC и вероятностей:

$$P(y=1 \mid x)=\frac{1}{1+e^{(A\times f(x)+B)}},$$

где A и B — параметры векторов; $f(x)$ — удаленность наблюдения i от гиперплоскости. При наличии более двух классов используется расширенный вариант калибровки Платта.

С практической точки зрения создание прогнозируемых вероятностей имеет два недостатка. Во-первых, так как мы обучаем вторую модель с использованием кросс-валидации, процесс создания прогнозируемых вероятностей может ощутимо замедлить обучение нашей модели. Во-вторых, из-за использования кросс-валидации для создания прогнозируемых вероятностей они не всегда соответствуют прогнозируемым классам. Другими словами, наблюдение может быть присвоено классу 1 при значении прогнозируемой вероятности класса 1 менее 0,5.

В scikit-learn прогнозируемые вероятности должны создаваться в процессе обучения модели. Для этого нам нужно указать значение `True` для параметра `probability` объекта класса `SVC`. После обучения модели мы можем вывести значения прогнозируемых вероятностей для каждого класса при помощи метода `predict_proba`.

17.4. Поиск опорных векторов

Задача

Вам необходимо определить, какие наблюдения являются опорными векторами гиперплоскости решений.

Решение

Обучим модель и используем атрибут support_vectors_:

```
# Загрузить библиотеки
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Загрузить данные с двумя классами
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать классификатор на основе опорного вектора
svc = SVC(kernel="linear", random_state=0)

# Обучить классификатор
model = svc.fit(features_standardized, target)

# Вывести опорные векторы
model.support_vectors_
array([[-0.5810659 ,  0.42196824, -0.80497402, -0.50860702],
       [-1.52079513, -1.67737625, -1.08231219, -0.86427627],
       [-0.89430898, -1.4674418 ,  0.30437864,  0.38056609],
       [-0.5810659 , -1.25750735,  0.09637501,  0.55840072]])
```

Обсуждение

Название метода опорных векторов (support vector machines) отражает суть процесса создания гиперплоскости из сравнительно небольшого количества наблюдений, которые называются *опорными векторами*. Можно сказать, что гиперплоскость как бы "лежит" на опорных векторах. По этой причине опорные векторы являются крайне важной частью нашей модели. К примеру, удаление наблюдений, не являющихся опорными векторами, из данных не сильно повлияет на модель; если же мы удалим опорный вектор, то не получим максимальный зазор для нашей модели.

В scikit-learn представлено множество способов определения опорных векторов, которые мы можем использовать после обучения модели SVC. В рассматриваемом примере мы при помощи атрибута support_vectors_ вывели признаки наблюдений

четырех опорных векторов нашей модели. В качестве другого решения мы можем просмотреть индексы опорных векторов, используя атрибут support_:

```
model.support_
```

```
array([23, 41, 57, 98], dtype=int32)
```

Наконец, с помощью атрибута n_support_ мы можем определить количество опорных векторов каждого класса:

```
model.n_support_
```

```
array([2, 2], dtype=int32)
```

17.5. Управление несбалансированными классами

Задача

Вам нужно обучить классификатор на основе метода опорных векторов, при этом у вас есть несбалансированные классы.

Решение

Увеличим штраф за неправильную классификацию немногочисленного класса при помощи параметра class_weight:

```
# Загрузить библиотеку
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Загрузить данные с двумя классами и двумя признаками
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]

# Нарушить баланс, удалив 40 первых наблюдений класса
features = features[40:,:]
target = target[40:]

# Создать целевой вектор, где класс равен 0, остальное равно 1
target = np.where((target == 0), 0, 1)

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)
```

```
# Создать классификатор на основе опорного вектора
svc = SVC(kernel="linear", class_weight="balanced", C=1.0, random_state=0)

# Обучить классификатор
model = svc.fit(features_standardized, target)
```

Обсуждение

В модели на основе опорных векторов есть гиперпараметр C, который определяет величину штрафа за ошибочную классификацию наблюдения. Одним из способов управления несбалансированными классами моделей с опорными векторами является определение веса C для каждого класса:

$$C_k = C \times w_j,$$

где C — величина штрафа за неправильную классификацию; w_j — вес, обратно пропорциональный частоте появления класса j; C_k — значение C класса k. Идея заключается в увеличении штрафа за неправильную классификацию малочисленных классов, чтобы не допустить их "подавления" большим классом.

При использовании объекта класса `SVC` в scikit-learn мы можем автоматически присвоить значения C_k, указав `class_weight="balanced"`. Параметр `balanced` автоматически увеличит веса классов:

$$w_j = \frac{n}{kn_j},$$

где w_j — вес класса j; n — количество наблюдений; n_j — количество наблюдений класса j; k — количество всех классов.

ГЛАВА 18

Наивный байесовский классификатор

Введение

Теорема Байеса является одной из основных теорем теории вероятностей и позволяет понять вероятность события $P(A \mid B)$, взяв в расчет новую информацию $P(B \mid A)$ и уже известные данные вероятности события $P(A)$:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}.$$

За последнее десятилетие метод Байеса получил широкое распространение как альтернатива традиционному частотному (frequentist) подходу в образовании, государственном управлении и бизнесе. В машинном обучении теорема Байеса применяется в качестве основы *наивного байесовского классификатора* (naive Bayes classifier). При помощи наивных байесовских классификаторов мы можем объединить множество полезных качеств из области машинного обучения внутри единого классификатора. К ним относятся:

- интуитивный подход;
- возможность работать с небольшими данными;
- невысокие требования к вычислительным мощностям для обучения и прогнозирования;
- уверенные результаты при гибкости настроек.

В основе наивного байесовского классификатора, в частности, лежит формула:

$$P(y \mid x_1, ..., x_j) = \frac{P(x_1, ..., x_j \mid y)P(y)}{P(x_1, ..., x_j)},$$

где:

- $P(y \mid x_1, ..., x_j)$ называется *апостериорной* вероятностью (posterior) и выражает вероятность отнесения наблюдения к классу при известных значениях $x_1, ..., x_j$ наблюдений j;
- $P(x_1, ..., x_j \mid y)$ называется функцией *правдоподобия* (likelihood) и выражает вероятность присвоения значений $x_1, ..., x_j$ признакам наблюдения известного класса y;

- $P(y)$ называется *априорной* вероятностью (prior) и выражает наше предположение о вероятности класса y для учета данных;
- $P(x_1, ..., x_j)$ называется *частным,* или *маргинальным, распределением* (marginal probability).

При работе с наивным байесовским классификатором мы сравниваем показатели апостериорной вероятности наблюдений каждого возможного класса. Поскольку частное распределение при любом сравнении является константой, мы сравниваем числители апостериорной вероятности каждого класса. Каждое наблюдение присваивается классу $\hat{y}$ с наибольшим числителем апостериорной вероятности.

Стоить отметить две особенности, присущие наивным байесовским классификаторам. Во-первых, нам необходимо допустить статистическое распределение правдоподобия $P(x_j \mid y)$ для каждого признака данных. Наиболее характерными вариантами являются нормальное распределение (Гаусса), полиномиальное распределение и распределение Бернулли. Выбор распределения обычно основывается на сущности признаков (непрерывные, двоичные и т. д.). Во-вторых, понятие наивности в названии байесовского классификатора означает, что все признаки и получаемые правдоподобия являются независимыми. Это допущение зачастую является ошибочным, хотя на практике никак не мешает создавать эффективные классификаторы.

В этой главе мы узнаем, как при помощи инструментов scikit-learn можно обучить три вида наивных байесовских классификаторов с использованием трех различных значений функции правдоподобия. Затем мы научимся калибровать прогнозы наивных байесовских моделей для улучшения их интерпретируемости.

18.1. Обучение классификатора для непрерывных признаков

Задача

Вы хотите обучить наивный байесовский классификатор для работы только с непрерывными признаками.

Решение

Воспользуемся наивным байесовским классификатором на основе распределения Гаусса из библиотеки scikit-learn:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB

# Загрузить данные
iris = datasets.load_iris()
```

```
features = iris.data
target = iris.target

# Создать объект наивной байесовской классификации Гаусса
classifer = GaussianNB()

# Обучить модель
model = classifer.fit(features, target)
```

Обсуждение

Наибольшее распространение среди всех типов получил *наивный байесовский классификатор на основе распределения Гаусса*. Он основан на предположении, что правдоподобие значений признаков x при наличии наблюдения класса y выводится из нормального распределения:

$$p(x_j \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_j - \mu_y)^2}{2\sigma_y^2}\right),$$

где σ_y^2 и μ_y — дисперсия и среднее значение признака x_j класса y. Ввиду наличия нормального распределения данный классификатор лучше всего подходит для работы с непрерывными признаками.

В scikit-learn для обучения наивного байесовского классификатора Гаусса применяется метод `fit` по аналогии с другими классификаторами. Таким образом он может быть использован для прогнозирования классов:

```
# Создать новое наблюдение
new_observation = [[ 4,  4,  4,  0.4]]

# Спрогнозировать класс
model.predict(new_observation
array([1])
```

Одной примечательной особенностью этих классификаторов является возможность назначить априорную вероятность для целевых классов. Для этого используется параметр `priors` в классе `GaussianNB`, который принимает список из значений вероятностей для каждого класса целевого вектора:

```
# Создать объект наивной байесовской классификации методом Гаусса
# с априорной вероятностью для каждого класса
clf = GaussianNB(priors=[0.25, 0.25, 0.5])

# Обучить модель
model = classifer.fit(features, target)
```

Мы можем оставить значение аргумента параметра априорной вероятности пустым, тогда модель самостоятельно произведет настройку на основе данных.

Наконец, необходимо иметь в виду, что сырые значения прогнозируемых вероятностей, получаемые при использовании гауссовского наивного байесовского классификатора (с помощью метода predict_proba), не откалиброваны. Другими словами, мы не можем доверять этим данным. Для того чтобы эти прогнозируемые вероятности стали более пригодны для использования, нам нужно откалибровать их при помощи изотонической регрессии (isotonic regression) или схожего метода.

Дополнительные материалы для чтения

- How the Naive Bayes Classifier Works in Machine Learning ("Принцип работы наивного байесовского классификатора в машинном обучении"; **https://oreil.ly/9yqSw**).

18.2. Обучение классификатора для дискретных и количественных признаков

Задача

Вы хотите обучить наивный байесовский классификатор для работы только с дискретными и количественными данными.

Решение

Воспользуемся полиномиальным наивным байесовским классификатором:

```
# Загрузить библиотеки
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer

# Создать текст
text_data = np.array(['I love Brazil. Brazil!',
                      'Brazil is best',
                      'Germany beats both']

# Создать мешок слов
count = CountVectorizer()
bag_of_words = count.fit_transform(text_data)

# Создать матрицу признаков
features = bag_of_words.toarray()
```

```
# Создать целевой вектор
target = np.array([0,0,1])

# Создать полиномиальный наивный байесовский классификатор
classifer = MultinomialNB(class_prior=[0.25, 0.5])

# Обучить модель
model = classifer.fit(features, target)
```

Обсуждение

Принцип работы *полиномиального наивного байесовского классификатора* аналогичен гауссовскому наивному байесовскому классификатору, однако для признаков характерно полиномиальное распределение. На практике это означает, что данный классификатор обычно используется при работе с дискретными данными (например, при выяснении рейтинга фильма в диапазоне от 1 до 5). Полиномиальный наивный байесовский классификатор, в частности, применяется для классификации текстов при помощи мешка слов или подходов *tf-idf* (*см. рецепты 6.9* и *6.10*).

В нашем примере мы создали игрушечный набор данных с тремя наблюдениями и преобразовали текст из строк в матрицу признаков из мешка слов и сопутствующий целевой вектор. Затем мы использовали класс `MultinomialNB` для обучения модели, определив априорные вероятности для двух классов (`pro-brazil` и `pro-germany`).

Классификатор `MultinomialNB` аналогичен `GaussianNB`; для обучения модели используется метод `fit`, а для прогнозирования наблюдений — метод `predict`:

```
# Создать новое наблюдение
new_observation = [[0, 0, 0, 1, 0, 1, 0]]

# Спрогнозировать класс нового наблюдения
model.predict(new_observation)
array([0])
```

Если мы не указываем `class_prior`, то априорные вероятности получаются на основе имеющихся данных. Если же мы укажем `fit_prior=False`, то в качестве априорной вероятности будет использоваться равномерное распределение (uniform distribution).

Наконец, в `MultinomialNB` содержится дополнительный гиперпараметр коэффициента сглаживания `alpha`, который также необходимо настроить. По умолчанию он имеет значение `1.0`; значение `0.0` указывает на отсутствие сглаживания.

18.3. Обучение наивного байесовского классификатора для бинарных признаков

Задача

У вас есть данные с бинарными признаками, и вам нужно обучить наивный байесовский классификатор.

Решение

Используем наивный байесовский классификатор на основе распределения Бернулли:

```
# Загрузить библиотеку
import numpy as np
from sklearn.naive_bayes import BernoulliNB

# Создать три бинарных признака
features = np.random.randint(2, size=(100, 3))

# Создать целевой бинарный вектор
target = np.random.randint(2, size=(100, 1)).ravel()

# Создать объект наивной байесовской классификации Бернулли
# с априорной вероятностью для каждого класса
classifer = BernoulliNB(class_prior=[0.25, 0.5])

# Обучить модель
model = classifer.fit(features, target)
```

Обсуждение

Принцип *наивного байесовского классификатора с распределением Бернулли* основывается на допущении, что все признаки двоичны и могут принимать только два значения (например, номинальный категориальный признак, созданный при помощи унитарного кода). Наивный байесовский классификатор Бернулли также используется для классификации текста, однако матрица признаков выражает наличие или отсутствие слова в документе. Более того, в `BernoulliNB`, как и в `MultinomialNB`, есть гиперпараметр коэффициента сглаживания `alpha`, который мы можем настроить при помощи методов выбора модели. Наконец, для использования априорной вероятности мы можем задать параметр `class_prior`, принимающий список значений априорных вероятностей для каждого класса. Для использования равномерной априорной вероятности нужно указать `fit_prior=False`:

```
model_uniform_prior = BernoulliNB(class_prior=None, fit_prior=False)
```

18.4. Калибровка прогнозируемых вероятностей

Задача

Вы хотите произвести калибровку прогнозируемых вероятностей, получаемых из наивного байесовского классификатора, для улучшения их интерпретируемости.

Решение

Используем класс `CalibratedClassifierCV`:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
from sklearn.calibration import CalibratedClassifierCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект наивной байесовской классификации Гаусса
classifer = GaussianNB()

# Создать объект сигмоидальной калибровки на основе кросс-валидации
classifer_sigmoid = CalibratedClassifierCV(classifer, cv=2, method='sigmoid')

# Произвести калибровку вероятностей
classifer_sigmoid.fit(features, target)

# Создать новое наблюдение
new_observation = [[ 2.6,  2.6,  2.6,  0.4]]

# Вывести значения калиброванных вероятностей
classifer_sigmoid.predict_proba(new_observation)
array([[0.31859969, 0.63663466, 0.04476565]])
```

Обсуждение

Вероятности классов являются важной частью моделей машинного обучения. В большинстве алгоритмов scikit-learn мы можем узнать прогнозируемые вероятности принадлежности к классу при помощи метода `predict_proba`. Это может быть

особенно полезно, если, например, для присвоения объекта определенному классу показатель вероятности должен быть не менее 90%. Тем не менее выводимые некоторыми моделями (включая наивные байесовские классификаторы) значения вероятностей не имеют связи с реальными данными. Другими словами, метод `predict_proba` может спрогнозировать вероятность принадлежности наблюдения к классу на уровне 0,70, тогда как в реальности этот показатель будет равен 0,10 или 0,99. В наивных байесовских классификаторах, в частности, при распределении прогнозируемых вероятностей различных целевых классов получаемые сырые данные могут иметь крайние значения, близкие к 0 и 1.

Для получения пригодных к использованию значений прогнозируемых вероятностей нам нужно произвести так называемую *калибровку*. Для этого в scikit-learn есть класс `CalibratedClassifierCV`, при помощи которого мы можем создать хорошо откалиброванные вероятности на основе *k*-блочной кросс-валидации. В `CalibratedClassifierCV` для обучения модели используются обучающие наборы, а для калибровки берется тестовый набор. Получаемые значения вероятностей являются средними показателями *k* блоков.

Мы можем видеть разницу между сырыми и откалиброванными вероятностями на рассмотренном примере. В нашем решении мы воспользовались наивным байесовским классификатором Гаусса. Если мы обучим классификатор и попробуем предсказать вероятности для новых наблюдений, то в итоге получим слишком экстремальные значения вероятности:

```
# Обучить наивный байесовский классификатор Гаусса и предсказать вероятности классов
classifer.fit(features, target).predict_proba(new_observation)
array([[2.31548432e-04, 9.99768128e-01, 3.23532277e-07]])
```

В свою очередь, после калибровки (как мы сделали в нашем решении) мы получим совершенно иные результаты:

```
# Вывести откалиброванные вероятности
array([[0.31859969, 0.63663466, 0.04476565]])
array([[ 0.31859969,  0.63663466,  0.04476565]])
```

В классе `CalibratedClassifierCV` представлены два метода калибровки: сигмоидальная модель Платта и изотоническая регрессия, которые определяются параметром `method`. Нужно отметить, что изотоническая регрессия относится к непараметрическим методам и при малом размере примеров (скажем, 100 наблюдений) нередко страдает от переобучения. В нашем решении мы использовали сигмоидальную модель, поскольку набор данных ирисов включает 150 наблюдений.

ГЛАВА 19
Кластеризация

Введение

До этого момента мы во многом рассматривали контролируемое машинное обучение, где у нас есть доступ и к признакам, и к целевым показателям. К сожалению, в реальности мы не всегда работаем с такими данными. Очень часто нам доступна лишь информация признаков. К примеру, представим, что у нас есть данные продаж овощного магазина, и мы на их основе хотим разделить покупателей по принципу участия в программе лояльности. Мы не можем использовать обучение с учителем для этих целей, поскольку у нас нет целевых показателей, на основе которых мы можем построить и оценить модели. Однако есть другой способ — неконтролируемое обучение (или обучение без учителя). Если поведение участников программы лояльности действительно отличается от других покупателей овощного магазина, то средняя величина отличий поведения двух покупателей одной группы будет меньше показателя между покупателями, участвующими и не участвующими в программе лояльности. Другими словами, мы получим два кластера наблюдений.

Цель алгоритмов кластеризации заключается в определении скрытого группирования наблюдений, что при правильном использовании позволяет нам прогнозировать класс наблюдений даже без целевого вектора. Существует множество алгоритмов кластеризации, построенных на различных подходах к разграничению кластеров данных. В этой главе мы научимся подбирать подходящий алгоритм кластеризации в scikit-learn и использовать его.

19.1. Кластеризация методом *k* средних

Задача

Вы хотите разделить наблюдения на *k* групп.

Решение

Используем *кластеризацию методом k средних*:

```
# Загрузить библиотеку
from sklearn import datasets
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Создать объект k средних
cluster = KMeans(n_clusters=3, random_state=0, n_init="auto")

# Обучить модель
model = cluster.fit(features_std)
```

Обсуждение

Среди всех способов кластеризации наибольшее распространение получил метод *k* средних (k-means clustering). При кластеризации этим методом алгоритм пытается разделить наблюдения на *k* групп с примерно равной дисперсией. Количество групп *k* пользователь указывает самостоятельно в качестве гиперпараметра. При использовании *k* средних, в частности, происходит:

1. Создание *k* "центральных" точек кластеров в случайных местах.
2. Для каждого наблюдения:
   - расчет расстояния от каждого наблюдения до *k* центральных точек;
   - присвоение наблюдения кластеру ближайшей центральной точки.
3. Перемещение центральных точек к средним значениям (т. е. центрам) соответствующих им кластеров.
4. Повторение шагов 2 и 3 до прекращения изменений кластерной принадлежности наблюдений.

На этом моменте выполнение алгоритма завершается.

Необходимо выделить три особенности метода *k* средних. Во-первых, при кластеризации методом *k* средних кластеры имеют выпуклую форму (т. е. круг, сфера). Во-вторых, все признаки масштабируются одинаково. Для того чтобы не нарушить данное правило, в нашем примере мы использовали стандартизацию признаков. В-третьих, все группы уравновешиваются (т. е. имеют примерно одинаковое количество признаков). Если же у нас не получается соблюсти все требования, следует обратиться к другому методу кластеризации.

В scikit-learn метод кластеризации на основе *k* средних реализуется через класс `KMeans`. Наиболее важным является параметр `n_clusters`, при помощи которого указывается количество кластеров *k*. Иногда значение *k* определяется сущностью данных (например, в наборе данных об учениках школы каждый кластер может соответствовать классу с учениками), однако в большинстве случаев мы не знаем количество кластеров. Тогда нам может потребоваться некоторый критерий для выбора значения *k*. Например, коэффициенты силуэта (*см. рецепт 11.9*) сопоставляют значения сходства внутри кластеров и между кластерами. Более того, ввиду существенной нагрузки на вычислительные мощности нам могут понадобиться все ядра системы. Для этого нужно добавить параметр `n_jobs=-1`.

В нашем примере мы немного упростили задачу, воспользовавшись примером ирисов, поскольку уже знали, что в нем представлены три класса. Соответственно, мы указали значение `k = 3`. Мы можем просмотреть спрогнозированные классы каждого наблюдения при помощи атрибута `labels_`:

```
# Вывести предугаданный класс
model.labels_
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2, 1, 2,
       2, 1, 2, 1, 1, 2, 2, 2, 2, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 2, 2, 2,
       2, 1, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 1], dtype=int32)
```

Если мы сравним эти данные с истинным классом наблюдений, то увидим, что несмотря на некоторое несоответствие разметки (т. е. 0, 1 и 2), метод *k* средних справился неплохо:

```
# Вывести истинный класс
iris.target
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Вместе с тем можно верно предположить, что эффективность *k* средних существенно упадет, если мы укажем неверное число кластеров.

Наконец, мы можем использовать обученную модель кластеризации, как и другие модели в scikit-learn, для прогнозирования значений новых наблюдений:

```
# Создать новое наблюдение
new_observation = [[0.8, 0.8, 0.8, 0.8]]
```

```
# Спрогнозировать кластер наблюдения
model.predict(new_observation)
```

```
array([2], dtype=int32)
```

Наблюдение присваивается кластеру с наиболее близкой центральной точкой. Мы также можем просмотреть значения этих центральных точек при помощи атрибута `cluster_centers_`:

```
# Отобразить центры классстеров
model.cluster_centers_
```

```
array([[-1.01457897,  0.85326268, -1.30498732, -1.25489349],
       [-0.01139555, -0.87600831,  0.37707573,  0.31115341],
       [ 1.16743407,  0.14530299,  1.00302557,  1.0300019 ]])
```

Дополнительные материалы для чтения

- Introduction to K-means Clustering ("Введение в кластеризацию методом *k* средних"; **https://oreil.ly/HDfUz**).

19.2. Ускорение кластеризации методом *k* средних

Задача

Вы хотите разделить наблюдения на *k* групп, однако метод *k* средних отнимает слишком много времени.

Решение

Используем мини-пакетную кластеризацию *k* средних (mini-batch k-means):

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MiniBatchKMeans

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
```

```
# Создать объект k средних
cluster = MiniBatchKMeans(n_clusters=3, random_state=0, batch_size=100,
        n_init="auto")

# Обучить модель
model = cluster.fit(features_std)
```

Обсуждение

Принцип работы *мини-пакетной кластеризации k средних* схож с алгоритмом *k* средних из *рецепта 19.1*. Коротко говоря, в мини-пакетной кластеризации *k* средних ресурсоемкие вычисления производятся не на всех наблюдениях, а лишь на случайной выборке. Такой подход позволяет существенно сократить время поиска соответствий (т. е. обучение данных) алгоритмом при незначительном ухудшении качества.

Класс `MiniBatchKMeans` во многом совпадает с `KMeans`, однако в нем есть важный дополнительный параметр `batch_size`, при помощи которого мы можем задать количество случайных наблюдений в каждой группе (batch). С увеличением размера группы возрастает нагрузка на вычислительные системы в процессе обучения.

19.3. Кластеризация со сдвигом среднего

Задача

Вы хотите группировать наблюдения без предварительных условий количества и формы кластеров.

Решение

Используем кластеризацию со сдвигом среднего (mean shift clustering):

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MeanShift

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
```

```
# Создать объект сдвига среднего значения
cluster = MeanShift(n_jobs=-1)

# Обучить модель
model = cluster.fit(features_std)
```

Обсуждение

Одними из недостатков кластеризации методом *k* средних являются необходимость указания количества кластеров *k* до начала обучения и допущение относительно формы кластеров. Сдвиг среднего значения (mean shift) позволяет обойти данное ограничение.

Сдвиг среднего значения — это достаточно простая концепция, которую, однако, сложно объяснить. Для простоты попробуем привести пример. Представьте футбольное поле в очень туманную погоду (т. е. двумерное пространство признаков) и 100 человек на нем (т. е. наши наблюдения). Из-за тумана один человек видит только объекты вблизи себя. Каждую минуту каждый человек смотрит вокруг и делает шаг в направлении наиболее хорошо различимого соседа. Постепенно люди начнут объединяться в группы, поскольку будут двигаться в направлении большего скопления людей. В результате мы увидим кластеры людей по всему полю. Каждый человек причисляется к кластеру, в который он попал.

Метод сдвига среднего значения, представленный в scikit-learn при помощи класса `MeanShift`, более сложен для понимания, хотя следует описанной выше логике. В `MeanShift` есть два важных параметра, которые нужно знать. Во-первых, параметр `bandwidth` регулирует радиус области (т. е. ядро), которое обследует наблюдение для определения направления сдвига. В терминах нашей аналогии это дальность видимости в тумане. По умолчанию подбирается оптимальный параметр `bandwidth` (со значительным ростом затрат на вычисления), однако мы также можем настроить его вручную. Во-вторых, при использовании сдвига среднего значения в некоторых случаях наблюдения могут отсутствовать за пределами одного ядра. Другими словами, человек на футбольном поле никого не видит. В `MeanShift` такие наблюдения-сироты по умолчанию причисляются к ядру ближайшего наблюдения. При помощи `cluster_all=False` мы можем оставить эти наблюдения отдельно; в таком случае они будут иметь метку `-1`.

Дополнительные материалы для чтения

- The mean shift clustering algorithm, EFAVDB ("Алгоритм кластеризации со сдвигом среднего"; **https://oreil.ly/Gb3VG**).

19.4. Кластеризация DBSCAN

Задача

Вам нужно объединить наблюдения в кластеры с высокой плотностью.

Решение

Воспользуемся кластеризацией DBSCAN[1]:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Создать объект DBSCAN
cluster = DBSCAN(n_jobs=-1)

# Обучить модель
model = cluster.fit(features_std)
```

Обсуждение

Метод кластеризации *DBSCAN* строится на идее образования кластеров из плотно расположенных наблюдений, без условий относительно формы. В DBSCAN, в частности:

1. Выбирается случайное наблюдение x_i.
2. Точка x_i с минимальным количеством ближайших соседей рассматривается как часть кластера.
3. Шаг 2 повторяется для всех наблюдений x_i, соседних наблюдений и т. д. Они являются основными наблюдениями кластера.

---

[1] Density-Based Spatial Clustering of Applications with Noise — плотностной алгоритм пространственной кластеризации с присутствием шума. — *Прим. ред.*

4. После обработки всех наблюдений на шаге 3 выбирается новая случайная точка (весь процесс начинается сначала).

После этой процедуры мы получаем множество основных наблюдений (core observations) для некоторого количества кластеров. Наконец, каждое наблюдение, близкое к кластеру, но не его ядру, становится частью этого кластера, а более дальние наблюдения маркируются как выбросы.

В DBSCAN необходимо настраивать три основных параметра.

`eps`

Наибольшее расстояние от одного наблюдения до другого, при котором оно считается соседним.

`min_samples`

Наименьшее количество наблюдений на расстоянии не больше `eps` от наблюдения, при котором оно считается основным.

`metric`

Метрика расстояния, используемая `eps`, например `minkowski` или `euclidean` (отметим, что при использовании расстояния Минковского для настройки интенсивности метрики можно использовать параметр `p`).

Если посмотреть на кластеры в нашем обучаемом наборе, можно увидеть, что два кластера помечены как `0` и `1`, а выбросы — как `-1`:

```
# Вывести принадлежность значений кластера
model.labels_
array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1, -1,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1, -1,
        0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,  0,  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,  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])
```

Дополнительные материалы для чтения

- DBSCAN (**https://oreil.ly/QBx3a**).

19.5. Иерархическая кластеризация

Задача

Вам нужно объединить наблюдения при помощи иерархии кластеров.

Решение

Воспользуемся агломеративной кластеризацией:

```
# Загрузить библиотеку
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Создать объект агломеративной кластеризации
cluster = AgglomerativeClustering(n_clusters=3)

# Обучить модель
model = cluster.fit(features_std)
```

Обсуждение

Агломеративная кластеризация (agglomerative clustering) — это мощный и гибкий алгоритм иерархической кластеризации. Здесь каждое наблюдение в изначальной форме представляет свой кластер. Далее происходит итеративный процесс слияния на основании некоторых критериев. Увеличение размеров кластеров продолжается до достижения некоторого условия. Класс `AgglomerativeClustering` библиотеки scikit-learn использует параметр `linkage` при определении стратегии слияния для минимизации:

- дисперсии объединенных кластеров (`ward`);
- среднего расстояния между наблюдениями двух разных кластеров (`average`);
- наибольшего расстояния между наблюдениями двух разных кластеров (`complete`).

Будет полезно знать еще два параметра. Во-первых, параметр `affinity` позволяет задать метрику расстояния, используемую при связывании (`minkowski`, `euclidean` и т. д.). Во-вторых, параметр `n_clusters` определяет количество кластеров, которые должен найти алгоритм классификации. Другими словами, кластеризация считается успешной при достижении указанного значения `n_clusters`.

Как и в других алгоритмах кластеризации, мы можем использовать атрибут `labels_` для вывода ассоциированных значений наблюдений кластера:

```
# Вывести принадлежность значений кластера
model.labels_
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, 2, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 0, 2, 0, 2, 0, 2, 0, 2, 2, 0, 2, 0, 2, 0,
       2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0, 2, 0, 0, 2,
       2, 2, 2, 0, 2, 2, 2, 2, 2, 0, 2, 2, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

ГЛАВА 20

Тензоры в PyTorch

Введение

Если изученная нами ранее библиотека NumPy является основополагающим инструментом технологического стека машинного обучения, то библиотека PyTorch — основополагающий инструмент для работы с тензорами технологического стека глубокого обучения. Прежде чем мы перейдем к изучению непосредственно глубокого обучения, следует узнать больше о тензорах в PyTorch и попробовать воспроизвести различные операции, которые мы уже выполняли при помощи NumPy в *главе 1*.

Хотя PyTorch — это не единственная библиотека для глубокого обучения, однако она пользуется невероятной популярностью как в научной, так и производственной сферах. Тензоры в PyTorch *очень* похожи на массивы в NumPy. Тем не менее при работе с ними мы можем использовать графические ускорители (специальное аппаратное обеспечение для глубокого обучения) для осуществления тензорных операций. В этой главе мы ознакомимся с основами работы с тензорами в PyTorch и множеством различных низкоуровневых операций.

20.1. Создание тензора

Задача

Вам нужно создать тензор.

Решение

Используем инструменты библиотеки PyTorch для создания тензора:

```
# Загрузить библиотеку
import torch

# Создать вектор-строку
tensor_row = torch.tensor([1, 2, 3])
```

```
# Создать вектор-столбец
tensor_column = torch.tensor(
    [
        [1],
        [2],
        [3]
    ]
)
```

Обсуждение

Тензоры являются основной структурой данных в PyTorch и во многом похожи на многомерные массивы NumPy, которые мы использовали в *главе 1*. Аналогично векторам и массивам, тензоры могут быть горизонтальными (т. е. строки) или вертикальными (т. е. столбцы).

Дополнительные материалы для чтения

- Документация PyTorch: тензоры (**https://oreil.ly/utaTD**).

20.2. Создание тензора из массива NumPy

Задача

Вам необходимо преобразовать массив NumPy в тензор PyTorch.

Решение

Воспользуемся методом `from_numpy` библиотеки PyTorch:

```
# Загрузить библиотеки
import numpy as np
import torch

# Создать массив NumPy
vector_row = np.array([1, 2, 3])

# Преобразовать массив NumPy в тензор
tensor_row = torch.from_numpy(vector_row)
```

Обсуждение

Как вы могли заметить, синтаксис PyTorch очень похож на NumPy. Более того, он позволяет нам преобразовать массивы NumPy в тензоры PyTorch, чтобы использо-

вать графические (GPU), а также другие аппаратные ускорители для вычислений. На момент написания книги NumPy весьма часто упоминается в документации PyTorch. Кроме того, в PyTorch есть возможность совместного использования памяти тензорами PyTorch и массивами NumPy для предотвращения чрезмерной нагрузки.

Дополнительные материалы для чтения

- Bridge with NumPy ("Взаимодействие с NumPy", документация PyTorch; **https://oreil.ly/zEJo6**).

20.3. Создание разреженного тензора

Задача

У вас есть данные с минимальным количеством ненулевых значений, которые вы хотите представить при помощи тензора.

Решение

Используем метод `to_sparse` библиотеки PyTorch:

```
# Загрузить библиотеку
import torch

# Создать тензор
tensor = torch.tensor(
[
[0, 0],
[0, 1],
[3, 0]
]
)

# Преобразовать обычный тензор в разреженный
sparse_tensor = tensor.to_sparse()
```

Обсуждение

С точки зрения работы с памятью, разреженные тензоры позволяют наиболее эффективно представить данные, состоящие по большей части из нулей. В *главе 1* мы при помощи библиотеки scipy создавали матрицу со сжатым хранением строкой (CSR), что в сущности не является массивом NumPy.

С помощью класса torch.Tensor мы можем на основе одних объектов создавать как регулярные, так и разреженные матрицы. Мы можем проверить типы только что созданных нами тензоров и убедиться, что они принадлежат одному классу:

```
print(type(tensor))
print(type(sparse_tensor))
<class 'torch.Tensor'>
<class 'torch.Tensor'>
```

Дополнительные материалы для чтения

- Sparse Tensor ("Разреженные тензоры", документация PyTorch; **https://oreil.ly/8J3IO**).

20.4. Выбор элементов тензора

Задача

Нам нужно выбрать конкретные элементы тензора.

Решение

Используем NumPy-подобную индексацию и срезы для получения элементов:

```
# Загрузить библиотеку
import torch

# Создать векторный тензор
vector = torch.tensor([1, 2, 3, 4, 5, 6])

# Создать матричный тензор
matrix = torch.tensor(
    [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]
)

# Вывести третий элемент вектора
vector[2]
tensor(3)
```

```
# Вывести элемент второй строки и второго столбца
matrix[1,1]
tensor(5)
```

Обсуждение

Подобно массивам NumPy и большинству других объектов Python, индексация тензоров PyTorch начинается с нуля. Помимо индексации, они также поддерживают срезы. Одним из ключевых отличий является то, что при индексации тензора PyTorch для получения одного элемента функция возвращает тензор, а не значение непосредственно объекта (в виде целого числа или числа с плавающей запятой). Синтаксис для получения срезов аналогичен NumPy и возвращает объекты тензорного типа данных в PyTorch:

```
# Вывести все элементы вектора
vector[:]
array([1, 2, 3, 4, 5, 6])

# Вывести все элементы до третьего включительно
vector[:3]
tensor([1, 2, 3])

# Вывести все элементы после третьего
vector[3:]
tensor([4, 5, 6])

# Вывести последний элемент
vector[-1]
tensor(6)

# Вывести элементы первых двух строк и всех столбцов матрицы
matrix[:2,:]
tensor([[1, 2, 3],
        [4, 5, 6]])

# Вывести элементы всех строк и второго столбца
matrix[:,1:2]
tensor([[2],
        [5],
        [8]])
```

Одним из существенных отличий тензоров PyTorch является вре́менная невозможность использования отрицательных значений при получении срезов. По этой причине попытка развернуть тензор при получении срезов приводит к ошибке:

```
# Развернуть вектор
vector[::-1]
ValueError: step must be greater than zero
```

Вместо этого при необходимости такой операции мы можем использовать метод `flip`:

```
vector.flip(dims=(-1,))
tensor([6, 5, 4, 3, 2, 1])
```

Дополнительные материалы для чтения

- Operations on Tensors ("Операции с тензорами", документация PyTorch; **https://oreil.ly/8-xj7**).

20.5. Описание тензора

Задача

Вы хотите описать форму, тип данных и формат тензора, а также используемое аппаратное обеспечение.

Решение

Узнаем данные атрибутов `shape`, `dtype`, `layout` и `device` тензора:

```
# Загрузить библиотеку
import torch

# Создать тензор
tensor = torch.tensor([[1,2,3], [1,2,3]])

# Узнать форму тензора
tensor.shape
torch.Size([2, 3])

# Узнать тип данных элементов тензора
tensor.dtype
torch.int64

# Узнать структуру памяти тензора
tensor.layout
torch.strided
```

```
# Узнать используемое тензором устройство
tensor.device
device(type='cpu')
```

Обсуждение

Тензоры в PyTorch обладают набором полезных атрибутов для сбора информации о выбранном тензоре, к которым относятся:

`shape`

Возвращает данные размерности тензора.

`dtype`

Возвращает данные о типе данных объектов тензора.

`layout`

Возвращает данные о структуре памяти (наиболее часто это `strided` для плотных тензоров).

`device`

Возвращает данные об устройствах, используемых тензорами (CPU/GPU).

Ключевым отличием между тензорами и массивами является атрибут `device`, поскольку при работе с тензорами мы можем использовать аппаратные ускорители вроде GPU.

20.6. Операции над элементами

Задача

Вы хотите применить некоторую операцию ко всем элементам тензора.

Решение

Используем для наших целей *транслирование* (broadcasting) в PyTorch:

```
# Загрузить библиотеку
import torch

# Создать тензор
tensor = torch.tensor([1, 2, 3])

# Транслировать арифметическую операцию на все элементы тензора
tensor * 100
tensor([100, 200, 300])
```

Обсуждение

Мы можем использовать транслирование в PyTorch с базовыми операциями для распараллеливания процессов за счет поддержки аппаратных компонентов, например GPU. Это применимо к математическим операциям Python (+, –, ×, /), а также другим функциям, унаследованным PyTorch. В отличие от NumPy, PyTorch не использует метод `vectorize` для применения функции ко всем элементам тензора. Вместе с тем в PyTorch включены все математические инструменты, необходимые для распределения и ускорения привычных операций, используемых для машинного обучения.

Дополнительные материалы для чтения

- Broadcasting Semantics ("Семантика транслирования", документация PyTorch; **https://oreil.ly/NsPpa**).
- Vectorization and Broadcasting with PyTorch ("Векторизация и транслирование в PyTorch"; **https://oreil.ly/dfzIJ**).

20.7. Поиск наибольших и наименьших значений

Задача

Вам необходимо найти максимальные и минимальные значения элементов тензора.

Решение

Используем методы `max` и `min` библиотеки PyTorch:

```
# Загрузить библиотеку
import torch

# Создать тензор
torch.tensor([1,2,3])

# Найти наибольшее значение
tensor.max()
tensor(3)

# Найти наименьшее значение
tensor.min()
tensor(1)
```

Обсуждение

Методы max и min могут помочь нам найти максимальное и минимальное значения элементов тензора. Эти методы также применимы к многомерным тензорам:

```
# Создать многомерный тензор
tensor = torch.tensor([[1,2,3],[1,2,5]])

# Найти наибольшее значение
tensor.max()
```

```
tensor(5)
```

20.8. Изменение формы тензоров

Задача

Вы хотите изменить форму (количества строк и столбцов) тензора с сохранением значений элементов.

Решение

Используем метод reshape библиотеки PyTorch:

```
# Загрузить библиотеку
import torch

# Создать тензор формы 4x3
tensor = torch.tensor([[1, 2, 3],
                       [4, 5, 6],
                       [7, 8, 9],
                       [10, 11, 12]])

# Изменить форму тензора на 2x6
tensor.reshape(2, 6)
```

```
tensor([[ 1,  2,  3,  4,  5,  6],
        [ 7,  8,  9, 10, 11, 12]])
```

Обсуждение

В процессе глубокого обучения часто приходится производить различные манипуляции с формой тензоров. Это связано с тем, что используемые в нейронной сети тензоры должны иметь особую форму. Поскольку требования к форме тензора меняются от нейрона к нейрону внутри нейронной сети, будет полезно понимать низкоуровневые процессы ввода и вывода в глубоком обучении.

20.9. Транспонирование тензора

Задача

Вам нужно транспонировать тензор.

Решение

Воспользуемся методом `mT`:

```
# Загрузить библиотеку
import torch

# Создать двумерный тензор
tensor = torch.tensor([[[1,2,3]]])

# Транспонировать его
tensor.mT
tensor([[1],
        [2],
        [3]])
```

Обсуждение

Процесс транспонирования в PyTorch в некоторой степени отличается от NumPy. Используемый в массивах NumPy метод `T` поддерживается в PyTorch только для работы с двумерными тензорами; на момент написания данной книги при работе с другими формами возникала ошибка. Более предпочтительным вариантом работы с группами тензоров является метод `mT`, поскольку с его помощью мы можем масштабировать многомерные тензоры.

Еще одним способом транспонирования тензоров любой формы в PyTorch является метод `permute`:

```
tensor.permute(*torch.arange(tensor.ndim - 1, -1, -1))
tensor([[1],
        [2],
        [3]])
```

Данный метод также применим к одномерным тензорам (где значение транспонированного тензора совпадает с оригинальным).

20.10. Преобразование в одномерный тензор

Задача

Вам необходимо преобразовать тензор в одномерный тензор.

Решение

Используем метод flatten:

```
# Загрузить библиотеку
import torch

# Create tensor # Создадим тензор
tensor = torch.tensor([[1, 2, 3],
                       [4, 5, 6],
                       [7, 8, 9]])

# Flatten tensor # Преобразуем его в одномерный тензор
tensor.flatten()

tensor([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Обсуждение

Преобразование тензора в одномерный тензор помогает сжать многомерный тензор до одномерного.

20.11. Вычисление скалярного произведения

Задача

Вам нужно вычислить скалярное произведение двух тензоров.

Решение

Используем метод dot:

```
# Загрузить библиотеку
import torch

# Создать один тензор
tensor_1 = torch.tensor([1, 2, 3])
```

```
# Создать другой тензор
tensor_2 = torch.tensor([4, 5, 6])

# Вычислить скалярное произведение двух тензоров
tensor_1.dot(tensor_2)
tensor(32)
```

Обсуждение

Вычисление скалярного произведения двух тензоров является распространенной и полезной операцией как в области глубокого обучения, так и в области извлечения информации. Возможно, вы помните пример, приводимый ранее в книге, где мы использовали скалярное произведение двух векторов для поиска на основе косинусного коэффициента. Воспроизведение данной операции в PyTorch с применением GPU (вместо CPU в случае NumPy или scikit-learn) может обеспечить значительный прирост производительности при решении задач поиска информации.

Дополнительные материалы для чтения

- Vectorization and Broadcasting with PyTorch ("Векторизация и транслирование в PyTorch"; **https://oreil.ly/lIjtB**).

20.12. Умножение тензоров

Задача

Вам нужно выполнить умножение двух тензоров.

Решение

Воспользуемся базовыми арифметическими операциями Python:

```
# Загрузить библиотеку
import torch

# Создать один тензор
tensor_1 = torch.tensor([1, 2, 3])

# Создать другой тензор
tensor_2 = torch.tensor([4, 5, 6])

# Перемножить два тензора
tensor_1 * tensor_2
tensor([ 4, 10, 18])
```

Обсуждение

PyTorch поддерживает базовые арифметические операции, включая ×, +, – и /. Хотя одной из наиболее частых операций в глубоком обучении является умножение, не стоит забывать про возможность складывания, вычитания и деления тензоров.

Прибавим один тензор к другому:

```
tensor_1+tensor_2
```

```
tensor([5, 7, 9])
```

Вычтем один тензор из другого:

```
tensor_1-tensor_2
```

```
tensor([-3, -3, -3])
```

Разделим один тензор на другой:

```
tensor_1/tensor_2
```

```
tensor([0.2500, 0.4000, 0.5000])
```

ГЛАВА 21

Нейронные сети

Введение

В основе нейронных сетей лежат вычислительные единицы, называемые *нейронами* (neuron), или *узлами* (node или unit). Нейрон получает некоторую входную информацию, умножает ее на параметр (так называемый вес), суммирует взвешенные входные значения между собой, добавляя некоторое значение смещения (обычно 0), и подставляет полученное значение в функцию активации. Затем полученный результат передается другим нейронам сети на более глубокий уровень (при их наличии).

Нейронную сеть можно изобразить в виде последовательности связанных слоев, соединяющих значения признаков наблюдений с одной стороны и целевые значения (например, класс наблюдений) с другой. Самым простым видом искусственной нейронной сети считается *нейронная сеть с прямой связью* (feedforward neural networks), или *многослойный персептрон* (multilayer perceptron), которая используется при работе с любыми реальными данными. "Прямая связь" в названии означает, что значения признака наблюдения проходят непосредственно через сеть, преобразуясь в каждом слое, чтобы полученные в конечном счете выходные данные соответствовали (или были похожи) целевым значениям.

Нейронная сеть с прямой связью имеет три типа слоев (layers). Каждый нейрон входного слоя (input layer) сети содержит одно значение признака наблюдения. Например, если наблюдение имеет 100 признаков, входной слой будет состоять из 100 нейронов. В конце сети находится выходной слой (output layer), где происходит преобразование данных из промежуточных слоев (*скрытых слоев*) в подходящие для конкретной задачи значения. К примеру, если мы хотим осуществить бинарную классификацию, можем использовать выходной слой с одним нейроном, в котором при помощи сигмоидальной функции происходит преобразование выходного значения в пределах 0 и 1, что соответствует прогнозируемой вероятности класса.

Между входным и выходным слоями находятся скрытые слои (hidden layers). В них происходит преобразование входных значений признаков в выходные значения, соответствующие целевому классу. Нейронные сети, включающие большое количество скрытых слоев (например, 10, 100, 1000), считаются "глубокими". Соответственно, процесс обучения глубоких нейронных сетей называется *глубоким обучением*.

Нейронные сети, как правило, создаются на основе всех параметров, полученных при помощи нормального равномерного распределения в виде небольших случайных значений. После того как наблюдение (или чаще множество наблюдений, называемое пакетом (batch)) проходит через всю сеть, полученное выходное значение сравнивается с истинным значением наблюдения при помощи функции потерь. Это называется *прямым распространением* (forward propagation). Затем алгоритм движется в обратную сторону через сеть, чтобы определить влияние параметра на величину ошибки между прогнозируемыми и истинными значениями — это *обратное распространение* (back propagation). Алгоритм оптимизации определяет величину корректировки веса каждого параметра для улучшения выходного результата.

В процессе обучения нейронной сети происходит многократное повторение прямого и обратного распространения для каждого наблюдения обучающего набора данных (одна итерация, при которой все наблюдения проходят через нейронную сеть, называется эпохой; обучение обычно включает множество эпох) с цикличным обновлением значений параметров при помощи *градиентного спуска* для их последовательной оптимизации в соответствии с необходимыми выходными данными.

В этой главе мы научимся создавать, обучать и оценивать различные нейронные сети при помощи библиотеки PyTorch из предыдущей главы. Инструмент PyTorch широко используется в области глубокого обучения во многом благодаря наличию понятных API и интуитивной простоте низкоуровневых операций над тензорами, которые лежат в основе нейронных сетей. Ключевой особенностью PyTorch является библиотека *autograd*, которая обеспечивает автоматический расчет и хранение градиентов, участвующих в оптимизации параметров сети после завершения прямого и обратного распространения.

Нейронные сети, созданные при помощи кода PyTorch, могут обучаться как на CPU (т. е. ваш ноутбук), так и GPU (т. е. специальный компьютер для глубокого обучения). На практике для обучения нейронных сетей на основе реальных данных в основном используют GPU, поскольку это позволяет ускорить процесс обучения сложных сетей с большим количеством данных, в сравнении с CPU. Тем не менее приведенные в этой книге примеры нейронных сетей достаточно простые и небольшие, что позволяет за считаные минуты обучить их с помощью CPU или ноутбука. Вам лишь следует помнить, что использование CPU для обработки большого количества тренировочных данных при обучении крупных сетей может *существенно* замедлить процесс относительно GPU.

21.1. Использование autograd в PyTorch

Задача

Вы хотите с помощью библиотеки autograd из PyTorch рассчитать градиенты после завершения прямого и обратного распространения.

Решение

Создадим тензоры с параметром requires_grad, равным True:

```
# Импортировать библиотеки
import torch

# При помощи torch создать тензор, требующий градиенты
t = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

# Выполнить операцию с тензором, имитируя "прямое распространение"
tensor_sum = t.sum()

# Запустить обратное распространение
tensor_sum.backward()

# Вывести значения градиентов
t.grad
tensor([1., 1., 1.])
```

Обсуждение

Autograd является одной из ключевых особенностей PyTorch, во многом благодаря которой этот фреймворк так широко используется в глубоком обучении. Исследователи и энтузиасты, создающие нейронные сети с нуля, ценят PyTorch за интуитивность, которая обеспечивается простотой расчета, хранения и визуализации градиентов.

Благодаря использованию ориентированного ациклического графа (directed acyclic graph, DAG) PyTorch отслеживает все данные и проводимые на их основе вычисления. Это невероятно полезная опция, однако нужно внимательно следить за тем, какие операции мы проводим над данными при использовании градиентов. При работе с autograd мы не можем просто преобразовать тензоры в массивы NumPy и наоборот, не "разбив граф", что означает операции, которые не поддерживают autograd:

```
import torch

tensor = torch.tensor([1.0,2.0,3.0], requires_grad=True)
tensor.numpy()
RuntimeError: Can't call numpy() on Tensor that requires grad. Use
 tensor.detach().numpy() instead.
```

Для преобразования данного тензора в массив NumPy нам нужно использовать метод detach(), который разобьет граф и лишит нас возможности рассчитать градиенты автоматически. Стоит помнить, что несмотря на удобство данного метода, разделение тензора не позволяет автоматически рассчитывать градиент в PyTorch.

Дополнительные материалы для чтения

- PyTorch Autograd Tutorial ("Обучение PyTorch Autograd"; **https://oreil.ly/mOWSw**).

21.2. Предобработка данных для нейронных сетей

Задача

Вы хотите предварительно обработать данные для последующего использования в нейронной сети.

Решение

Воспользуемся классом `StandardScaler` библиотеки scikit-learn для стандартизации каждого признака:

```
# Загрузить библиотеки
from sklearn import preprocessing
import numpy as np

# Создать признак
features = np.array([[-100.1, 3240.1],
                     [-200.2, -234.1],
                     [5000.5, 150.1],
                     [6000.6, -125.1],
                     [9000.9, -673.1]])

# Создать масштабатор
scaler = preprocessing.StandardScaler()

# Преобразовать его в тензор
features_standardized_tensor = torch.from_numpy(features)

# Вывести признак
features_standardized_tensor
tensor([[-100.1000, 3240.1000],
        [-200.2000, -234.1000],
        [5000.5000,  150.1000],
        [6000.6000, -125.1000],
        [9000.9000, -673.1000]], dtype=torch.float64)
```

Обсуждение

Хотя мы уже видели подобный процесс в *рецепте 4.2*, нам следует обратиться к нему еще раз ввиду важности для нейронных сетей. Обычно параметры нейронной сети инициализируются (создаются) в форме небольших случайных значений. Работа нейронных сетей часто ухудшается, если значения признаков сильно больше значений параметров. Более того, поскольку значения признаков наблюдений комбинируются при прохождении отдельного нейрона, все признаки должны иметь единый масштаб.

По этой причине рекомендуется (хотя не всегда обязательно; например, если у нас есть только бинарные признаки) стандартизировать каждый признак так, чтобы среднее значение их показателей было 0, а стандартное отклонение было 1. Для этого подойдет класс `StandardScaler` из библиотеки scikit-learn.

Если же вы осуществляете эту операцию после создания тензора с аргументом `requires_grad=True`, вам нужно использовать встроенные возможности PyTorch, чтобы не разбить граф. Обычно мы проводим стандартизацию признаков до начала обучения нейронной сети, однако будет полезно знать, как это можно сделать в PyTorch:

```
# Загрузить библиотеку
import torch

# Создать признаки
torch_features = torch.tensor([[-100.1, 3240.1],
                               [-200.2, -234.1],
                               [5000.5, 150.1],
                               [6000.6, -125.1],
                               [9000.9, -673.1]], requires_grad=True)

# Вычислить среднее значение и стандартное отклонение
mean = torch_features.mean(0, keepdim=True)
standard_deviation = torch_features.std(0, unbiased=False, keepdim=True)

# При помощи среднего значения и стандартного отклонения
# стандартизировать признаки
torch_features_standardized = torch_features - mean
torch_features_standardized /= standard_deviation

# Вывести стандартизированные признаки
torch_features_standardized
tensor([[-1.1254,  1.9643],
        [-1.1533, -0.5007],
```

```
        [ 0.2953, -0.2281],
        [ 0.5739, -0.4234],
        [ 1.4096, -0.8122]], grad_fn=<DivBackward0>)
```

21.3. Разработка нейронной сети

Задача

Вы хотите смоделировать нейронную сеть.

Решение

Воспользуемся классом `nn.Module` библиотеки PyTorch для создания архитектуры простой нейронной сети:

```
# Загрузить библиотеки
import torch
import torch.nn as nn

# Определить нейронную сеть
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.fc1 = nn.Linear(10, 16)
        self.fc2 = nn.Linear(16, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, x):
        x = nn.functional.relu(self.fc1(x))
        x = nn.functional.relu(self.fc2(x))
        x = nn.functional.sigmoid(self.fc3(x))
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()

# Определить функцию потерь и оптимизацию
loss_criterion = nn.BCELoss()
optimizer = torch.optim.RMSprop(network.parameters())

# Вывести нейронную сеть на экран
network
```

```
SimpleNeuralNet(
  (fc1): Linear(in_features=10, out_features=16, bias=True)
  (fc2): Linear(in_features=16, out_features=16, bias=True)
  (fc3): Linear(in_features=16, out_features=1, bias=True)
)
```

Обсуждение

Нейронные сети состоят из слоев нейронов. Существует огромное множество типов слоев и вариантов их объединения для создания архитектуры нейронной сети. Несмотря на принятые паттерны (которые мы рассмотрим в следующей главе), выбор подходящей архитектуры относится скорее к искусству, чем к области исследований.

При создании нейронной сети с прямой связью в PyTorch нам понадобится определить необходимую архитектуру и процесс обучения. Важно помнить, что каждый нейрон скрытого слоя:

1. Принимает множество входных данных.
2. Взвешивает каждый входной элемент на основе значения параметра.
3. Складывает все веса входных данных с учетом некоторого смещения (обычно это 0).
4. В большинстве случаев применяет некоторую функцию (*функцию активации*).
5. Отправляет выходные данные нейронам следующего слоя.

Во-первых, нам необходимо определить количество нейронов, включенных в слой, и функцию активации для каждого скрытого и выходного слоя. В целом чем больше нейронов имеет слой, тем более сложные паттерны можно использовать для обучения модели. Вместе с тем большое количество нейронов может привести к переобучению нейронной сети на тренировочном наборе данных, что негативно скажется на обработке тестового набора данных.

Одной из популярных функций активации скрытого слоя является *выпрямленная линейная единица* (rectified linear unit, ReLU):

$$f(x) = \max(0, z),$$

где z — сумма взвешенных входных данных и смещения. Как можно заметить, при z больше 0 функция активации возвращает z; в противном случае, функция возвращает 0. Простая функция активации имеет ряд полезных свойств (мы не будем подробно разбирать их здесь), что делает ее удобной для использования в нейронных сетях. Вместе с тем нужно помнить, что существует много десятков вариантов функции активации.

Во-вторых, нам требуется определить количество скрытых слоев, используемых в нейронной сети. Увеличение количества слоев позволяет устанавливать более сложные связи, при этом растет потребление ресурсов.

В-третьих, мы должны задать структуру функции активации (при наличии) выходного слоя. Тип выходной функции, как правило, определяется целью создания нейронной сети. Ниже представлены некоторые популярные паттерны выходного слоя.

Бинарная классификация (binary classification).

Один нейрон с сигмоидальной функцией активации.

Многоклассовая классификация (multiclass classification).

k нейронов (где k является количеством целевых классов) и многопеременная логистическая функция активации.

Регрессия (regression).

Один нейрон без функции активации.

В-четвертых, нам нужно задать функцию потерь (измеряет качество соответствия прогнозов истинным значениям), которая в большинстве случаев также определяется нашей конечной целью.

Бинарная классификация (binary classification).

Бинарная перекрестная энтропия.

Многоклассовая классификация (multiclass classification).

Категориальная перекрестная энтропия.

Регрессия (regression).

Среднеквадратическая ошибка.

В-пятых, нам необходимо задать оптимизацию, которая будет отражать нашу стратегию поиска значений параметра при прохождении через функцию потерь для минимизации ошибки. К наиболее популярным методам оптимизации относятся стохастический градиентный спуск (stochastic gradient descent), стохастический градиентный спуск методом импульса (stochastic gradient descent with momentum), среднеквадратичное распространение RMSProp (root mean square propagation) и алгоритм Adam (adaptive moment estimation) — оценка адаптивного момента (более подробную информацию об оптимизации можно узнать по ссылке в *разд. "Дополнительные материалы для чтения" далее*).

В-шестых, мы можем выбрать одну или несколько метрик для оценки качества, например правильность (accuracy).

В нашем примере мы используем пространство имен `torch.nn.Module` для создания простой последовательной нейронной сети для бинарной классификации. При стандартном подходе в PyTorch мы создаем дочерний класс от класса `torch.nn.Module`, отражающий архитектуру сети в методе `__init__` и определяющий набор математических операций, которые хотим совершить при каждой прямой итерации метода `forward` для данного класса. В PyTorch существует множество способов определения нейронной сети. В данном случае мы использовали функциональные методы для наших функций активации (например, `nn.functional.relu`), однако мы также можем представить функции активации в виде слоев. Для того что-

бы организовать все компоненты нейронной сети в слой, мы можем воспользоваться классом Sequential:

```
# Импортировать библиотеки
import torch

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать и вывести на экран сеть
SimpleNeuralNet()
```

```
SimpleNeuralNet(
  (sequential): Sequential(
    (0): Linear(in_features=10, out_features=16, bias=True)
    (1): ReLU()
    (2): Linear(in_features=16, out_features=16, bias=True)
    (3): ReLU()
    (4): Linear(in_features=16, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
```

В обоих случаях у нас получилась двухслойная нейронная сеть (при подсчете слоев мы не учитываем входной слой, т. к. в нем нет параметров для обучения), определенная при помощи последовательной модели PyTorch. Каждый слой является "плотным" (или "полностью связным"), другими словами, все нейроны предыдущего слоя связаны со всеми нейронами последующего слоя.

Аргумент out_features=16 для скрытого слоя означает, что в нем содержится 16 нейронов. В соответствии с установками в методе forward нашего класса эти нейроны

имеют функции активации ReLU: `x = nn.functional.relu(self.fc1(x))`. Первый слой нашей сети имеет размер (10, 16), значит, каждое наблюдение первого слоя, полученное из входного слоя, будет иметь 10 значений признаков. Поскольку данная нейронная сеть создавалась для бинарной классификации, выходной слой содержит всего один нейрон с сигмоидальной функцией активации, и на выходе мы получаем значения от 0 до 1 (вероятность принадлежности наблюдения к классу 1).

Дополнительные материалы для чтения

- PyTorch tutorial: Build the Neural Network ("Руководство по PyTorch: Создание нейронной сети"; **https://oreil.ly/iT8iv**).
- Loss functions for classification ("Классификация с помощью функции потерь", Википедия; **https://oreil.ly/4bPXv**).
- Katarzyna Janocha, Wojciech Marian Czarnecki "On Loss Functions for Deep Neural Networks in Classification" (Катажина Яноха, Войцех Мариан Чарнецкий "О функциях потерь при глубоком обучении сетей для классификации"; **https://oreil.ly/pplP-**).

21.4. Обучение бинарного классификатора

Задача

Вы хотите обучить нейронную сеть для бинарной классификации.

Решение

При помощи PyTorch создадим и обучим нейронную сеть с прямой связью:

```
# Импортировать библиотеки
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы
features, target = make_classification(n_classes=2, n_features=10,
    n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)
```

```
# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()

# Определить функцию потерь и оптимизацию
criterion = nn.BCELoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)
```

```
# Обучить нейронную сеть
epochs = 3
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
    print("Epoch:", epoch+1, "\tLoss:", loss.item())

# Оценить нейронную сеть
with torch.no_grad():
    output = network(x_test)
    test_loss = criterion(output, y_test)
    test_accuracy = (output.round() == y_test).float().mean()
    print("Test Loss:", test_loss.item(), "\tTest Accuracy:",
        test_accuracy.item())
```

```
Epoch: 1        Loss: 0.19006995856761932
Epoch: 2        Loss: 0.14092367887496948
Epoch: 3        Loss: 0.03935524448752403
Test Loss: 0.06877756118774414  Test Accuracy: 0.9700000286102295
```

Обсуждение

В *рецепте 21.3* мы обсуждали процесс создания нейронной сети при помощи последовательной модели PyTorch. В данном примере мы обучаем нейронную сеть на основе 10 признаков и 1000 наблюдений псевдоклассификации, созданной функцией `make_classification` библиотеки scikit-learn.

Используемая нами нейронная сеть аналогична созданной в *рецепте 21.3* (обратитесь к данному рецепту для более получения подробного объяснения). Отличие заключается в том, что мы только создали нейронную сеть, без последующего обучения.

В конце мы использовали `with torch.no_grad()` для оценки нашей сети. Это означает, что нам не следует производить вычисления градиентов для любых операций с тензорами в данной части кода. Так как градиенты используются лишь на этапе обучения модели, нам незачем хранить новые градиенты для операций за ее пределами (например, прогнозирование или оценка).

При помощи переменной `epochs` мы определяем количество эпох, используемых в процессе обучения. `batch_size` означает количество наблюдений, которые пройдут через нейронную сеть до обновления параметров.

Затем мы итеративно запускаем несколько эпох, используя метод forward для прямого прохода данных через сеть и последующего обратного прохода для обновления градиентов. В результате мы получаем обученную модель.

21.5. Обучение многоклассового классификатора

Задача

Вы хотите обучить нейронную сеть для многоклассовой классификации.

Решение

При помощи инструментов PyTorch создадим нейронную сеть с прямой связью с многопеременной логистической функцией в выходном слое:

```
# Импортировать библиотеки
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

N_CLASSES=3
EPOCHS=3

# Создать обучающий и тестовый наборы
features, target = make_classification(n_classes=N_CLASSES, n_informative=9,
    n_redundant=0, n_features=10, n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Задать случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.nn.functional.one_hot(torch.from_numpy(target_train).long(),
    num_classes=N_CLASSES).float()
```

```
x_test = torch.from_numpy(features_test).float()
y_test = torch.nn.functional.one_hot(torch.from_numpy(target_test).long(),
    num_classes=N_CLASSES).float()

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,3),
            torch.nn.Softmax()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()

# Определить функцию потерь и оптимизацию
criterion = nn.CrossEntropyLoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)

# Обучить нейронную сеть
for epoch in range(EPOCHS):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
```

```
        loss.backward()
        optimizer.step()
    print("Epoch:", epoch+1, "\tLoss:", loss.item())

# Оценить нейронную сеть
with torch.no_grad():
    output = network(x_test)
    test_loss = criterion(output, y_test)
    test_accuracy = (output.round() == y_test).float().mean()
    print("Test Loss:", test_loss.item(), "\tTest Accuracy:",
        test_accuracy.item())
```

```
Epoch: 1        Loss: 0.8022041916847229
Epoch: 2        Loss: 0.775616466999054
Epoch: 3        Loss: 0.7751263380050659
Test Loss: 0.8105319142341614   Test Accuracy: 0.8199999928474426
```

Обсуждение

В данном случае мы создали нейронную сеть, схожую с бинарным классификатором из предыдущего рецепта, но с некоторыми важными отличиями. Например, мы добавили переменную `N_CLASSES=3` для создаваемых данных. Помимо этого мы использовали `nn.CrossEntropyLoss()` с быстрым кодированием целевых показателей для управления многоклассовой классификацией. Мы, в частности, воспользовались функцией `torch.nn.functional.one_hot`, в результате чего получили двоичный массив, где позиция `1.` означает класс имеющегося наблюдения:

```
# Вывести целевую матрицу
y_train
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [1., 0., 0.],
        ...,
        [0., 1., 0.],
        [1., 0., 0.],
        [0., 0., 1.]])
```

Поскольку это многоклассовая классификация, мы использовали выходной слой со значением 3 (для каждого класса) и многопеременной логистической (softmax) функцией активации. Функция возвращает массив с тремя значениями, дающими в сумме 1. Эти три значения означают вероятность отнесения наблюдения к какому-либо из трех классов. Как было отмечено ранее в рецепте, мы использовали категориальную функцию потерь перекрестной энтропии, подходящую для многоклассовой классификации: `nn.CrossEntropyLoss()`.

21.6. Обучение регрессора

Задача

Вы хотите обучить нейронную сеть для регрессии.

Решение

При помощи инструментов PyTorch создадим нейронную сеть с прямой связью с одним нейроном без функции активации:

```
# Импортировать библиотеки
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

EPOCHS=5

# Создать обучающий и тестовый наборы
features, target = make_regression(n_features=10, n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Задать случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1,1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1,1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
```

```
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,1),
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()

# Определить функцию потерь и оптимизацию
criterion = nn.MSELoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)

# Обучить нейронную сеть
for epoch in range(EPOCHS):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
    print("Epoch:", epoch+1, "\tLoss:", loss.item())

# Оценить нейронную сеть
with torch.no_grad():
    output = network(x_test)
    test_loss = float(criterion(output, y_test))
    print("Test MSE:", test_loss)
```

```
Epoch: 1        Loss: 10764.02734375
Epoch: 2        Loss: 1356.510009765625
Epoch: 3        Loss: 504.9664306640625
Epoch: 4        Loss: 199.11314392089844
Epoch: 5        Loss: 191.20834350585938
Test MSE: 162.24497985839844
```

Обсуждение

В зависимости от наших задач мы можем создать нейронную сеть, которая вместо вероятностей классов будет предсказывать непрерывные значения. В случае нашего бинарного классификатора (*см. рецепт 21.4*) мы использовали выходной слой с одним нейроном и сигмоидальной функцией активации для прогнозирования вероятности отнесения наблюдения к классу 1. Важно отметить, что выходные данные сигмоидальной функции активации имели значения от 0 до 1. Если же мы откажемся от функции активации, то выходные данные будут иметь непрерывные значения.

Более того, при обучении регрессии мы должны использовать подходящую функцию потерь и метрику оценки, например среднеквадратическую ошибку (mean square error):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2,$$

где n — количество наблюдений; y_i — истинное значение целевого показателя y, которое мы пытаемся спрогнозировать для наблюдения i; $\hat{y}_i$ — прогнозируемое значение модели для y_i.

Наконец, мы не стали стандартизировать признаки, поскольку создали искусственные данные при помощи метода `make_regression` библиотеки scikit-learn. Однако стоит отметить, что стандартизация необходима почти всегда при работе с реальными данными.

21.7. Создание прогнозов

Задача

Вы хотите использовать нейронную сеть для создания прогнозов.

Решение

При помощи PyTorch создадим нейронную сеть с прямой связью и при помощи функции `forward` построим прогноз:

```
# Импортировать библиотеки
import torch
```

```
import torch.nn as nn
import numpy as np

from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features, target = make_classification(n_classes=2, n_features=10,
    n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Установить целое начальное значение
torch.manual_seed(0) np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()
```

```
# Определить функцию потерь и оптимизацию
criterion = nn.BCELoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)

# Обучить нейронную сеть
epochs = 3
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
    print("Epoch:", epoch+1, "\tLoss:", loss.item())

# Оценить нейронную сеть
with torch.no_grad():
    predicted_class = network.forward(x_train).round()

predicted_class[0]
```

```
Epoch: 1        Loss: 0.19006995856761932
Epoch: 2        Loss: 0.14092367887496948
Epoch: 3        Loss: 0.03935524448752403
tensor([1.])
```

Обсуждение

Фреймворк PyTorch отлично подходит для создания прогнозов. После завершения обучения нейронной сети мы можем воспользоваться методом `forward` (уже примененным в процессе обучения), который принимает набор признаков в качестве выходных данных и отправляет его через всю сеть. В нашем примере мы создавали нейронную сеть для бинарной классификации, поэтому получаемые значения являлись вероятностью принадлежности к классу 1. Если прогнозируемое значение на-

блюдения стремилось к 1, то оно причислялось к классу 1, тогда как при приближении к 0 повышалась вероятность причисления к классу 0. Для преобразования этих значений в нули и единицы для бинарной классификации мы использовали метод round.

21.8. Визуализация истории обучения

Задача

Вы хотите определить золотую середину для функции потерь и/или показателя правильности нейронной сети.

Решение

Используем библиотеку Matplotlib для визуализации потерь тестового и обучающего наборов данных в каждой эпохе (рис. 21.1):

```
# Загрузить библиотеки
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

import numpy as np
import matplotlib.pyplot as plt

# Создать обучающий и тестовый наборы
features, target = make_classification(n_classes=2, n_features=10, n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)
```

```
# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()

# Определить функцию потерь и оптимизацию
criterion = nn.BCELoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)

# Обучить нейронную сеть
epochs = 8
train_losses = []
test_losses = []
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
```

```
    with torch.no_grad():
        train_output = network(x_train)
        train_loss = criterion(output, target)
        train_losses.append(train_loss.item())

        test_output = network(x_test)
        test_loss = criterion(test_output, y_test)
        test_losses.append(test_loss.item())

# Визуализировать историю потерь
epochs = range(0, epochs)
plt.plot(epochs, train_losses, "r--")
plt.plot(epochs, test_losses, "b-")
plt.legend(["Потери в обучающих данных", "Потери в тестовых данных"])
plt.xlabel("Эпоха")
plt.ylabel("Потери")
plt.show()
```

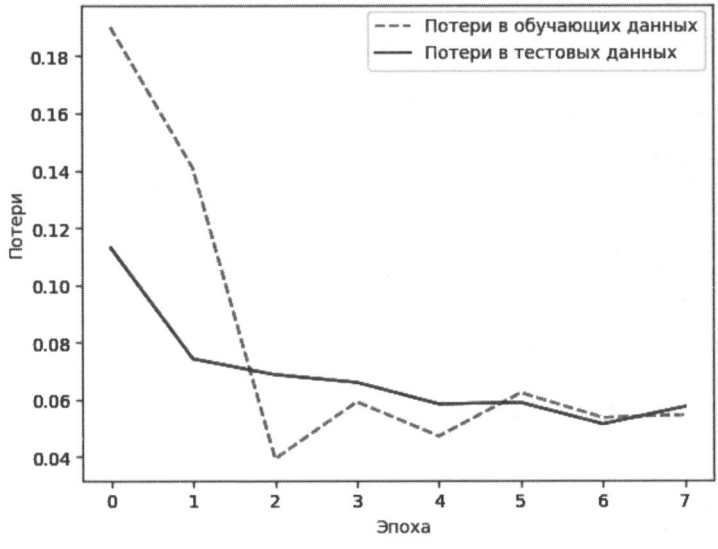

Рис. 21.1

Обсуждение

Созданная с нуля нейронная сеть обладает низкими показателями качества. Благодаря тренировке этой сети на обучающем наборе данных мы можем постепенно уменьшить количество ошибок модели, возникающих при работе как с обучающим, так и тестовым наборами данных. Однако в определенный момент нейронная сеть может начать "запоминать" данные обучающего набора, что приводит к пере-

обучению. В результате количество ошибок снижается при работе с обучающим набором, но увеличивается при работе с тестовым набором. Таким образом, следует найти золотую середину — стремиться к минимальному количеству ошибок при работе с тестовым набором (эти ошибки нас интересуют в большей степени). В качестве одного из решений мы можем визуализировать потери обучающего и тестового наборов в каждой эпохе. Обратите внимание, что наименьший уровень ошибок тестового набора находится примерно на эпохе 6, а далее их количество увеличивается при неизменности ошибок обучающего набора. Другими словами, на этом этапе модель начинает переобучаться.

21.9. Регуляризация весов для снижения переобучения

Задача

Вы хотите уменьшить эффект переобучения при помощи регуляризации весов вашей нейронной сети.

Решение

Воспользуемся установкой штрафов для параметров нейронной сети или регуляризацией весов:

```
# Импортировать библиотеки
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features, target = make_classification(n_classes=2, n_features=10,
    n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
```

```
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module)
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()

# Определить функцию потерь и оптимизацию
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(network.parameters(), lr=1e-4, weight_decay=1e-5)

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)

# Обучить нейронную сеть
epochs = 100
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
```

```
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

# Оценить нейронную сеть
with torch.no_grad():
    output = network(x_test)
    test_loss = criterion(output, y_test)
    test_accuracy = (output.round() == y_test).float().mean()
    print("Test Loss:", test_loss.item(), "\tTest Accuracy:",
        test_accuracy.item())
```

```
Test Loss: 0.4030887186527252  Test Accuracy: 0.9599999785423279
```

Обсуждение

В качестве одного из способов предотвращения переобучения нейронных сетей можно использовать штрафы для параметров (т. е. весов) нейронной сети, которые принимают небольшие значения и создают тем самым простую модель, менее подверженную переобучению. Этот метод называется регуляризацией (weight regularization), или сокращением веса (weight decay). Суть регуляризации весов заключается в добавлении штрафов к функции потерь, например, нормы L2.

В PyTorch мы можем настроить регуляризацию весов при помощи добавления `weight_decay=1e-5` в непосредственный оптимизатор. В нашем примере `1e-5` означает величину штрафа для больших значений параметра. В PyTorch регуляризация L2 применяется к значениям больше 0.

21.10. Снижение переобучения через раннюю остановку

Задача

Вы хотите снизить переобучение при помощи остановки процесса обучения в момент расхождения обучающего и тестового наборов данных.

Решение

Используем библиотеку Lightning из PyTorch для добавления стратегии *ранней остановки*:

```
# Импортировать библиотеки
import torch
import torch.nn as nn
```

```
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
import lightning as pl
from lightning.pytorch.callbacks.early_stopping import EarlyStopping
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features, target = make_classification(n_classes=2, n_features=10,
    n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x class LightningNetwork(pl.LightningModule):
```

```
    def __init__(self, network):
        super().__init__()
        self.network = network
        self.criterion = nn.BCELoss()
        self.metric = nn.functional.binary_cross_entropy

    def training_step(self, batch, batch_idx):
        # Функция training_step определяет цикл обучения
        data, target = batch
        output = self.network(data)
        loss = self.criterion(output, target)
        self.log("val_loss", loss)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=1e-3)

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Инициализировать нейронную сеть
network = LightningNetwork(SimpleNeuralNet())

# Обучить нейронную сеть
trainer = pl.Trainer(callbacks=[EarlyStopping(monitor="val_loss", mode="min",
    patience=3)], max_epochs=1000)
trainer.fit(model=network, train_dataloaders=train_loader)
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name      | Type            | Params
---------------------------------------------
0 | network   | SimpleNeuralNet | 465
1 | criterion | BCELoss         | 0
---------------------------------------------
```

```
465       Trainable params
0         Non-trainable params
465       Total params
0.002     Total estimated model params size (MB)
/usr/local/lib/python3.10/site-packages/lightning/pytorch/trainer/
    connectors/data_connector.py:224: PossibleUserWarning:
    The dataloader, train_dataloader, does not have many workers which
    may be a bottleneck. Consider increasing the value of the `num_workers`
    argument (try 7 which is the number of cpus on this machine)
    in the `DataLoader` init to improve performance.
  rank_zero_warn(
/usr/local/lib/python3.10/site-packages/lightning/pytorch/trainer/
    trainer.py:1609: PossibleUserWarning: The number of training batches (9)
    is smaller than the logging interval Trainer(log_every_n_steps=50).     Set a
lower value for log_every_n_steps if you want to see logs
    for the training epoch.
  rank_zero_warn(
Epoch 23: 100%|██████████| 9/9 [00:00<00:00, 59.29it/s, loss=0.147, v_num=5]
```

Обсуждение

Ранее в *рецепте 21.8* мы уже описали, что обычно в процессе нескольких первых эпох обучения наблюдается снижение ошибок как обучающего, так и тренировочного наборов, однако затем нейронная сеть в определенный момент начинает "запоминать" данные обучающего набора, в результате чего снижение ошибок обучающего набора продолжается при увеличении ошибок тестового набора. Одним из наиболее распространенных и надежных методов, позволяющих избежать переобучения, будет отслеживание процесса обучения и его остановка при увеличении количества ошибок тестового множества. Это называется *ранней остановкой* (early stopping).

При работе с PyTorch ранняя остановка может быть реализована в виде функции обратного вызова. Функции обратного вызова можно применять на определенных этапах процесса обучения, например в конце каждой эпохи. Поскольку класс ранней остановки не определен в PyTorch по умолчанию, мы использовали готовое решение — популярную библиотеку Lightning (известную как PyTorch Lightning). Модель PyTorch Lightning представляет собой высокоуровневую библиотеку для PyTorch, которая содержит множество полезных особенностей. В нашем примере мы добавили метод `EarlyStopping(monitor="val_loss", mode="min", patience=3)` из PyTorch Lightning для запуска отслеживания потерь тестового набора (валидации) в каждой эпохе и остановки обучения при отсутствии улучшений на протяжении трех эпох (по умолчанию).

Если же мы не добавим функцию обратного вызова `EarlyStopping`, то модель пройдет без остановки все 1000 возможных эпох обучения:

```
# Обучить нейронную сеть
trainer = pl.Trainer(max_epochs=1000)
trainer.fit(model=network, train_dataloaders=train_loader)
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs


  | Name      | Type            | Params
---------------------------------------------
0 | network   | SimpleNeuralNet | 465
1 | criterion | BCELoss         | 0
---------------------------------------------
465       Trainable params
0         Non-trainable params
465       Total params
0.002     Total estimated model params size (MB)
Epoch 999: 100%|██████████| 9/9 [00:01<00:00,  7.95it/s, loss=0.00188, v_num=6]
`Trainer.fit` stopped: `max_epochs=1000` reached. Epoch 999: 100%|██████████| 9/9
[00:01<00:00,  7.80it/s, loss=0.00188, v_num=6]
```

21.11. Уменьшение эффекта переобучения методом исключения

Задача

Вам нужно уменьшить эффект переобучения.

Решение

Воспользуемся методом исключения (dropout), чтобы добавить шум в архитектуру нашей нейронной сети:

```
# Загрузить библиотеку
import torch
import torch.nn as nn
import numpy as np
```

```
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features, target = make_classification(n_classes=2, n_features=10,
    n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Dropout(0.1), # Удалить 10% нейронов
            torch.nn.Sigmoid(),
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()
```

```
# Определить функцию потерь и оптимизацию
criterion = nn.BCELoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)

# Обучить нейронную сеть
epochs = 3
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
    print("Epoch:", epoch+1, "\tLoss:", loss.item())

# Оценить нейронную сеть
with torch.no_grad():
    output = network(x_test)
    test_loss = criterion(output, y_test)
    test_accuracy = (output.round() == y_test).float().mean()
    print("Test Loss:", test_loss.item(), "\tTest Accuracy:",
        test_accuracy.item())
```

```
Epoch: 1        Loss: 0.18791493773460388
Epoch: 2        Loss: 0.17331615090370178
Epoch: 3        Loss: 0.1384529024362564
Test Loss: 0.12702330946922302  Test Accuracy: 0.9100000262260437
```

Обсуждение

Метод исключения, или *прореживания* (dropout), часто используется для регуляризации небольших нейронных сетей. Применение этого подхода подразумевает, что каждый раз при создании группы наблюдений для обучения происходит умножение части нейронов одного или нескольких слоев на ноль (т. е. удаление). При та-

кой конфигурации каждая группа обучается при помощи одной нейронной сети (т. е. одинаковых параметров), но с немного отличающимися версиями *архитектуры* сети.

Эффективность метода достигается за счет непрерывного и случайного исключения нейронов каждого блока, из-за чего при обучении нейронов используются параметры, которые могут работать с различными вариантами архитектуры нейронной сети. Другими словами, они могут сопротивляться распаду (т. е. шуму) в других скрытых нейронах, что позволяет предотвратить простое запоминание данных обучающего множества.

Мы можем добавить метод исключения как в скрытые, так и во входные слои. При выпадении входного слоя значения признаков для этого блока не попадают в нейронную сеть.

В PyTorch метод исключения вводится добавлением слоя `nn.Dropout` в архитектуру сети. Каждый слой `nn.Dropout` исключает заданные пользователем гиперпараметры нейронов в каждой группе предыдущего слоя.

21.12. Сохранение прогресса обучения модели

Задача

Ввиду длительности процесса обучения нейронной сети вы хотите сохранить текущий прогресс на случай его прерывания.

Решение

Используем функцию `torch.save` для сохранения модели после завершения эпохи:

```
# Загрузить библиотеки
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features, target = make_classification(n_classes=2, n_features=10,
    n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)
```

```
# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Dropout(0.1), # Удалить 10% нейронов
            torch.nn.Sigmoid(),
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()

# Определить функцию потерь и оптимизацию
criterion = nn.BCELoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)
```

```
# Обучить нейронную сеть
epochs = 5
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        # Сохранить модель после завершения эпохи
        torch.save(
            {
                'epoch': epoch,
                'model_state_dict': network.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'loss': loss,
            },
            "model.pt"
        )
    print("Epoch:", epoch+1, "\tLoss:", loss.item())
```

```
Epoch: 1        Loss: 0.18791493773460388
Epoch: 2        Loss: 0.17331615090370178
Epoch: 3        Loss: 0.1384529024362564
Epoch: 4        Loss: 0.1435958743095398
Epoch: 5        Loss: 0.17967987060546875
```

Обсуждение

При работе с реальными данными процесс обучения нейронных сетей может растягиваться на часы и даже дни. В этот момент может случаться многое: отключение питания компьютера, остановка сервера или рассеянность студента, потерявшего свой ноутбук.

Для того чтобы смягчить последствия этой проблемы, мы можем сохранить модель после завершения эпохи, используя `torch.save`. В частности, мы указываем в качестве второго аргумента функции `torch.save` путь к model.pt для сохранения модели после каждой эпохи. Если мы добавляем только одно имя файла (например, model.pt), то он будет перезаписываться при каждом сохранении.

Мы также можем добавить дополнительное правило сохранения модели, например, после нескольких эпох, только при снижении потерь и т. д. При помощи PyTorch

Lightning мы даже можем совместить этот подход с уже изученной ранней остановкой, чтобы модель сохранялась вне зависимости от момента остановки обучения.

21.13. Настройка нейронных сетей

Задача

Вы хотите настроить автоматический отбор лучших гиперпараметров для вашей нейронной сети.

Решение

Используем библиотеку ray для настройки в PyTorch:

```
# Загрузить библиотеки
from functools import partial
import numpy as np
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim import RMSprop
from torch.utils.data import random_split, DataLoader, TensorDataset
from ray import tune
from ray.tune import CLIReporter
from ray.tune.schedulers import ASHAScheduler
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features, target = make_classification(n_classes=2, n_features=10,
   n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
```

```
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self, layer_size_1=10, layer_size_2=10):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, layer_size_1),
            torch.nn.ReLU(),
            torch.nn.Linear(layer_size_1, layer_size_2),
            torch.nn.ReLU(),
            torch.nn.Linear(layer_size_2, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x

config = {
    "layer_size_1": tune.sample_from(lambda _: 2 ** np.random.randint(2, 9)),
    "layer_size_2": tune.sample_from(lambda _: 2 ** np.random.randint(2, 9)),
    "lr": tune.loguniform(1e-4, 1e-1),
}

scheduler = ASHAScheduler(
    metric="loss",
    mode="min",
    max_t=1000,
    grace_period=1,
    reduction_factor=2
)

reporter = CLIReporter(
    parameter_columns=["layer_size_1", "layer_size_2", "lr"],
    metric_columns=["loss"]
)

# Обучить нейронную сеть
def train_model(config, epochs=3):
    network = SimpleNeuralNet(config["layer_size_1"], config["layer_size_2"])
```

```
    criterion = nn.BCELoss()
    optimizer = optim.SGD(network.parameters(), lr=config["lr"], momentum=0.9)

    train_data = TensorDataset(x_train, y_train)
    train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

    # Скомпилировать модель при помощи оптимизатора torch 2.0
    network = torch.compile(network)

    for epoch in range(epochs):
        for batch_idx, (data, target) in enumerate(train_loader):
            optimizer.zero_grad()
            output = network(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            tune.report(loss=(loss.item()))

result = tune.run(
    train_model,
    resources_per_trial={"cpu": 2},
    config=config,
    num_samples=1,
    scheduler=scheduler,
    progress_reporter=reporter
)

best_trial = result.get_best_trial("loss", "min", "last")
print("Best trial config: {}".format(best_trial.config))
print("Best trial final validation loss: {}".format(
    best_trial.last_result["loss"]))

best_trained_model = SimpleNeuralNet(best_trial.config["layer_size_1"],
    best_trial.config["layer_size_2"])
```

```
== Status ==
Current time: 2023-03-05 23:31:33 (running for 00:00:00.07)
Memory usage on this node: 1.7/15.6 GiB
Using AsyncHyperBand: num_stopped=0
```

```
Bracket: Iter 512.000: None | Iter 256.000: None | Iter 128.000: None |
    Iter 64.000: None | Iter 32.000: None | Iter 16.000: None |
    Iter 8.000: None | Iter 4.000: None | Iter 2.000: None |
    Iter 1.000: None
Resources requested: 2.0/7 CPUs, 0/0 GPUs, 0.0/8.95 GiB heap,
    0.0/4.48 GiB objects
Result logdir: /root/ray_results/train_model_2023-03-05_23-31-33
Number of trials: 1/1 (1 RUNNING)
...
```

Обсуждение

Ранее в *рецептах 12.1* и *12.2* мы разобрали способы поиска лучших гиперпараметров посредством выбора моделей в scikit-learn. Хотя мы можем использовать подход scikit-learn для нейронных сетей, при помощи специального API библиотеки ray мы можем проводить эксперименты как с CPU, так и GPU.

Нам следует внимательно подбирать гиперпараметры модели, т. к. они крайне важны. При этом у нас могут быть весьма ограничены время и ресурсы на проведение экспериментов для поиска гиперпараметров. По этой причине нам в некоторых случаях может пригодиться автоматическая настройка гиперпараметров нашей нейронной сети, хотя это не избавляет от всех проблем.

В нашем примере мы искали различные параметры в соответствии с размерами слоев и коэффициентом скорости обучения оптимизатора. Аргумент `best_trial.config` выводит параметры конфигурации ray с лучшими результатами и наименьшими потерями.

21.14. Визуализация нейронной сети

Задача

Вы хотите быстро визуализировать архитектуру нейронной сети.

Решение

Используем функцию `make_dot` из библиотеки torchviz:

```
# Загрузить библиотеку
import torch
import torch.nn as nn
import numpy as np
```

```
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from torchviz import make_dot
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features, target = make_classification(n_classes=2, n_features=10,
    n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи Sequential
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.sequential(x)
        return x
```

```
# Инициализировать нейронную сеть
network = SimpleNeuralNet()
# Определить функцию потерь и оптимизацию
criterion = nn.BCELoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)

# Обучить нейронную сеть
epochs = 3
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

make_dot(output.detach(), params=dict(
    list(
        network.named_parameters()
        )
      )
    ).render(
        "simple_neural_network",
        format="png"
)
'simple_neural_network.png'
```

Если мы откроем полученное изображение, то увидим граф, подобный представленному на рис. 21.2.

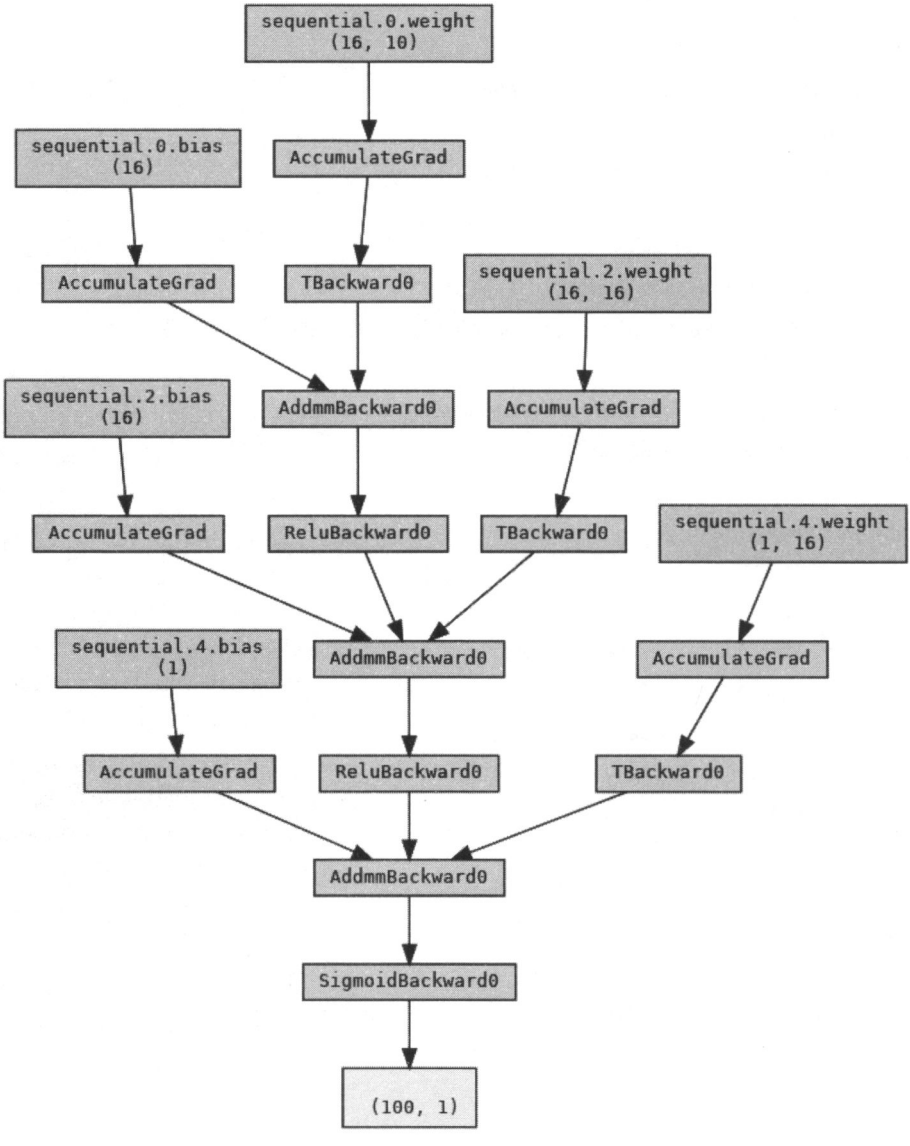

Рис. 21.2

Обсуждение

В библиотеке torchviz представлены простые функции для быстрой визуализации нейронных сетей и их выгрузки в формат изображения.

ГЛАВА 22

Нейронные сети с неструктурированными данными

Введение

В предыдущей главе мы использовали нейронные сети для обработки *структурированных данных*, например табличных. При этом стремительный прогресс за последние несколько лет во многом обусловлен использованием нейронных сетей и глубокого обучения для обработки *неструктурированных данных*, таких как тексты или изображения. Работа с неструктурированными наборами данных в некоторой степени отличается от работы со структурированными источниками.

Глубокое обучение особенно эффективно при работе с неструктурированными данными, где "классические" методы машинного обучения (например, деревья и бустинг) в большинстве случаев не могут охватить всю сложность и особенности текстовых данных, аудиоматериалов, изображений, видео и т. д. В этой главе мы научимся обрабатывать конкретно текстовые данные и изображения с помощью глубокого обучения.

Работа с текстами и изображениями в области контролируемого обучения включает наличие подзадач и различных "типов" обучения. Вот несколько примеров (хотя это, разумеется, неполный список):

- классификация текстов и изображений (пример: классификация изображения по принципу наличия или отсутствия хот-дога);
- трансферное обучение (пример: использование предобученной контекстуальной модели вроде BERT и ее тонкая настройка для определения спама);
- обнаружение объектов (пример: идентификация и классификация конкретных объектов на изображении);
- генеративные модели (пример: модели создания текста на основе ввода вроде моделей GPT).

С ростом популярности и повсеместного внедрения глубокого обучения становятся более доступными как открытые, так и коммерческие решения. В этой главе мы рассмотрим несколько основных библиотек, которые станут хорошей отправной точкой в вопросе решения задач с использованием глубокого обучения. Мы, в ча-

стности, будем использовать основанные на Python библиотеки PyTorch, torchvision и transformers для решения набора задач по обработке текстов и изображений на основе машинного обучения.

22.1. Обучение нейронной сети для классификации изображений

Задача

Вам необходимо обучить нейронную сеть для классификации изображений.

Решение

Используем сверточную нейронную сеть из PyTorch:

```
# Импортировать библиотеки
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms

# Определить архитектуру сверточной нейронной сети
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(64 * 14 * 14, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.relu(self.conv2(x))
        x = nn.functional.max_pool2d(self.dropout1(x), 2)
        x = torch.flatten(x, 1)
        x = nn.functional.relu(self.fc1(self.dropout2(x)))
        x = self.fc2(x)
        return nn.functional.log_softmax(x, dim=1)
```

```
# Указать используемое устройство
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Определить этапы обработки данных
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Загрузить набор данных MNIST
train_dataset = datasets.MNIST('./data', train=True, download=True,
    transform=transform)
test_dataset = datasets.MNIST('./data', train=False, transform=transform)

# Создать загрузчики данных
batch_size = 64
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
    shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
    shuffle=True)

# Инициализировать модель и оптимизацию
model = Net().to(device)
optimizer = optim.Adam(model.parameters())

# Скомпилировать модель при помощи оптимизатора torch 2.0
model = torch.compile(model)

# Настроить цикл обучения
model.train()
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    output = model(data)
    loss = nn.functional.nll_loss(output, target)
    loss.backward()
    optimizer.step()

# Настроить тестовый цикл
model.eval()
test_loss = 0
```

```
correct = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        output = model(data)

        # узнать индекс максимального значения логарифма вероятности
        test_loss += nn.functional.nll_loss(
            output, target, reduction='sum'
        ).item()  # сложить потери групп
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)
```

Обсуждение

Сверточные нейронные сети (convolutional neural networks, CNN) используются, как правило, для задач распознавания изображений и компьютерного зрения. Они обычно состоят из слоев свертки (convolutional layers), пулинговых слоев (pooling layers) и полносвязного слоя (fully connected layer).

Слой свертки осуществляет запоминание важных признаков изображения, которые в дальнейшем используются для решения задач. Суть их работы заключается в наложении фильтра на определенную область изображения (размер свертки). Затем происходит обучение весов этого слоя для распознавания определенных, необходимых для классификации признаков изображения. Например, при обучении модели распознавания руки фильтр может научиться распознавать пальцы.

Пулинговый слой в основном служит для снижения размерности входных данных из предыдущего слоя. Здесь также используется фильтр, применяемый к части входных данных, но без активации. Вместо этого происходит уплотнение входных данных при помощи *максимального пулинга* (выбираются пикселы в фильтре с наибольшим значением) или *обычного пулинга* (выбираются средние пикселы входных данных).

Наконец, *полносвязный слой* может быть использован с подобием многопеременной логистической функции для создания задач бинарной классификации.

Дополнительные материалы для чтения

- Convolutional Neural Networks ("Сверточные нейронные сети"; **https://oreil.ly/HoO9g**).

22.2. Обучение нейронной сети для классификации текстов

Задача

Вам необходимо обучить нейронную сеть для классификации текстовых данных.

Решение

Воспользуемся нейронной сетью PyTorch, первый слой которой равен размеру нашего словаря:

```
# Импортировать библиотеки
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Загрузить набор данных 20newsgroups
cats = ['alt.atheism', 'sci.space']
newsgroups_data = fetch_20newsgroups(subset='all', shuffle=True,
    random_state=42, categories=cats)

# Разделить набор данных на обучающий и тестовый
X_train, X_test, y_train, y_test = train_test_split(newsgroups_data.data,
    newsgroups_data.target, test_size=0.2, random_state=42)

# Создать векторизацию текстовых данных методом мешка слов
vectorizer = CountVectorizer(stop_words='english')
X_train = vectorizer.fit_transform(X_train).toarray()
X_test = vectorizer.transform(X_test).toarray()

# Преобразовать данные в тензоры PyTorch
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.long)
```

```
# Определить модель
class TextClassifier(nn.Module):
    def __init__(self, num_classes):
        super(TextClassifier, self).__init__()
        self.fc1 = nn.Linear(X_train.shape[1], 128)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return nn.functional.log_softmax(x, dim=1)

# Описать модель, определить функцию потерь и оптимизацию
model = TextClassifier(num_classes=len(cats))
loss_function = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Скомпилировать модель при помощи оптимизатора torch 2.0
model = torch.compile(model)

# Обучить модель
num_epochs = 1
batch_size = 10
num_batches = len(X_train) // batch_size
for epoch in range(num_epochs):
    total_loss = 0.0
    for i in range(num_batches):
        # Подготовить входные и целевые данные для текущей группы
        start_idx = i * batch_size
        end_idx = (i + 1) * batch_size
        inputs = X_train[start_idx:end_idx]
        targets = y_train[start_idx:end_idx]

        # Указать нулевые градиенты для оптимизатора
        optimizer.zero_grad()

        # Пропустить данные через модель и рассчитать потери
        outputs = model(inputs)
        loss = loss_function(outputs, targets)

        # Обратно пропустить данные через модель и обновить параметры
        loss.backward()
        optimizer.step()
```

```
        # Обновить общий показатель потерь для эпохи
        total_loss += loss.item()

    # Рассчитать параметр правильности для эпохи на основе тестового набора
    test_outputs = model(X_test)
    test_predictions = torch.argmax(test_outputs, dim=1)
    test_accuracy = accuracy_score(y_test, test_predictions)

    # Вывести номер эпохи, среднее значение потерь и правильность тестового набора
    print(f"Epoch: {epoch+1}, Loss: {total_loss/num_batches}, Test Accuracy:"
        "{test_accuracy}")
```

Обсуждение

В отличие от изображений, текстовые данные — в сущности, нечисловые. Нам необходимо перед обучением модели преобразовать текст в числовой формат, который модель сможет использовать для определения слов и словосочетаний, наиболее существенных для классификации. В рассмотренном примере мы использовали класс `CountVectorizer` из scikit-learn для кодирования словаря в форме вектора, соответствующего размеру всего словаря, в котором каждое слово имеет определенный индекс, а его значение соответствует количеству случаев появления слова в некотором отрывке. В этом случае мы можем узнать размер словаря при помощи обучающего набора:

```
X_train.shape[1]
```

```
25150
```

Мы используем эти же значения в первом слое нейронной сети для определения размера входного слоя: `self.fc1 = nn.Linear(X_train.shape[1], 128)`. Это позволяет использовать в нашей сети *векторное представление слов* (word embeddings), полученных при помощи задач контролируемого обучения, пример которых можно найти в этом рецепте. Эта задача позволяет нам использовать векторное представление слов размерностью 128, хотя такое представление в первую очередь будет полезно для этой конкретной задачи и словаря.

22.3. Тонкая настройка предобученной модели для классификации изображений

Задача

Вы хотите получить модель классификации изображений на основе предобученной модели.

Решение

Используем библиотеку transformers вместе с torchvision для тонкой настройки предобученной модели с нашими данными:

```
# Импортировать библиотеки
import torch
from torchvision.transforms import(
    RandomResizedCrop, Compose, Normalize, ToTensor
     )
from transformers import Trainer, TrainingArguments, DefaultDataCollator
from transformers import ViTFeatureExtractor, ViTForImageClassification
from datasets import load_dataset, load_metric, Image

# Определить вспомогательную функцию для преобразования изображения в формат RGB
def transforms(examples):
    examples["pixel_values"] = [_transforms(img.convert("RGB")) for img in
        examples["image"]]
    del examples["image"]
    return examples

# Определить вспомогательную функцию для вычисления метрик
def compute_metrics(p):
    return metric.compute(predictions=np.argmax(p.predictions, axis=1),
        references=p.label_ids)

# Загрузить набор данных fashion_mnist
dataset = load_dataset("fashion_mnist")

# Добавить объект обработки из модели VIT
image_processor = ViTFeatureExtractor.from_pretrained(
    "google/vit-base-patch16-224-in21k"
)

# Настроить метки из набора данных
labels = dataset['train'].features['label'].names

# Загрузить предобученную модель
model = ViTForImageClassification.from_pretrained(
    "google/vit-base-patch16-224-in21k",
    num_labels=len(labels),
    id2label={str(i): c for i, c in enumerate(labels)},
    label2id={c: str(i) for i, c in enumerate(labels)}
)
```

```
# Определить сортировщик, нормализатор и преобразования
collate_fn = DefaultDataCollator()
normalize = Normalize(mean=image_processor.image_mean,
    std=image_processor.image_std)
size = (
    image_processor.size["shortest_edge"]
    if "shortest_edge" in image_processor.size
    else (image_processor.size["height"], image_processor.size["width"])
)
_transforms = Compose([RandomResizedCrop(size), ToTensor(), normalize])

# Загрузить набор данных для преобразования
dataset = dataset.with_transform(transforms)

# В качестве метрики использовать значение правильности
metric = load_metric("accuracy")

# Настроить аргументы для обучения
training_args = TrainingArguments(
    output_dir="fashion_mnist_model",
    remove_unused_columns=False,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=0.01,
    per_device_train_batch_size=16,
    gradient_accumulation_steps=4,
    per_device_eval_batch_size=16,
    num_train_epochs=1,
    warmup_ratio=0.1
    logging_steps=10,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
    push_to_hub=False,
)

# Настроить объект обучения
trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=collate_fn,
    compute_metrics=compute_metrics,
    train_dataset=dataset["train"],
```

```
    eval_dataset=dataset["test"],
    tokenizer=image_processor,
)

# Обучить модель, метрики журналов и сохранений
train_results = trainer.train()
trainer.save_model()
trainer.log_metrics("train", train_results.metrics)
trainer.save_metrics("train", train_results.metrics)
trainer.save_state()
```

Обсуждение

При работе с неструктурированными данными, например текстом или изображениями, мы очень часто начинаем с предварительного обучения модели на основе большого набора данных, особенно при отсутствии доступа к размеченным данным. С помощью представлений (embeddings) и другой информации из большой модели мы можем осуществить тонкую настройку собственной модели для новых задач без необходимости иметь большое количество размеченных данных. Кроме того, предобученная модель может содержать информацию, не включенную в обучающий набор данных, что положительно сказывается на общей производительности. Этот процесс называется *трансферным обучением*.

В данном примере мы загрузили веса из модели Google's ViT (Vision Transformer). Затем мы использовали библиотеку transformers для тонкой настройки под задачи классификации по аналогии с простым набором данных MNIST, состоящим из предметов одежды. Такой подход может быть использован для улучшения производительности любого набора данных для машинного зрения, а библиотека transformers предоставляет высокоуровневый интерфейс для тонкой настройки нашей модели на основе больших, заранее обученных моделей без необходимости написания огромного кода.

Дополнительные материалы для чтения

- Документация и сайт Hugging Face (**https://oreil.ly/5F3Rf**).

22.4. Тонкая настройка предобученной модели для классификации текста

Задача

Вы хотите обучить модель классификации текста на основе данных предобученной модели.

Решение

Воспользуемся библиотекой transformers:

```
# Импортировать библиотеки
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding
from transformers import (
    AutoModelForSequenceClassification, TrainingArguments, Trainer
    )
import evaluate
import numpy as np

# Загрузить набор данных imdb
imdb = load_dataset("imdb")

# Создать лексический анализатор и сортировщик
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Создать токены для набора данных imdb
tokenized_imdb = imdb.map(
    lambda example: tokenizer(
        example["text"], padding="max_length", truncation=True
    ),
    batched=True,
)

# Указать метрику правильности
accuracy = evaluate.load("accuracy")

# Определить вспомогательную функцию для создания метрик
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = np.argmax(predictions, axis=1)
    return accuracy.compute(predictions=predictions, references=labels)

# Создать словарь для привязки индексов к меткам и наоборот
id2label = {0: "NEGATIVE", 1: "POSITIVE"}
label2id = {"NEGATIVE": 0, "POSITIVE": 1}

# Загрузить предобученную модель
model = AutoModelForSequenceClassification.from_pretrained(
```

```
    "distilbert-base-uncased", num_labels=2, id2label=id2label,
        label2id=label2id
)

# Указать аргументы для обучения
training_args = TrainingArguments(
    output_dir="my_awesome_model",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=2,
    weight_decay=0.01,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
)

# Настроить объект обучения
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_imdb["train"],
    eval_dataset=tokenized_imdb["test"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

# Обучить модель
trainer.train()
```

Обсуждение

Предобученные языковые модели, как и модели для обработки изображений, содержат внушительный объем контекстных данных о языке, поскольку зачастую обучение происходит на основе огромного множества открытых источников из Интернета. При использовании предобученной модели в качестве основы мы фактически переносим слой классификации существующей сети в нашу собственную. Это позволяет нам изменять веса нейронной сети, обученной под конкретные задачи.

В нашем примере мы осуществили тонкую настройку модели DistilBERT, которая определяет принадлежность рецензий на фильмы на ресурсе IMDB к положительным (1) или отрицательным (0). В дополнение к весам нейронной сети, усвоенным из предыдущих задач обучения, предобученная модель DistilBERT имеет большой корпус из слов и контекстов для них. Благодаря трансферному обучению мы можем в полной мере использовать результаты проделанной ранее работы по обучению модели DistilBERT и перестроить их под нашу задачу, которая в данном случае заключается в классификации рецензий на фильмы.

Дополнительные материалы для чтения

- Text classification in transformers ("Классификация текстов при помощи библиотеки transformers"; **https://oreil.ly/uhrjI**).

ГЛАВА 23

Сохранение, загрузка и развертывание обученных моделей

Введение

На примере почти 200 рецептов в 22 главах мы разобрали базовые способы того, как при помощи машинного обучения можно создавать качественные предсказательные модели на основе сырых данных. Вместе с тем нам остается разобрать еще один аспект, без которого полученный опыт не будет полным — нужно научиться *работать* с нашими моделями, например, встраивать их в существующие приложения. Для этого мы должны уметь сохранять модели после обучения, загружать их для использования приложением и делать запросы к приложениям для получения прогнозов.

Модели машинного обучения обычно разворачиваются в виде простых веб-серверов и умеют принимать входные данные и возвращать прогнозы. Таким образом, модель становится доступной любому пользователю нейронной сети, и при помощи различных сервисов (например, UI, пользователи и т. д.) можно в режиме реального времени получать прогнозы, независимо от физического местонахождения. Хорошим примером такого использования машинного обучения будет поиск товаров на торговых интернет-площадках, где соответствующая модель обрабатывает данные о пользователях и списках товаров и возвращает вероятность совершения ими покупок из этих списков. Результаты поиска должны быть доступны в режиме реального времени через приложение для интернет-торговли, которое обрабатывает данные поисковых запросов пользователей для последующего предоставления результатов.

23.1. Сохранение и загрузка модели scikit-learn

Задача

У вас есть обученная модель scikit-learn, которую нужно сохранить и загрузить в другом месте.

Решение

Сохраним модель в виде файла pickle:

```
# Загрузить библиотеки
import joblib
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект классификации методом случайного леса
classifer = RandomForestClassifier()

# Обучить модель
model = classifer.fit(features, target)

# Сохранить модель в виде файла pickle
joblib.dump(model, "model.pkl") ['model.pkl']
```

После сохранения модели мы можем использовать инструменты scikit-learn в соответствующем приложении (например, веб-приложении) для ее загрузки:

```
# Загрузить модель из файла
classifer = joblib.load("model.pkl")
```

И создавать с ее помощью прогнозы:

```
# Создать новое наблюдение
new_observation = [[ 5.2,  3.2,  1.1,  0.1]]

# Спрогнозировать класс наблюдения
classifer.predict(new_observation)
array([0])
```

Обсуждение

Первый шаг внедрения модели — ее сохранение в виде файла, который будет доступен для загрузки другим приложением или рабочим процессом. Для этого подойдет файл в формате pickle — ориентированный на Python формат данных, который позволяет сериализовать объекты Python и записывать их в файл. Мы, в частности, воспользовались библиотекой joblib, которая позволяет выгружать большие массивы NumPy в формате pickle — распространенная задача при обучении моделей в scikit-learn.

В процессе сохранения моделей следует помнить, что полученные экземпляры могут быть совместимы не со всеми версиями scikit-learn; по этой причине будет не лишним указывать использованную при обучении версию scikit-learn в названии файла модели:

```
# Загрузить библиотеку
import sklearn

# Узнать версию scikit-learn
scikit_version = sklearn.__version__

# Сохранить модель в виде файла pickle
joblib.dump(model, "model_{version}.pkl".format(version=scikit_version))
['model_1.2.0.pkl']
```

23.2. Сохранение и загрузка модели TensorFlow

Задача

У вас есть обученная модель TensorFlow, которую нужно сохранить и загрузить в другом месте.

Решение

Сохраним модель при помощи функции `saved_model` из TensorFlow:

```
# Загрузить библиотеку
import numpy as np
from tensorflow import keras

# Установить случайное целое начальное значение
np.random.seed(0)

# Создать модель с одним скрытым слоем
input_layer = keras.Input(shape=(10,))
hidden_layer = keras.layers.Dense(10)(input_layer)
output_layer = keras.layers.Dense(1)(input_layer)
model = keras.Model(input_layer, output_layer)
model.compile(optimizer="adam", loss="mean_squared_error")

# Обучить модель
x_train = np.random.random((1000, 10))
y_train = np.random.random((1000, 1))
model.fit(x_train, y_train)
```

```
# Сохранить модель в папку save_model
model.save("saved_model")
```

```
32/32 [==============================] - 1s 8ms/step - loss: 0.2056
INFO:tensorflow:Assets written to: saved_model/assets
```

После этого мы можем добавить модель в отдельное приложение или продолжить ее обучение:

```
# Загрузить нейронную сеть
model =  keras.models.load_model("saved_model")
```

Обсуждение

Мы не так часто использовали библиотеку TensorFlow в наших примерах, однако будет полезно уметь сохранять и загружать модели на ее основе. В отличие от scikit-learn, где используется привычный в Python формат pickle, в TensorFlow практикуется собственный метод сохранения и загрузки моделей. При помощи функции `saved_model` создается каталог, в котором хранятся модели и вся информация, необходимая для загрузки модели и последующего создания прогнозов, в формате protocol buffer (как правило, с расширением файла .pb):

```
ls saved_model
assets fingerprint.pb keras_metadata.pb  saved_model.pb  variables
```

Мы не будем подробно разбирать здесь эту функцию; достаточно знать, что это стандартный способ сохранения, загрузки и развертывания моделей, обученных при помощи TensorFlow.

Дополнительные материалы для чтения

- Serialization and Saving Keras Models ("Сериализация и сохранение моделей Keras"; **https://oreil.ly/CDPvo**).
- TensorFlow Saved Model Format ("Формат сохранения моделей TensorFlow"; **https://oreil.ly/StpSL**).

23.3. Сохранение и загрузка модели PyTorch

Задача

У вас есть обученная модель PyTorch, которую нужно сохранить и загрузить в другом месте.

Решение

Воспользуемся функциями torch.save и torch.load:

```
# Загрузить библиотеки
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
from torch.optim import RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Создать обучающий и тестовый наборы данных
features, target = make_classification(n_classes=2, n_features=10,
    n_samples=1000)
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Установить случайное целое начальное значение
torch.manual_seed(0)
np.random.seed(0)

# Преобразовать данные в тензоры PyTorch
x_train = torch.from_numpy(features_train).float()
y_train = torch.from_numpy(target_train).float().view(-1, 1)
x_test = torch.from_numpy(features_test).float()
y_test = torch.from_numpy(target_test).float().view(-1, 1)

# Определить нейронную сеть при помощи `Sequential`
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Dropout(0.1), # Удалить 10% нейронов
            torch.nn.Sigmoid(),
        )
```

```
    def forward(self, x):
        x = self.sequential(x)
        return x

# Инициализировать нейронную сеть
network = SimpleNeuralNet()

# Определить функцию потерь и оптимизацию
criterion = nn.BCELoss()
optimizer = RMSprop(network.parameters())

# Определить загрузчик данных
train_data = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

# Скомпилировать модель при помощи оптимизатора torch 2.0
network = torch.compile(network)

# Обучить нейронную сеть
epochs = 5
for epoch in range(epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

# Сохранить модель после обучения
torch.save(
    {
        'epoch': epoch,
        'model_state_dict': network.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': loss,
    },
    "model.pt"
)

# Заново инициализировать нейронную сеть
network = SimpleNeuralNet()
```

```
state_dict = torch.load(
    "model.pt",
    map_location=torch.device('cpu')
    )["model_state_dict"]
network.load_state_dict(state_dict, strict=False)
network.eval()
```

```
SimpleNeuralNet(
  (sequential): Sequential(
    (0): Linear(in_features=10, out_features=16, bias=True)
    (1): ReLU()
    (2): Linear(in_features=16, out_features=16, bias=True)
    (3): ReLU()
    (4): Linear(in_features=16, out_features=1, bias=True)
    (5): Dropout(p=0.1, inplace=False)
    (6): Sigmoid()
  )
)
```

Обсуждение

Мы уже использовали подобный алгоритм в *главе 21* для сохранения промежуточных вариантов в процессе обучения. Теперь мы можем применить этот подход для загрузки модели обратно в память и последующего создания предсказаний. Файл model.pt, в который сохраняется модель, фактически является словарем, содержащим параметры этой модели. Для сохранения состояния модели в словарь ключей мы использовали функцию `model_state_dict`; чтобы обратно загрузить модель, нам нужно заново инициализировать нейронную сеть и загрузить сохраненное состояние при помощи функции `network.load_state_dict`.

Дополнительные материалы для чтения

- PyTorch tutorial: Saving and Loading Models ("Руководство по PyTorch: сохранение и загрузка моделей"; **https://oreil.ly/WO3X1**).

23.4. Развертывание моделей scikit-learn

Задача

Вы хотите открыть доступ к обученной модели scikit-learn через веб-сервер.

Решение

При помощи фреймворка Flask на Python создадим приложение для загрузки модели, которую мы обучили ранее в этой книге:

```
# Импортировать библиотеки
import joblib
from flask import Flask, request

# Инициализировать приложение на основе Flask
app = Flask(__name__)

# Загрузить модель с диска
model = joblib.load("model.pkl")

# Создать декоратор для получения прогнозов, который принимает данные JSON,
# создает прогнозы и возвращает их
@app.route("/predict", methods = ["POST"])
def predict():
    print(request.json)
    inputs = request.json["inputs"]
    prediction = model.predict(inputs)
    return {
        "prediction" : prediction.tolist()
     }

# Запустить приложение
if __name__ == "__main__":
    app.run()
```

Убедитесь, что у вас установлен Flask:

```
python3 -m pip install flask==2.2.3 joblib==1.2.0 scikit-learn==1.2.0
```

И затем запустите приложение:

```
python3 app.py
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
    Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Теперь мы можем создавать прогнозы через приложение и получать результаты при помощи команды curl:

```
curl -X POST http://127.0.0.1:5000/predict  -H 'Content-Type: application/json'
    -d '{"inputs":[[5.1, 3.5, 1.4, 0.2]]}'
{"prediction":[0]}
```

Обсуждение

В данном примере мы использовали популярную открытую библиотеку Flask для создания веб-приложений на языке Python. Мы определили путь /predict при помощи декоратора, который принимает данные JSON через запрос POST и возвращает словарь, содержащий предсказания. Хотя созданный таким образом сервер еще не является законченным продуктом (см. предупреждение Flask об использовании версии сервера для разработчиков), мы можем легко расширить и перенести его на более подходящий веб-фреймворк, получив тем самым готовый продукт.

23.5. Развертывание моделей TensorFlow

Задача

Вы хотите открыть доступ к обученной модели TensorFlow через веб-сервер.

Решение

Воспользуемся открытым сервером TensorFlow Serving и контейнером Docker:

```
docker run -p 8501:8501 -p 8500:8500 \
--mount type=bind,source=$(pwd)/saved_model,target=/models/saved_model/1 \
-e MODEL_NAME=saved_model -t tensorflow/serving
```

Обсуждение

TensorFlow Serving является сервером с открытым исходным кодом, оптимизированным под модели TensorFlow. Зная путь расположения модели, мы можем легко запустить готовые HTTP- и gRPC-серверы с дополнительными возможностями для разработчиков.

Команда `docker run` запускает контейнер, использующий свободно доступный образ `tensorflow/serving`, и монтирует путь `saved_model` из нашего текущего каталога (`$(pwd)/ saved_model`) к `/models/saved_model/1` внутри контейнера. Таким образом, сохраненная нами ранее в этой главе модель автоматически загружается в запущенный контейнер Docker, которому мы теперь можем отправлять запросы на создание прогнозов.

Если вы перейдете по адресу **http://localhost:8501/v1/models/saved_model** в браузере, то увидите подобный результат в формате JSON:


```
{
    "model_version_status": [
        {
            "version": "1",
            "state": "AVAILABLE",
            "status": {
                "error_code": "OK",
                "error_message": ""
            }
        }
    ]
}
```


Более подробную информацию о модели можно найти в папке /metadata по адресу **http://localhost:8501/v1/models/saved_model/metadata**:


```
{
"model_spec":{
 "name": "saved_model",
 "signature_name": "",
 "version": "1"
}
,
"metadata": {"signature_def": {
 "signature_def": {
  "serving_default": {
   "inputs": {
    "input_8": {
     "dtype": "DT_FLOAT",
     "tensor_shape": {
      "dim": [
       {
        "size": "-1",
        "name": ""
       },
       {
        "size": "10",
        "name": ""
       }
      ],
      "unknown_rank": false
     },
```

```
      "name": "serving_default_input_8:0"
     }
    },
    "outputs": {
     "dense_11": {
      "dtype": "DT_FLOAT",
      "tensor_shape": {
       "dim": [
        {
         "size": "-1",
         "name": ""
        },
        {
         "size": "1",
         "name": ""
        }
       ],
       "unknown_rank": false
      },
      "name": "StatefulPartitionedCall:0"
     }
    },
    "method_name": "tensorflow/serving/predict"
   },
   "__saved_model_init_op": {
    "inputs": {},
    "outputs": {
     "__saved_model_init_op": {
      "dtype": "DT_INVALID",
      "tensor_shape": {
       "dim": [],
       "unknown_rank": true
      },
      "name": "NoOp"
     }
    },
    "method_name": ""
   }
  }
 }
 }
 }
```

При помощи команды curl и сообщения переменных мы можем получить прогнозы в виде конечных точек в REST (данная нейронная сеть принимает 10 признаков):

```
curl -X POST http://localhost:8501/v1/models/saved_model:predict
    -d '{"inputs":[[1,2,3,4,5,6,7,8,9,10]]}'

{
    "outputs": [
        [
            5.59353495
        ]
    ]
}
```

Дополнительные материалы для чтения

- Serving Models ("Развертывание моделей", документация TensorFlow; **https://oreil.ly/5ZEQo**).

23.6. Развертывание моделей PyTorch через Seldon

Задача

Вы хотите предоставить доступ к обученной модели PyTorch в режиме реального времени.

Решение

Используем декоратор Python Seldon Core для развертывания модели:

```
# Импортировать библиотеки
import torch
import torch.nn as nn
import logging

# Создадим класс модели PyTorch
class SimpleNeuralNet(nn.Module):
    def __init__(self):
        super(SimpleNeuralNet, self).__init__()
        self.sequential = torch.nn.Sequential(
            torch.nn.Linear(10, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16,16),
```

```
            torch.nn.ReLU(),
            torch.nn.Linear(16, 1),
            torch.nn.Dropout(0.1), # Удалить 10% нейронов
            torch.nn.Sigmoid(),
        )

# Создать объект модели Seldon и назовем его `MyModel`
class MyModel(object):

    # Загрузить модель
    def __init__(self):
        self.network = SimpleNeuralNet()
        self.network.load_state_dict(
            torch.load("model.pt")["model_state_dict"],
            strict=False
        )
        logging.info(self.network.eval())

    # Создать прогноз
    def predict(self, X, features_names=None):
        return self.network.forward(X)
```

И запустим в контейнере Docker:

```
docker run -it -v $(pwd):/app -p 9000:9000 kylegallatin/seldon-example
    seldon-core-microservice MyModel --service-type MODEL

2023-03-11 14:40:52,277 - seldon_core.microservice:main:578 -
    INFO:  Starting microservice.py:main
2023-03-11 14:40:52,277 - seldon_core.microservice:main:579 -
    INFO:  Seldon Core version: 1.15.0
2023-03-11 14:40:52,279 - seldon_core.microservice:main:602 -
    INFO:  Parse JAEGER_EXTRA_TAGS []
2023-03-11 14:40:52,287 - seldon_core.microservice:main:605 -
    INFO:  Annotations: {}
2023-03-11 14:40:52,287 - seldon_core.microservice:main:609 -
    INFO:  Importing MyModel
2023-03-11 14:40:55,901 - root:__init__:25 - INFO:  SimpleNeuralNet(
  (sequential): Sequential(
    (0): Linear(in_features=10, out_features=16, bias=True)
    (1): ReLU()
    (2): Linear(in_features=16, out_features=16, bias=True)
    (3): ReLU()
```

```
    (4): Linear(in_features=16, out_features=1, bias=True)
    (5): Dropout(p=0.1, inplace=False)
    (6): Sigmoid()
  )
)
2023-03-11 14:40:56,024 - seldon_core.microservice:main:640 -
    INFO:  REST gunicorn microservice running on port 9000
2023-03-11 14:40:56,028 - seldon_core.microservice:main:655 -
    INFO:  REST metrics microservice running on port 6000
2023-03-11 14:40:56,029 - seldon_core.microservice:main:665 -
    INFO:  Starting servers
2023-03-11 14:40:56,029 - seldon_core.microservice:start_servers:80 -
    INFO:  Using standard multiprocessing library
2023-03-11 14:40:56,049 - seldon_core.microservice:server:432 -
    INFO:  Gunicorn Config:  {'bind': '0.0.0.0:9000', 'accesslog': None,
    'loglevel': 'info', 'timeout': 5000, 'threads': 1, 'workers': 1,
    'max_requests': 0, 'max_requests_jitter': 0, 'post_worker_init':
    <function post_worker_init at 0x7f5aee2c89d0>, 'worker_exit':
    functools.partial(<function worker_exit at 0x7f5aee2ca170>,
    seldon_metrics=<seldon_core.metrics.SeldonMetrics object at
    0x7f5a769f0b20>), 'keepalive': 2}
2023-03-11 14:40:56,055 - seldon_core.microservice:server:504 -
    INFO:  GRPC Server Binding to 0.0.0.0:5000 with 1 processes.
2023-03-11 14:40:56,090 - seldon_core.wrapper:_set_flask_app_configs:225 -
    INFO:  App Config:  <Config {'ENV': 'production', 'DEBUG': False,
    'TESTING': False, 'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY': None,
    'PERMANENT_SESSION_LIFETIME': datetime.timedelta(days=31),
    'USE_X_SENDFILE': False, 'SERVER_NAME': None, 'APPLICATION_ROOT': '/',
    'SESSION_COOKIE_NAME': 'session', 'SESSION_COOKIE_DOMAIN': None,
    'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_HTTPONLY': True,
    'SESSION_COOKIE_SECURE': False, 'SESSION_COOKIE_SAMESITE': None,
    'SESSION_REFRESH_EACH_REQUEST': True, 'MAX_CONTENT_LENGTH': None,
    'SEND_FILE_MAX_AGE_DEFAULT': None, 'TRAP_BAD_REQUEST_ERRORS': None,
    'TRAP_HTTP_EXCEPTIONS': False, 'EXPLAIN_TEMPLATE_LOADING': False,
    'PREFERRED_URL_SCHEME': 'http', 'JSON_AS_ASCII': None,
    'JSON_SORT_KEYS': None, 'JSONIFY_PRETTYPRINT_REGULAR': None,
    'JSONIFY_MIMETYPE': None, 'TEMPLATES_AUTO_RELOAD': None,
    'MAX_COOKIE_SIZE': 4093}>
2023-03-11 14:40:56,091 - seldon_core.wrapper:_set_flask_app_configs:225 -
    INFO:  App Config:  <Config {'ENV': 'production', 'DEBUG': False,
    'TESTING': False, 'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY': None,
```

```
    'PERMANENT_SESSION_LIFETIME': datetime.timedelta(days=31),
    'USE_X_SENDFILE': False, 'SERVER_NAME': None, 'APPLICATION_ROOT': '/',
    'SESSION_COOKIE_NAME': 'session', 'SESSION_COOKIE_DOMAIN': None,
    'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_HTTPONLY': True,
    'SESSION_COOKIE_SECURE': False, 'SESSION_COOKIE_SAMESITE': None,
    'SESSION_REFRESH_EACH_REQUEST': True, 'MAX_CONTENT_LENGTH': None,
    'SEND_FILE_MAX_AGE_DEFAULT': None, 'TRAP_BAD_REQUEST_ERRORS': None,
    'TRAP_HTTP_EXCEPTIONS': False, 'EXPLAIN_TEMPLATE_LOADING': False,
    'PREFERRED_URL_SCHEME': 'http', 'JSON_AS_ASCII': None,
    'JSON_SORT_KEYS': None, 'JSONIFY_PRETTYPRINT_REGULAR': None,
    'JSONIFY_MIMETYPE': None, 'TEMPLATES_AUTO_RELOAD': None,
    'MAX_COOKIE_SIZE': 4093}>
2023-03-11 14:40:56,096 - seldon_core.microservice:_run_grpc_server:466 - INFO:
    Starting new GRPC server with 1 threads.
[2023-03-11 14:40:56 +0000] [23] [INFO] Starting gunicorn 20.1.0
[2023-03-11 14:40:56 +0000] [23] [INFO] Listening at: http://0.0.0.0:6000 (23)
[2023-03-11 14:40:56 +0000] [23] [INFO] Using worker: sync
[2023-03-11 14:40:56 +0000] [30] [INFO] Booting worker with pid: 30
[2023-03-11 14:40:56 +0000] [1] [INFO] Starting gunicorn 20.1.0
[2023-03-11 14:40:56 +0000] [1] [INFO] Listening at: http://0.0.0.0:9000 (1)
[2023-03-11 14:40:56 +0000] [1] [INFO] Using worker: sync
[2023-03-11 14:40:56 +0000] [34] [INFO] Booting worker with pid: 34
2023-03-11 14:40:56,217 - seldon_core.gunicorn_utils:load:103 - INFO:
    Tracing not active
```

Обсуждение

В данном примере мы использовали декоратор Python Seldon Core для развертывания нашей модели PyTorch, хотя существует также множество других инструментов. Seldon Core — это популярный фреймворк для развертывания моделей в виде готовых продуктов, имеющий множество полезных особенностей, которые упрощают их использование и масштабирование в сравнении с приложением на основе Flask. Нам достаточно написать простой класс (выше мы использовали `MyModel`), а библиотека на Python самостоятельно развернет все компоненты и конечные точки. Мы можем воспользоваться сервисом при помощи команды `seldon-core-microservice`, которая запустит сервер REST, сервер gRPC и даже предоставит метрики конечных точек. Для того чтобы создать прогноз, мы можем обратиться к конечной точке сервиса через порт 9000:

```
curl -X POST http://127.0.0.1:9000/predict  -H 'Content-Type: application/json'
    -d '{"data": {"ndarray":[[0, 0, 0, 0, 0, 0, 0, 0, 0]]}}'
```

Вы должны увидеть следующий вывод:

```
{"data":{"names":["t:0","t:1","t:2","t:3","t:4","t:5","t:6","t:7","t:8"],
    "ndarray":[[0,0,0,0,0,0,0,0,0]]},"meta":{}}
```

Дополнительные материалы для чтения

- Seldon Core Python Package ("Пакет Seldon Core на Python"; **https://oreil.ly/FTofY**).
- Документация TorchServe (**https://oreil.ly/fjmrE**).

Предметный указатель

A

Accuracy 128, 229
AdaBoost, алгоритм 297
all_timezones, библиотека 152
Approximate nearest neighbors (ANN) 310
Area under the ROC curve (AUC ROC) 235
Autograd, библиотека 368

B

Beautiful Soup 133

C

Compressed sparse column (CSC) 21
Compressed sparse row (CSR) 20
Confusion matrix 237
Convolutional neural networks (CNN) 412

D

DataFrame, создание 58
Dictionary of keys (DOK) 21

F

F1-мера 128, 231
Faiss, библиотека 310
Flatten array 30

G

Google Sheet 53
GrabCut, алгоритм 177

I

Interquartile range (IQR) 102

J

joblib, библиотека 423

K

k ближайших соседей 111, 114, 122, 303

L

lightgbm, библиотека 300, 301
Linear discriminant analysis (LDA) 203
List of lists (LIL) 21

M

Matplotlib, библиотека 45, 387
Mean squared error (MSE) 240
Min-max scaling 93

N

Naive Bayes classifier 336
Natural Language Toolkit (NLTK) 134
NLTK, библиотека 135, 137
NumPy, библиотека 17

O

One-hot encode 115
OpenCV, библиотека 162
Out-of-bag (OOB) 290

P

pandas 46
PCA 198
Pillow, библиотека 167
Precision 128, 230
Principal component analysis (PCA) 95
PyTorch, библиотека 354
pytz, библиотека 152

R

ray, библиотека 402
Recall 128, 231
Receiver operating characteristic (ROC) 232
Rectified linear unit (ReLU) 373
ROC-кривая 128

S

S3 bucket 54
spaCy, библиотека 139
Structured query language (SQL) 51
Support vector machines (SVM) 324

T

transformers, библиотека 147, 148, 416, 418, 419

X

XGBoost, алгоритм 300
xgboost, библиотека 299

Z

z-оценка 95

А

Алгоритм *k* ближайших соседей 122
Анализ:
◊ главных компонент 198
◊ линейный дискриминантный 203
Атрибут size 28

Б

База данных SQL, подключение 52
Бустинг 297

В

Валидация 223
Вектор 17
◊ вывод элемента 22
◊ опорный 333
◊ скалярное произведение 34
◊ транспонирование 29
Векторизация 25
Вероятность:
◊ апостериорная 336
◊ априорная 337
Вес 253
Выбор статистической модели 253
Выброс:
◊ обнаружение 101
◊ управление 103
Выделение признаков 210
Выравнивание гистограммы 171

Г

Гиперпараметр 253
◊ оптимизация 253
Гиперплоскость 324

Д

Данные:
◊ обработка первичная 57
◊ очистка базовая 129
◊ предобработка для нейронной сети 370
Дата:
◊ выбор 152
◊ разница между двумя датами 154
День недели 155
Дерево 280
◊ размер, контроль 295
◊ решений 280
Детектор:
◊ границ Кэнни 180
◊ углов Харриса 183
◊ Ши-Томаси, угловой 183
Дискретизация признаков 105
Дисперсия:
◊ обработка пороговая 211
◊ уменьшение 276, 278, 319

З

Замена значений 66
Знак препинания, удаление 133
Значение:
◊ отсутствующее 71
◊ псевдослучайное 38
◊ уникальное 70

И

Изображение:
◊ бинаризация 175
◊ гистограмма цветов 187
◊ границы 180
◊ загрузка 55, 162
◊ кадрирование 167
◊ классификация 194, 410
◊ контраст 171
◊ признаки, создание 185
◊ размер, изменение 166
◊ размытие 168
◊ распознавание 193
◊ резкость 170
◊ сохранение 165
◊ углы 181
◊ фон, удаление 177
◊ цвет, распознавание 173

Интерполяция 160
Исключение признаков рекурсивное 218

К

Класс:
- ◊ AdaBoostClassifier 297
- ◊ AdaBoostRegressor 297
- ◊ AgglomerativeClustering 352
- ◊ BernoulliNB 341
- ◊ Binarizer 107
- ◊ CalibratedClassifierCV 342, 343
- ◊ CountVectorizer 141
- ◊ DecisionTreeClassifier 280
- ◊ DecisionTreeRegressor 282, 283
- ◊ DummyRegressor 226
- ◊ EllipticEnvelope 103
- ◊ FeatureUnion 262
- ◊ FunctionTransformer 100
- ◊ GridSearchCV 254, 258, 262, 308
- ◊ kernelPCA 202
- ◊ KMeans 346
- ◊ KNeighborsClassifier 306
- ◊ LabelBinarizer 115
- ◊ LinearRegression 269
- ◊ LinearSVC 325
- ◊ LogisticRegression 265, 316–318, 321, 322
- ◊ LogisticRegressionCV 265
- ◊ MeanShift 349
- ◊ MiniBatchKMeans 348
- ◊ MinMaxScaler 92
- ◊ MultinomialNB 340
- ◊ NearestNeighbors 303
- ◊ Normalizer 96
- ◊ PolynomialFeatures 271, 275
- ◊ PorterStemmer 136
- ◊ RadiusNeighborsClassifier 309
- ◊ RandomForestClassifier 286
- ◊ RandomForestRegressor 288
- ◊ RandomizedSearchCV 256
- ◊ RidgeCV 277
- ◊ RobustScaler 95, 105
- ◊ SelectFromModel 293
- ◊ SimpleImputer 111
- ◊ StandardScaler 94, 306, 370
- ◊ SVC 331
- ◊ TfidfVectorizer 143
- ◊ vectorize 25
- ◊ несбалансированный 124, 294, 322, 334

Классификатор:
- ◊ *k* ближайших соседей 306
- ◊ бинарный 229, 316, 376
  - – оценка порогов 232
- ◊ для дискретных признаков 339
- ◊ для непрерывных признаков 337
- ◊ лес случайный 286
- ◊ линейный 324
- ◊ многоклассовый 236, 318, 379
- ◊ наивный байесовский 336
  - – на основе распределения Гаусса 338
  - – полиномиальный 339
  - – с распределением Бернулли 341
- ◊ Хаара, каскадный 194

Классификация 43
- ◊ бинарная 374
- ◊ многоклассовая 374
- ◊ тональности 147

Кластеризация 242
- ◊ DBSCAN 350
- ◊ *k* средних 344
  - – мини-пакетная 347
- ◊ агломеративная 352
- ◊ иерархическая 351
- ◊ со сдвигом среднего 348

Кодирование унитарное 115
Конкатенация 87
Коэффициент:
- ◊ Джини 281
- ◊ силуэта 242

Кривая ROC 232
Кросс-валидация 218
- ◊ *k*-блочная 223
- ◊ вложенная 266

Л

Лес случайный 287
Лист 280

М

Максимум 26
Массив:
- ◊ двумерный 18
- ◊ заданный заранее 21
- ◊ максимум 26
- ◊ минимум 26
- ◊ многомерный 18
- ◊ одномерный, создание 17
- ◊ применение функции 24
- ◊ форма, изменение 28

Масштабирование 93
- ◊ в заданном диапазоне 93

Матрица 18
- ◊ вывод элемента 22
- ◊ вычитание 35
- ◊ диагональ 32
  - – сумма 33
- ◊ обратная 37
- ◊ ошибок 128, 237
- ◊ преобразование в одномерный массив 30
- ◊ размерность 24
- ◊ разреженная 19, 20, 142
- ◊ ранг 31
- ◊ след 33
- ◊ словарь ключей 21
- ◊ сложение 35
- ◊ со сжатым хранением столбцом 21
- ◊ со сжатым хранением строкой 20
- ◊ список списков 21
- ◊ транспонирование 29
- ◊ умножение 36
  - – элементное 36
- ◊ форма 24
- ◊ число элементов 24

Метод:
- ◊ agg 82
- ◊ apply 85, 86
- ◊ day_name 155
- ◊ describe 60
- ◊ diagonal 32
- ◊ digitize 107
- ◊ drop 73
- ◊ drop_duplicates 76
- ◊ duplicated 78
- ◊ fillna 73
- ◊ flatten 30, 185, 364
- ◊ from_numpy 355
- ◊ get_feature_names_out 121, 141
- ◊ groupby 78, 86
- ◊ head 59
- ◊ info 60
- ◊ interpolate 161
- ◊ isna 73
- ◊ isnull 72
- ◊ *k* средних 344
- ◊ max 361
- ◊ min 361
- ◊ mT 363
- ◊ notnull 72
- ◊ permute 363
- ◊ predict_proba 234
- ◊ ravel 30
- ◊ rename 67
- ◊ replace 66, 118
- ◊ resample 80
- ◊ reshape 28, 362
- ◊ rolling 157
- ◊ Series.dt 153
- ◊ shift 156
- ◊ sort_values 65
- ◊ to_sparse 356
- ◊ toarray 141
- ◊ trace 33
- ◊ translate 133
- ◊ tz_localize 151
- ◊ unique 70
- ◊ value_counts 71
- ◊ weekday 156
- ◊ опорных векторов 324
- ◊ полного перебора 254
- ◊ приближенных ближайших соседей 310
- ◊ скользящего окна 158

Метод главных компонент 95
Мешок слов 141
Минимум 26
Модель:
- ◊ базовая регрессионная 225
- ◊ дерева решений, визуализация 283
- ◊ классификации, базовая 227
- ◊ кластеризации 242

- ◊ кросс-валидация 221
- ◊ оценка качества 266
- ◊ поиск:
  - – лучшей 254, 256, 258, 260
  - – ускорение 263, 264
- ◊ развертывание 428
- ◊ регрессионная 240
- ◊ сохранение и загрузка 422

Н

Наблюдение вне пакета 290
Набор данных:
- ◊ для классификации 43
- ◊ для линейной регрессии 42
- ◊ загрузка 40
- ◊ игрушечный 41
- ◊ искусственный 42
- ◊ обучающий 223
- ◊ тестовый 223

Нейрон 367
Норма:
- ◊ евклидова 97
- ◊ манхэттенская 97

О

Обработка пороговая 175
- ◊ адаптивная 175

Обучение:
- ◊ визуализация истории 387
- ◊ глубокое 367
- ◊ трансферное 191, 418

Окно скользящее 157
Оператор:
- ◊ * 36
- ◊ @ 34, 36
- ◊ + 35

Оптимизация гиперпараметров 253
Остановка ранняя 395
Отбор признаков 210
Оценка стандартизированная 95
Очистка базовая 129
Ошибка:
- ◊ среднеквадратическая 240, 283
- ◊ средняя абсолютная 283

П

Параметр 253
Передискретизация 127
Переобучение 390, 392, 396
Площадь под ROC-кривой 235
Полнота 128, 231
Пояс часовой 151
Правильность 128, 229
Признак:
- ◊ бинарный, отбор 212
- ◊ исключение рекурсивное 218
- ◊ сокращение 198, 200, 202, 205
- ◊ удаление 215
- ◊ числовой, отбор 211

Проверка удерживающая 223
Прогноз 384

Р

Разделение:
- ◊ на предложения 134
- ◊ на слова 134

Разложение:
- ◊ неотрицательное матричное 205
- ◊ усеченное сингулярное 207

Размах межквартильный 102
Ранг матрицы 31
Распределение маргинальное 337
Распространение:
- ◊ обратное 368
- ◊ прямое 368

Регрессия 374
- ◊ гребневая 277
- ◊ лассо 277, 278
- ◊ линейная 42, 269
- ◊ логистическая 316
  - – полиномиальная 319
- ◊ полиномиальная 273

Регрессор 282
- ◊ лес случайный 288
- ◊ обучение 382

Регуляризация 276, 320, 392

С

Свойство:
- ◊ iloc 62
- ◊ loc 62

Сдвиг среднего значения 349
Сеть:
- ◊ нейронная:
  - – визуализация 405
  - – классификация изображений 410
  - – классификация текстов 413
  - – настройка 402
  - – предобработка данных 370
  - – разработка 372
  - – с прямой связью 367
  - – сверточная 412
- ◊ эластичная 277

След матрицы 33
Слияние 88
Словарь:
- ◊ ключей 21
- ◊ преобразование в матрицу 120

Слой:
- ◊ входной 367
- ◊ выходной 367
- ◊ полносвязный 412
- ◊ пулинговый 412
- ◊ свертки 412
- ◊ скрытый 367

Смещение 273, 275
Соединение:
- ◊ внешнее 89, 90
  - – левое 89
  - – правое 89
- ◊ внутреннее 88, 90
- ◊ левое 90
- ◊ правое 91

Сокращение веса 392
Сортировка 65
Список списков 21
Спуск градиентный 368
Срез 23, 61
Стемминг 136, 137
Столбец:
- ◊ переименование 67
- ◊ применение функции ко всем элементам 85
- ◊ удаление 73

Стоп-слово 135
Строка:
- ◊ группировка 78
  - – по времени 80
- ◊ преобразование в дату 149
- ◊ применение функции к группе 86
- ◊ удаление 75
  - – дублирующейся 76

Субдискретизация 126

Т

Текст:
- ◊ анализ синтаксический 132
- ◊ загрузка 55
- ◊ значимость слов 143
- ◊ классификация 413
- ◊ очистка 129
- ◊ разметка частей речи 137
- ◊ стемминг 136, 137
- ◊ токенизация 134
- ◊ тональность 147
- ◊ удаление:
  - – знаков препинания 133
  - – стоп-слов 135

Тензор:
- ◊ выбор элемента 357
- ◊ значение:
  - – наибольшее 361
  - – наименьшее 361
- ◊ одномерный 364
- ◊ операции над элементами 360
- ◊ описание 359
- ◊ разреженный 356
- ◊ создание 354
  - – из массива 355
- ◊ транспонирование 363
- ◊ умножение 365
- ◊ форма 362

Теорема Байеса 336
Токенизация 135
Тональность 147
Точность 128, 230
Транслирование 25, 360
Трюк ядерный 202

У

Узел 367

Ф

Файл:
- ◊ Avro 50
- ◊ CSV 46
- ◊ JSON 48
- ◊ Parquet 49
- ◊ SQL 51
- ◊ электронной таблицы 47

Функция:
- ◊ adaptiveThreshold 176
- ◊ add 35
- ◊ classification_report 248
- ◊ concat 87
- ◊ cornerHarris 181
- ◊ cross_val_score 229
- ◊ dot 34, 36
- ◊ equalizeHist 172
- ◊ filter2D 170
- ◊ goodFeaturesToTrack 183
- ◊ imread 162
- ◊ imwrite 165
- ◊ inv 37
- ◊ json_normalize 48
- ◊ make_blobs 43
- ◊ make_circles 201
- ◊ make_classification 43
- ◊ make_regression 42
- ◊ make_scorer 245
- ◊ matrix_rank 31
- ◊ max 26
- ◊ mean 27
- ◊ merge 88
- ◊ min 26
- ◊ open 56
- ◊ random 38
- ◊ read_avro 50
- ◊ read_csv 46, 73
- ◊ read_CSV 53
- ◊ read_excel 47
- ◊ read_json 48
- ◊ read_parquet 49
- ◊ read_sql_query 51
- ◊ roc_auc_score 235
- ◊ roc_curve 232
- ◊ silhouette_score 243
- ◊ std 27
- ◊ subtract 35
- ◊ T 29
- ◊ to_datetime 150
- ◊ validation_curve 252
- ◊ var 27
- ◊ агрегатная 82
- ◊ активации 373
- ◊ потерь 368
- ◊ правдоподобия 336
- ◊ применение к элементу массива 24

Ч

Частота документа 144

Ш

Шкала Ликерта 118
Шкалирование минимаксное 93
Штраф за уменьшение 276, 277

Э

Электронная таблица Excel 47
Эмбеддинг 129, 190
Эпоха 368

Я

Ядро:
- ◊ линейное 328
- ◊ полиномиальное 328
- ◊ радиальной базисной функции 3

Об авторах

Кайл Галлатин (Kyle Gallatin) — разработчик программного обеспечения для инфраструктуры машинного обучения, имеющий многолетний опыт работы в качестве аналитика данных, исследователя данных и ML-инженера. Он является профессиональным наставником в области науки о данных и в качестве волонтера преподает компьютерные науки, а также часто публикует статьи о разработке программного обеспечения и машинном обучении. В настоящее время Кайл является членом команды разработчиков программного обеспечения для платформы машинного обучения из Etsy.

Крис Элбон (Chris Albon) — руководитель отдела машинного обучения в Фонде Викимедиа — некоммерческой организации, поддерживающей инфраструктуру Википедии.

Об изображении на обложке

Птица на обложке этой книги — уздечковый африканский трогон, также известный как *Apaloderna narina*. Он был назван в честь возлюбленной французского орнитолога Франсуа Левальяна, которую тот ввиду сложности имени называл Нариной, что на языке готтентотов (Khoikhoi) означает "цветок". Уздечковый африканский трогон обитает на большей части Африки, в низинах и гористой местности, тропическом и умеренном климате, обустраивая гнездо в кроне деревьев. Благодаря большой популяции истребление данному виду не грозит.

Уздечковый африканский трогон питается насекомыми и небольшими беспозвоночными, а также мелкими грызунами и рептилиями. Особи мужского пола обладают ярким окрасом и издают резкое, низкое, повторяющееся гиканье для защиты территории и привлечения самок. Особи обоих полов имеют зеленое оперение верхней стороны и металлически-сине-зеленые перья на хвосте. Оперение на шее и груди самки коричневое, в то время как у самцов брюшко ярко-красное. По окрасу птенцы похожи на самок — белые полосы по середине крыла.

Международный союз охраны природы и природных ресурсов (International Union for Conservation of Nature and Natural Resources, IUCN) относит *Apaloderna narina* к видам, вызывающим наименьшие опасения в плане истребления. Многие из животных, изображенные на обложках книг издательства O'Reill, находятся под угрозой исчезновения; все они важны для планеты.

Иллюстрацию на обложке выполнила Карен Монтгомери (Karen Montgomery) на основе изображения из книги "Одушевленное творение" ("Animate creation") Джона Вуда (John G. Wood).

Кайл Галлатин, Крис Элбон

Машинное обучение с использованием Python. Сборник рецептов. 2 изд.

Перевод с английского И. Печего

ТОО "АЛИСТ"
010000, Республика Казахстан,
г. Астана, пр. Сарыарка, д. 17, ВП 30

Подписано в печать 11.06.24.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 36,12.
Тираж 1500 экз. Заказ № 9777.

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

O'REILLY®

Машинное обучение с использованием Python. Сборник рецептов

Книга содержит около 200 рецептов, которые помогут решить задачи машинного обучения, возникающие в повседневной работе практикующего специалиста, такие как загрузка и обработка текстовых или числовых данных, отбор модели, уменьшение размерности и многие другие. Рассмотрена работа с языком Python и его библиотеками, в том числе pandas и scikit-learn. Решения всех задач сопровождаются подробными объяснениями. Каждый рецепт содержит программный код, который можно скопировать и опробовать на игрушечном наборе данных (toy dataset). Затем этот код можно вставлять, объединять и адаптировать, создавая собственные приложения.

В книге вы найдете рецепты для:

- **векторов, матриц и массивов;**
- **работы с данными из CSV, JSON, SQL, базами данных, облачными хранилищами и другими источниками;**
- **обработки числовых и категориальных данных, текста, изображений, дат и времени;**
- **уменьшения размерности с использованием методов выделения или отбора признаков;**
- **оценивания и отбора моделей;**
- **сохранения и загрузки натренированных моделей.**

Вы научитесь решать задачи с использованием:

- **линейной и логистической регрессии, деревьев, лесов и *k* ближайших соседей;**
- **опорно-векторных машин (SVM), наивных байесовых классификаторов, кластеризации и нейронных сетей.**

«Я беспокоился, что никогда не смогу найти сборник рецептов, в котором кратко описываются метод приближенных ближайших соседей, гиперплоскости и отбор признаков с помощью случайного леса, но затем эта книга вошла в мою жизнь».

— Вики Бойкис, инженер по машинному обучению компании Duo

Кайл Галлатин — инженер-программист команды разработчиков ПО для крупнейшей торговой площадки Etsy с многолетним опытом работы в качестве аналитика данных, специалиста по данным и инженера по машинному обучению.

Крис Элбон — руководитель отдела машинного обучения в Фонде Викимедиа — некоммерческой организации, поддерживающей инфраструктуру Википедии.

ISBN 978-601-08-4119-2

9 786010 841192

alst