

# Машинное обучение

Портфолио  
реальных  
проектов

Алексей Григорьев



# *Machine Learning Bookcamp*

BUILD A PORTFOLIO OF REAL-LIFE PROJECTS

ALEXEY GRIGOREV



MANNING  
SHELTER ISLAND

# Машинное обучение

Портфолио реальных проектов

Алексей Григорьев



Санкт-Петербург · Москва · Минск

2023

*Алексей Григорьев*

## **Машинное обучение. Портфолио реальных проектов**

*Серия «Библиотека программиста»*

*Перевел с английского Р. Чикин*

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Хлебина</i>
Обложка	<i>В. Мостипан</i>
Корректоры	<i>М. Лауконен, Н. Сидорова</i>
Верстка	<i>М. Жданова</i>

ББК 32.973.2-018 +32.813 УДК 004.41+004.85

**Григорьев Алексей**

Г83 Машинное обучение. Портфолио реальных проектов. — СПб.: Питер, 2023. — 496 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1978-3

Изучите ключевые концепции машинного обучения, работая над реальными проектами! Машинное обучение — то, что поможет вам в анализе поведения клиентов, прогнозировании тенденций движения цен, оценке рисков и многом другом. Чтобы освоить машинное обучение, вам нужны отличные примеры, четкие объяснения и много практики. В книге все это есть!

Автор описывает реалистичные, практические сценарии машинного обучения, а также дает предельно понятные объяснения ключевых концепций. Вы разберете интересные проекты, такие как сервис прогнозирования цен на автомобили с использованием линейной регрессии и сервис прогнозирования оттока клиентов. Вы выйдете за рамки алгоритмов и изучите важные техники, например развертывание приложений в бессерверных системах и запуск моделей с помощью Kubernetes и Kubeflow. Пришло время закатать рукава и прокачать свои навыки в области машинного обучения!

В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, такие как Meta Platforms Inc., Facebook, Instagram и др.

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

ISBN 978-1617296819 англ.

© 2021 by Manning Publications Co. All rights reserved.

ISBN 978-5-4461-1978-3 рус.

© Перевод на русский язык ООО «Прогресс книга», 2023

© Издание на русском языке, оформление ООО «Прогресс книга», 2023

© Серия «Библиотека программиста», 2023

Права на издание получены по соглашению с Manning Publications.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

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

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес:  
194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,  
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.  
Подписано в печать 25.01.23. Формат 70x100/16. Бумага офсетная. Усл. п. л. 39,990. Тираж 700. Заказ

# *Краткое содержание*

---

<b>Предисловие</b> . . . . .	14
<b>Введение</b> . . . . .	16
<b>Благодарности</b> . . . . .	18
<b>Об этой книге.</b> . . . . .	20
<b>Об авторе</b> . . . . .	24
<b>Иллюстрация на обложке</b> . . . . .	25
<b>Глава 1.</b> Введение в машинное обучение. . . . .	26
<b>Глава 2.</b> Машинное обучение для регрессии . . . . .	47
<b>Глава 3.</b> Машинное обучение для классификации . . . . .	99
<b>Глава 4.</b> Оценочные показатели для классификации . . . . .	151
<b>Глава 5.</b> Разворачивание моделей машинного обучения . . . . .	196
<b>Глава 6.</b> Деревья решений и ансамблевое обучение . . . . .	225
<b>Глава 7.</b> Нейронные сети и глубокое обучение . . . . .	274
<b>Глава 8.</b> Бессерверное глубокое обучение . . . . .	326
<b>Глава 9.</b> Предоставление моделей с помощью Kubernetes и Kubeflow . . . . .	348
<b>Приложение А.</b> Подготовка среды . . . . .	387
<b>Приложение Б.</b> Введение в Python . . . . .	418
<b>Приложение В.</b> Введение в NumPy . . . . .	435
<b>Приложение Г.</b> Введение в Pandas . . . . .	465
<b>Приложение Д.</b> AWS SageMaker . . . . .	487

# Оглавление

---

<b>Предисловие . . . . .</b>	14
<b>Введение . . . . .</b>	16
<b>Благодарности . . . . .</b>	18
<b>Об этой книге . . . . .</b>	20
Кому адресована книга . . . . .	20
Структура издания . . . . .	20
О коде . . . . .	23
Другие онлайн-ресурсы . . . . .	23
От издательства . . . . .	23
<b>Об авторе . . . . .</b>	24
<b>Иллюстрация на обложке . . . . .</b>	25
<b>Глава 1. Введение в машинное обучение . . . . .</b>	26
1.1. Машинное обучение . . . . .	27
1.1.1. Машинное обучение в сравнении с системами, основанными на правилах . . . . .	29
1.1.2. Когда машинное обучение бесполезно . . . . .	33
1.1.3. Контролируемое машинное обучение . . . . .	33
1.2. Процесс машинного обучения . . . . .	35
1.2.1. Бизнес-анализ . . . . .	37
1.2.2. Анализ данных . . . . .	37
1.2.3. Подготовка данных . . . . .	38
1.2.4. Моделирование . . . . .	38
1.2.5. Оценка . . . . .	39
1.2.6. Развертывание . . . . .	39
1.2.7. Повтор . . . . .	40
1.3. Моделирование и проверка модели . . . . .	40
Резюме . . . . .	45

<b>Глава 2. Машинное обучение для регрессии . . . . .</b>	47
2.1. Проект по прогнозированию цен на автомобили . . . . .	48
2.1.1. Загрузка набора данных . . . . .	49
2.2. Исследовательский анализ данных . . . . .	50
2.2.1. Набор инструментов для исследовательского анализа данных . . . . .	50
2.2.2. Считывание и подготовка данных . . . . .	52
2.2.3. Анализ целевых переменных . . . . .	55
2.2.4. Проверка на наличие пропущенных значений . . . . .	58
2.2.5. Платформа проверки . . . . .	59
2.3. Машинное обучение для регрессии . . . . .	63
2.3.1. Линейная регрессия . . . . .	63
2.3.2. Обучающая модель линейной регрессии . . . . .	71
2.4. Прогнозирование цены . . . . .	74
2.4.1. Базовое решение . . . . .	74
2.4.2. RMSE: оценка качества модели . . . . .	77
2.4.3. Проверка модели . . . . .	81
2.4.4. Простое конструирование признаков . . . . .	83
2.4.5. Обработка категориальных переменных . . . . .	85
2.4.6. Регуляризация . . . . .	90
2.4.7. Использование модели . . . . .	95
2.5. Следующие шаги . . . . .	96
2.5.1. Упражнения . . . . .	96
2.5.2. Другие проекты . . . . .	96
Резюме . . . . .	96
Ответы к упражнениям . . . . .	98
<b>Глава 3. Машинное обучение для классификации . . . . .</b>	99
3.1. Проект по прогнозированию оттока клиентов . . . . .	100
3.1.1. Набор данных об оттоке телекоммуникационной компании . . . . .	101
3.1.2. Подготовка исходных данных . . . . .	102
3.1.3. Исследовательский анализ данных . . . . .	110
3.1.4. Важность признака . . . . .	113
3.2. Конструирование признаков . . . . .	124
3.2.1. Прямое кодирование для категориальных переменных . . . . .	124
3.3. Машинное обучение для классификации . . . . .	128
3.3.1. Логистическая регрессия . . . . .	129
3.3.2. Обучение логистической регрессии . . . . .	132
3.3.3. Интерпретация модели . . . . .	137
3.3.4. Использование модели . . . . .	145

## **8      Оглавление**

3.4. Следующие шаги . . . . .	147
3.4.1. Упражнения . . . . .	147
3.4.2. Другие проекты. . . . .	148
Резюме . . . . .	149
Ответы к упражнениям . . . . .	150
<b>Глава 4. Оценочные показатели для классификации . . . . .</b>	<b>151</b>
4.1. Показатели оценки. . . . .	152
4.1.1. Достоверность классификации . . . . .	152
4.1.2. Фиктивная базовая линия . . . . .	156
4.2. Матрица ошибок . . . . .	157
4.2.1. Введение в матрицу ошибок. . . . .	157
4.2.2. Вычисление матрицы ошибок с помощью NumPy . . . . .	160
4.2.3. Точность и отклик . . . . .	165
4.3. Кривая ROC и оценка AUC . . . . .	169
4.3.1. Доля истинно положительных и ложноположительных результатов . . . . .	169
4.3.2. Оценка модели при нескольких пороговых значениях . . . . .	171
4.3.3. Случайная базовая модель. . . . .	174
4.3.4. Идеальная модель . . . . .	176
4.3.5. Кривая ROC . . . . .	179
4.3.6. Площадь под кривой ROC (AUC) . . . . .	185
4.4. Настройка параметров . . . . .	188
4.4.1. К-кратная перекрестная проверка . . . . .	188
4.4.2. Поиск наилучших параметров . . . . .	190
4.5. Следующие шаги . . . . .	193
4.5.1. Упражнения . . . . .	193
4.5.2. Другие проекты. . . . .	194
Резюме . . . . .	194
Ответы к упражнениям . . . . .	195
<b>Глава 5. Развёртывание моделей машинного обучения . . . . .</b>	<b>196</b>
5.1. Модель прогнозирования оттока . . . . .	197
5.1.1. Использование модели . . . . .	197
5.1.2. Использование Pickle для сохранения и загрузки модели . . . . .	198
5.2. Доступность модели. . . . .	202
5.2.1. Веб-сервисы . . . . .	202
5.2.2. Flask . . . . .	204
5.2.3. Обеспечение доступа к модели оттока с помощью Flask . . . . .	206

5.3. Управление зависимостями . . . . .	209
5.3.1. Pipenv . . . . .	209
5.3.2. Docker . . . . .	214
5.4. Развёртывание . . . . .	219
5.4.1. AWS Elastic Beanstalk . . . . .	219
5.5. Следующие шаги . . . . .	223
5.5.1. Упражнения . . . . .	223
5.5.2. Другие проекты . . . . .	224
Резюме . . . . .	224
<b>Глава 6. Деревья решений и ансамблевое обучение . . . . .</b>	<b>225</b>
6.1. Проект по оценке кредитного риска . . . . .	226
6.1.1. Набор данных для оценки кредитоспособности . . . . .	227
6.1.2. Очистка данных . . . . .	228
6.1.3. Подготовка набора данных . . . . .	233
6.2. Деревья решений . . . . .	236
6.2.1. Классификатор дерева решений . . . . .	237
6.2.2. Алгоритм обучения дерева решений . . . . .	240
6.2.3. Настройка параметров для дерева решений . . . . .	248
6.3. Случайный лес . . . . .	250
6.3.1. Обучение случайного леса . . . . .	253
6.3.2. Настройка параметров для случайного леса . . . . .	255
6.4. Градиентный бустинг . . . . .	258
6.4.1. XGBoost: экстремальный градиентный бустинг . . . . .	259
6.4.2. Мониторинг производительности модели . . . . .	261
6.4.3. Настройка параметров XGBoost . . . . .	263
6.4.4. Тестирование окончательной модели . . . . .	269
6.5. Следующие шаги . . . . .	271
6.5.1. Упражнения . . . . .	271
6.5.2. Другие проекты . . . . .	272
Резюме . . . . .	272
Ответы к упражнениям . . . . .	273
<b>Глава 7. Нейронные сети и глубокое обучение . . . . .</b>	<b>274</b>
7.1. Модная классификация . . . . .	275
7.1.1. GPU vs CPU . . . . .	276
7.1.2. Загрузка набора данных . . . . .	276
7.1.3. TensorFlow и Keras . . . . .	278
7.1.4. Загрузка изображений . . . . .	279

## 10 Оглавление

7.2. Сверточные нейронные сети . . . . .	281
7.2.1. Использование предварительно обученной модели . . . . .	282
7.2.2. Получение прогнозов . . . . .	284
7.3. Внутри модели . . . . .	285
7.3.1. Сверточные слои . . . . .	286
7.3.2. Плотные слои . . . . .	289
7.4. Обучение модели . . . . .	292
7.4.1. Обучение с переносом . . . . .	292
7.4.2. Загрузка данных . . . . .	293
7.4.3. Создание модели . . . . .	294
7.4.4. Обучение модели . . . . .	298
7.4.5. Настройка скорости обучения . . . . .	302
7.4.6. Сохранение модели и контрольная точка . . . . .	304
7.4.7. Добавление дополнительных слоев . . . . .	306
7.4.8. Регуляризация и отсев . . . . .	308
7.4.9. Расширение данных . . . . .	313
7.4.10. Обучение более крупной модели . . . . .	318
7.5. Использование модели . . . . .	319
7.5.1. Загрузка модели . . . . .	319
7.5.2. Оценка модели . . . . .	320
7.5.3. Получение прогнозов . . . . .	321
7.6. Следующие шаги . . . . .	323
7.6.1. Упражнения . . . . .	323
7.6.2. Другие проекты . . . . .	324
Резюме . . . . .	325
Ответы к упражнениям . . . . .	325
<b>Глава 8. Бессерверное глубокое обучение . . . . .</b>	<b>326</b>
8.1. Бессерверно: AWS Lambda . . . . .	326
8.1.1. TensorFlow Lite . . . . .	328
8.1.2. Преобразование модели в формат TF Lite . . . . .	329
8.1.3. Подготовка изображений . . . . .	330
8.1.4. Использование модели TensorFlow Lite . . . . .	331
8.1.5. Код для лямбда-функции . . . . .	333
8.1.6. Подготовка образа Docker . . . . .	335
8.1.7. Отправка изображения в AWS ECR . . . . .	337
8.1.8. Создание лямбда-функции . . . . .	337
8.1.9. Создание API Gateway . . . . .	341

8.2. Следующие шаги . . . . .	346
8.2.1. Упражнения . . . . .	346
8.2.2. Другие проекты . . . . .	346
Резюме . . . . .	347
<b>Глава 9. Предоставление моделей с помощью Kubernetes и Kubeflow . . . . .</b>	<b>348</b>
9.1. Kubernetes и Kubeflow . . . . .	349
9.2. Предоставление моделей с помощью TensorFlow Serving . . . . .	350
9.2.1. Обзор архитектуры предоставления . . . . .	350
9.2.2. Формат saved_model . . . . .	352
9.2.3. Локальный запуск TensorFlow Serving . . . . .	353
9.2.4. Вызов модели TF Serving из Jupyter . . . . .	354
9.2.5. Создание службы Gateway . . . . .	358
9.3. Развёртывание модели с помощью Kubernetes . . . . .	362
9.3.1. Введение в Kubernetes . . . . .	362
9.3.2. Создание кластера Kubernetes на AWS . . . . .	364
9.3.3. Подготовка образов Docker . . . . .	366
9.3.4. Развёртывание в Kubernetes . . . . .	369
9.3.5. Тестирование сервиса . . . . .	375
9.4. Развёртывание модели с помощью Kubeflow . . . . .	376
9.4.1. Подготовка модели: загрузка ее в S3 . . . . .	377
9.4.2. Развёртывание моделей TensorFlow с помощью KFServing . . . . .	378
9.4.3. Доступ к модели . . . . .	379
9.4.4. Преобразователи KFServing . . . . .	381
9.4.5. Тестирование преобразователя . . . . .	384
9.4.6. Удаление кластера EKS . . . . .	384
9.5. Следующие шаги . . . . .	384
9.5.1. Упражнения . . . . .	385
9.5.2. Другие проекты . . . . .	385
Резюме . . . . .	385
<b>Приложение А. Подготовка среды . . . . .</b>	<b>387</b>
A.1. Установка Python и Anaconda . . . . .	387
A.1.1. Установка Python и Anaconda в Linux . . . . .	387
A.1.2. Установка Python и Anaconda в Windows . . . . .	389
A.1.3. Установка Python и Anaconda на macOS . . . . .	393
A.2. Запуск Jupyter . . . . .	393
A.2.1. Запуск Jupyter в Linux . . . . .	393
A.2.2. Запуск Jupyter в Windows . . . . .	395
A.2.3. Запуск Jupyter на macOS . . . . .	396

## 12 Оглавление

A.3. Установка Kaggle CLI . . . . .	397
A.4. Доступ к исходному коду . . . . .	398
A.5. Установка Docker . . . . .	399
A.5.1. Установка Docker в Linux . . . . .	399
A.5.2. Установка Docker в Windows . . . . .	400
A.5.3. Установка Docker на macOS . . . . .	400
A.6. Аренда сервера на AWS . . . . .	400
A.6.1. Регистрация на AWS . . . . .	401
A.6.2. Доступ к платежной информации . . . . .	405
A.6.3. Создание экземпляра EC2 . . . . .	407
A.6.4. Подключение к экземпляру . . . . .	414
A.6.5. Завершение работы экземпляра . . . . .	416
A.6.6. Настройка интерфейса AWS CLI . . . . .	417
<b>Приложение Б. Введение в Python . . . . .</b>	<b>418</b>
Б.1. Переменные . . . . .	418
Б.1.1. Поток управления . . . . .	421
Б.1.2. Коллекции . . . . .	423
Б.1.3. Повторное использование кода . . . . .	431
Б.1.4. Установка библиотек . . . . .	433
Б.1.5. Программы на Python . . . . .	433
<b>Приложение В. Введение в NumPy . . . . .</b>	<b>435</b>
В.1. NumPy . . . . .	435
В.1.1. Массивы NumPy . . . . .	436
В.1.2. Двумерные массивы NumPy . . . . .	441
В.1.3. Случайно сгенерированные массивы . . . . .	443
В.2. Операции NumPy . . . . .	445
В.2.1. Операции по элементам . . . . .	445
В.2.2. Суммирующие операции . . . . .	449
В.2.3. Сортировка . . . . .	451
В.2.4. Изменение формы и объединение . . . . .	452
В.2.5. Срезы и фильтрация . . . . .	456
В.3. Линейная алгебра . . . . .	458
В.3.1. Умножение . . . . .	458
В.3.2. Обратная матрица . . . . .	461
В.3.3. Нормальное уравнение . . . . .	463

<b>Приложение Г. Введение в Pandas . . . . .</b>	465
Г.1. Pandas . . . . .	465
Г.1.1. Датафрейм . . . . .	466
Г.1.2. Серии . . . . .	467
Г.1.3. Index . . . . .	469
Г.1.4. Доступ к строкам . . . . .	470
Г.1.5. Разделение датафрейма . . . . .	474
Г.2. Операции . . . . .	475
Г.2.1. Операции по элементам . . . . .	475
Г.2.2. Фильтрация . . . . .	477
Г.2.3. Операции со строками . . . . .	477
Г.2.4. Суммирующие операции . . . . .	481
Г.2.5. Отсутствующие значения . . . . .	482
Г.2.6. Сортировка . . . . .	484
Г.2.7. Группировка . . . . .	485
<b>Приложение Д. AWS SageMaker . . . . .</b>	487
Д.1. Блокноты AWS SageMaker . . . . .	487
Д.1.1. Увеличение лимитов квот графического процессора . . . . .	487
Д.1.2. Создание экземпляра блокнота . . . . .	490
Д.1.3. Обучение модели . . . . .	494
Д.1.4. Отключение блокнота . . . . .	495

# *Предисловие*

---

Я знаю Алексея уже более шести лет. Мы почти в одно время работали в команде анализа и обработки данных в IT-компании в Берлине: Алексей приступил к работе спустя несколько месяцев после моего ухода. Несмотря на это нам все же удалось познакомиться с помощью Kaggle, платформы для соревнований в области науки о данных, и благодаря одному общему другу. Мы состояли в одной команде в конкурсе Kaggle по обработке естественного языка, интересном проекте, который требовал тщательного использования предварительно обученных механизмов встраивания слов и их умелого смешения. В то время Алексей писал книгу и поэтому попросил меня быть научным редактором. Книга была о Java и науке о данных, и, читая ее, я особенно был впечатлен тем, насколько скрупулезно Алексей планировал и прорабатывал интересные примеры. Вскоре последовал новый этап сотрудничества: мы стали соавторами проектной книги о TensorFlow и работали над различными проектами: от обучения с подкреплением до рекомендательных систем, которые должны были стать источником вдохновения и примером для читателей.

Работая с Алексеем, я заметил, что он предпочитает учиться на практике и писать код, как и многие другие, кто перешел в data science из разработки.

Поэтому я не очень удивился, узнав, что он взялся за еще одну книгу, основанную на проекте. Алексей попросил меня оставить отзыв о его работе, так что я читал книгу еще на этапе рукописи, и она меня чрезвычайно увлекла. Книга представляет собой практическое введение в машинное обучение с упором на практику. Она написана для людей, имеющих тот же опыт, который есть у Алексея, — для тех разработчиков, кто интересуется data science и нуждается в том, чтобы быстро получить полезный опыт работы с данными, который можно было бы применять в своей работе.

Как автор более десятка книг по науке о данных и ИИ, я знаю, что на эту тему создано уже множество книг и курсов. Однако эта книга не похожа на другие. На ее страницах вы не найдете тех же избитых задач по работе с данными, которые предлагаются в других книгах. Нет здесь и педантичной, повторяющейся череды тем, которая подобна проторенной дороге, всегда ведущей в места, где вы уже были.

Все в книге вращается вокруг практических и почти реальных примеров. Вы узнаете, как спрогнозировать цену автомобиля, предсказать отток клиентов и оценить риск невозврата кредита. Затем вы классифицируете фотографии одежды, распределив их по категориям: футболки, платья, брюки и др. Проект особенно любопытен тем, что Алексей лично курировал этот набор данных, а вы можете обогатить его одеждой из собственного гардероба.

Прочитав эту книгу, вы, конечно, научитесь применять машинное обучение для решения распространенных проблем и будете использовать наиболее простые и эффективные методы достижения наилучших результатов. Первые главы начинаются с описания основных алгоритмов, таких как линейная и логистическая регрессия. Затем читатель постепенно переходит к градиентному бустингу и нейронным сетям. Тем не менее сильной стороной книги является то, что, рассказывая о машинном обучении на практике, она также готовит вас к реальному миру. Вы столкнетесь с несбалансированными классами и распределениями с длинным хвостом, а также узнаете, как обрабатывать грязные данные. Вы оцените собственные модели и развернете их с помощью AWS Lambda и Kubernetes. И это лишь некоторые из новых техник, которые вы встретите на страницах этой книги.

Рассуждая с позиции инженера, можно сказать, что книга составлена таким образом, что вы получите те основные 20 % знаний, которые охватывают 80 % объема, позволяющего стать специалистом по обработке данных. Что еще более важно, вы будете читать и практиковаться под руководством Алексея, знания и умения которого основаны на его работе и опыте в Kaggle. На этой ноте я пожелаю вам доброго пути по страницам и проектам книги. Я уверен, что она поможет вам найти подход к науке о данных и ее проблемам, инструментам и решениям.

*Лука Массарон*

# *Введение*

---

Я начинал свою карьеру, будучи разработчиком Java. Примерно в 2012–2013 годах я заинтересовался наукой о данных (data science) и машинным обучением. Сначала я смотрел онлайн-курсы, а затем поступил на магистерскую программу и в течение двух лет изучал различные аспекты бизнес-аналитики и науки о данных. В итоге я окончил университет в 2015 году и начал работать специалистом по обработке данных.

Мой коллега познакомил меня с Kaggle — платформой для соревнований в области науки о данных. Я подумал: «Имея все навыки, которые я получил на курсах, и мою степень магистра, я смогу легко победить в любом соревновании». Но когда я начал участвовать в соревнованиях, меня ждал полный провал. Все мои теоретические знания оказались бесполезны на Kaggle. Мои модели были ужасны, так что в итоге я оказался в самом конце рейтинга.

Следующие девять месяцев я провел, участвуя в соревнованиях, связанных с наукой о данных. У меня не очень хорошо получалось, но именно тогда я по-настоящему научился машинному обучению.

Я понял, что для меня лучший способ учиться — это работать над проектами. Со-средоточиваясь на задаче, что-то внедряя, экспериментируя, я действительно учусь. Но если фокусируюсь на курсах и теории, то трачу слишком много времени на изучение вещей, которые оказываются неважными и бесполезными на практике.

И я не одинок. Рассказывая эту историю, я много раз слышал: «И у меня так!» Вот почему основное внимание в книге уделяется обучению в процессе выполнения проектов. Я считаю, что инженеры-программисты — люди с таким же, как у меня, опытом — лучше всего учатся на практике.

Мы начнем с проекта прогнозирования цен на автомобили и изучим линейную регрессию. Затем определим, не хотят ли клиенты прекратить пользоваться

услугами нашей компании. Для этого изучим логистическую регрессию. Чтобы изучить деревья принятия решений, мы оценим клиентов банка и определим, в силах ли они погасить кредит. Наконец, мы используем глубокое обучение, чтобы распределить изображение одежды по таким категориям, как футболки, брюки, обувь, верхняя одежда и т. д.

Каждый проект в книге начинается с описания задачи. Затем мы ее решаем, используя различные инструменты и фреймворки. Мы сосредоточиваемся на задаче и охватываем только те аспекты, которые важны для ее решения. Встречается и теория, но я свожу ее к минимуму и фокусируюсь на практической части.

Тем не менее иногда мне приходилось включать формулы в некоторые главы. Не-возможно совсем избавиться от формул в книге о машинном обучении. Я знаю, что они пугают некоторых из нас. Меня они тоже пугали. Вот почему я также объясняю все формулы с помощью кода. Увидев формулу, не позволяйте ей испугать вас. Сначала попробуйте понять код, а затем вернитесь к формуле, чтобы увидеть, как код преобразуется в нее. Тогда она уже не будет такой устрашающей!

В книге вы не найдете всех возможных тем. Я сосредоточился на самых фундаментальных вещах — тех концепциях, которые вы будете использовать со 100 %-ной вероятностью, начав работать с машинным обучением. Есть и другие важные темы, которые я не затронул: анализ временных рядов, кластеризация, обработка естественного языка. Прочитав данную книгу, вы будете иметь базовые знания, которые позволят вам изучить эти темы самостоятельно.

Три главы в книге посвящены развертыванию модели. Это крайне важные главы — может быть, самые важные. Возможность развертывания модели определяет разницу между успешным проектом и неудачным. Даже самая лучшая модель бесполезна, если никто иной не сможет ее использовать. Вот почему стоит потратить время на то, чтобы узнать, как сделать ее доступной для других. И именно по этой причине я рассказываю об этом почти в начале, сразу после того, как мы изучим логистическую регрессию.

Последняя глава посвящена развертыванию моделей с помощью Kubernetes. Это непростая глава, но в настоящее время Kubernetes — наиболее часто используемая система управления контейнерами. Вполне вероятно, что вам придется с ней работать, и именно поэтому она включена в книгу. Наконец, каждая глава книги содержит упражнения. Может возникнуть соблазн пропустить их, но я не рекомендую этого делать. Даже просто изучая данную книгу, вы узнаете много нового. Но если вы не примените эти знания на практике, то довольно быстро забудете большую их часть. Упражнения помогут вам применить эти новые навыки на практике, и вы гораздо лучше запомните то, что узнали.

Желаю вам получить удовольствие от чтения! Не стесняйтесь связаться со мной в любое время!

*Алексей Григорьев*

# *Благодарности*

---

На работу над книгой у меня ушло много времени. Я провел за ней бесчисленное количество вечеров и бессонных ночей. Вот почему прежде всего я хотел бы поблагодарить свою жену за терпение и поддержку.

Далее я хотел бы поблагодарить моего редактора Сьюзан Этридж за ее терпение. Первая ознакомительная версия книги была выпущена в январе 2020 года. Вскоре после этого мир вокруг нас сошел с ума, и все оказались запертыми дома. Работа над книгой стала для меня чрезвычайно сложной задачей. Я не знаю, сколько дедлайнов я пропустил (очень много!), но Сьюзен не давила на меня и позволяла мне работать в моем собственном темпе.

Первым человеком, который должен был прочитать все главы (после Сьюзен), был Майкл Лунд. Я хотел бы поблагодарить его за бесценную обратную связь и за все комментарии, которые он оставил на моих черновиках. Один из рецензентов написал, что «внимание к деталям в книге просто поразительное», и главная причина этого — вклад Майкла.

Найти мотивацию для работы над книгой во время карантина было непросто. Иногда у меня вообще не оставалось никаких сил. Но отзывы рецензентов и читателей МЕАР заставляли меня двигаться вперед. Все это помогло мне закончить книгу, несмотря на все трудности. Итак, я хотел бы поблагодарить вас всех за чтение черновиков, за вашу обратную связь и — самое главное — ваши добрые слова и поддержку!

Хочу отдельно поблагодарить некоторых читателей, поделившихся со мной своими отзывами: Мартина Шенделя, Агнешку Каминскую и Алексея Швеца. Кроме того, я хотел бы поблагодарить всех, кто оставил отзыв в разделе комментариев LiveBook или на канале #ml-bookcamp в группе DataTalks. Club Slack.

В главе 7 я использую набор данных с одеждой для проекта классификации изображений. Он создавался и подбирался специально для этой книги. Я хотел бы поблагодарить всех, кто предоставил изображения своей одежды, особенно Кенеса Шангеря и Тагиаса, обеспечивших 60 % всего набора данных.

Последнюю главу я посвятил развертыванию модели с помощью Kubernetes и Kubeflow. Kubeflow – это относительно новая технология, и некоторые аспекты еще недостаточно хорошо документированы. Вот почему я хотел бы поблагодарить своих коллег, Теофилоса Папапанайоту и Антонио Бернардино, за их помощь с Kubeflow.

Большинство читателей не смогли бы получить эту книгу без помощи отдела маркетинга издательства «Мэннинг». Я хотел бы отдельно поблагодарить Лану Класич и Радмилу Эрцеговац за их помощь в организации мероприятий по продвижению книги и за проведение кампаний в социальных сетях для привлечения большего количества читателей. Я также хотел бы поблагодарить моих редакторов Дейдре Хайам, Адриану Сабо, Памелу Хант и корректора Мелоди Долаб.

Всем рецензентам: Адаму Гладстону, Амарешу Раджасекхарану, Эндрю Кортеру, Бену Макнамаре, Билли О'Каллагану, Чаду Дэвису, Кристоферу Коттмайеру, Кларку Дорману, Дэну Шейху, Джорджу Томасу, Густаво Филипе Рамосу Гомесу, Джозефу Перенье, Кришне Чайтанье Анипинди, Ксении Легостай, Лурду Мата Редди Куниредди, Майку Кадди, Монике Гимарайни, Наге Паван Кумар Ти, Натану Дельбу, Нур Тавил, Оливеру Кортену, Полу Силистяну, Рами Мадиан, Себастьяну Мохану, Шону Ламу, Вишвеш Рави Шримали, Уильяму Помпей – ваши предложения помогли улучшить эту книгу.

И последнее, но не менее важное: я хотел бы поблагодарить Луку Массарона за то, что он вдохновил меня на написание книг. Я никогда не стану таким плодовитым автором книг, как ты, Лука, но спасибо тебе за то, что послужил для меня отличной мотивацией!

# *Об этой книге*

---

## **КОМУ АДРЕСОВАНА КНИГА**

Эта книга написана для людей, которые умеют программировать и могут быстро освоить основы Python. Вам не понадобится предварительный опыт работы с машинным обучением.

Идеальный читатель – инженер-программист, который хотел бы начать работать с машинным обучением. Однако мотивированному студенту вуза, которому нужно писать код для учебы и сторонних проектов, книга, несомненно, пригодится.

Кроме того, книга будет полезна и людям, которые уже работают с машинным обучением, но хотят узнать больше. Многие из тех, кто уже работает в качестве специалистов по обработке данных и аналитиков данных, сказали, что она оказалась полезной для них, особенно главы о развертывании.

## **СТРУКТУРА ИЗДАНИЯ**

В книге девять глав, и в них мы работаем над четырьмя различными проектами.

- В главе 1 мы лишь приблизимся к теме: обсудим разницу между традиционной разработкой программного обеспечения и машинным обучением. Мы рассмотрим процесс организации проектов машинного обучения, начиная с этапа понимания бизнес-требований и заканчивая заключительным этапом развертывания модели. Мы более подробно изучим этап моделирования

в процессе и поговорим о том, как можно оценить наши модели и выбрать из них лучшую. Проиллюстрируем концепции этой главы с помощью задачи обнаружения спама.

- В главе 2 мы начнем наш первый проект: прогноз цены автомобиля. Мы узнаем, как использовать для этого линейную регрессию. Сначала мы подготовим набор данных и выполним легкую очистку данных. Затем проведем предварительный анализ данных, чтобы лучше в них разобраться. После этого собственноручно реализуем модель линейной регрессии с помощью NumPy, что позволит понять модели машинного обучения с точки зрения внутреннего устройства. Наконец, мы обсудим такие темы, как регуляризация и оценка качества модели.
- В главе 3 мы рассмотрим проблему обнаружения оттока. Мы работаем в телекоммуникационной компании и хотим определить, кто из клиентов вскоре захочет прекратить пользоваться нашими услугами. Это проблема классификации, которую мы решаем с помощью логистической регрессии. Мы начнем с анализа важности признаков, чтобы понять, какие факторы являются наиболее значимыми для данной задачи. Затем обсудим прямое кодирование как способ обработки категориальных переменных (таких факторов, как пол, тип контракта и т. д.). Наконец, с помощью Scikit мы научим модель логистической регрессии понимать, какие клиенты скоро от нас уйдут.
- В главе 4 мы возьмем модель, разработанную в главе 3, и оценим ее эффективность. Мы рассмотрим наиболее важные показатели оценки классификации: достоверность, точность и отклик. Обсудим таблицу запутанности, после чего перейдем к подробностям анализа ROC и рассчитаем AUC. А завершим мы эту главу обсуждением К-кратной перекрестной проверки.
- В главе 5 возьмем модель прогнозирования оттока и развернем ее как веб-сервис. Это важный шаг в изучении: если мы не сделаем нашу модель доступной, то она в итоге окажется бесполезной. Мы начнем с Flask, фреймворка Python для создания веб-сервисов. Затем рассмотрим Pipenv и Docker, предназначенные для управления зависимостями, и закончим развертыванием нашего сервиса на AWS.
- В главе 6 мы начнем проект по оценке рисков. Нужно будет понять, возникнут ли у клиента банка проблемы с выплатой кредита. Для этого мы узнаем, как работают деревья принятия решений, и обучим простую модель с помощью Scikit-learn. Затем перейдем к более сложным древовидным моделям, таким как случайный лес и градиентный бустинг.
- В главе 7 мы создадим проект классификации изображений. Обучим модель распределять изображения одежды по десяти категориям, таким как футболки, платья, брюки и т. д. Для обучения нашей модели воспользуемся TensorFlow и Keras и разберем такие темы, как обучение с переносом опыта, что даст нам возможность обучать модель на относительно небольшом наборе данных.

## 22    Об этой книге

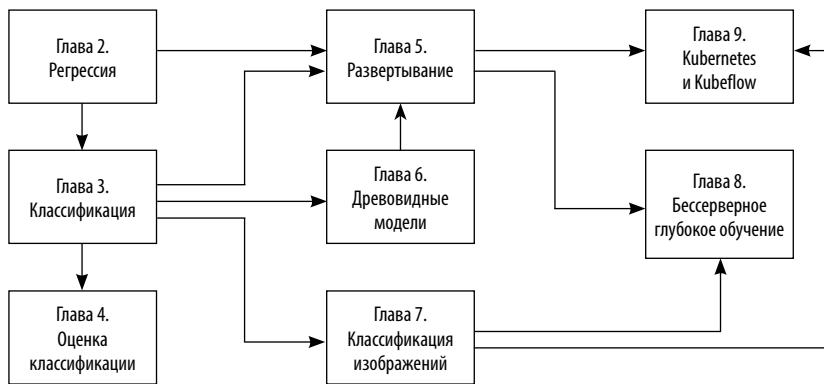
- В главе 8 мы возьмем модель классификации одежды, которую обучали в главе 7, и развернем ее с помощью TensorFlow Lite и AWS Lambda.
- В главе 9 развернем модель классификации одежды, но в первой части используем Kubernetes и TensorFlow, а во второй — Kubeflow Serving.

Чтобы помочь вам начать работу с книгой, а также с Python и связанными с ним библиотеками, я подготовил пять приложений:

- в приложении А объясняется, как настроить среду для книги. Мы покажем, как установить Python с помощью Anaconda, запустить Jupyter Notebook, установить Docker и создать учетную запись AWS;
- приложение Б посвящено основам Python;
- приложение В охватывает основы NumPy и содержит краткое введение в наиболее важные концепции линейной алгебры, которые нужны для машинного обучения: умножение и обращение матриц;
- в приложении Г речь пойдет о Pandas;
- в приложении Д объясняется, как получить Jupyter Notebook с GPU на AWS Sage-Maker.

Эти приложения необязательны, но они могут оказаться полезными, особенно если вы раньше не работали с Python или AWS.

Вам не нужно читать книгу целиком и полностью. Ориентироваться в материале вам поможет следующая карта:



Главы 2 и 3 наиболее важны. Все остальные главы зависят от них. Прочитав их, вы переходите к главе 5, чтобы развернуть модель, главе 6, чтобы узнать о древовидных моделях, или главе 7, чтобы узнать о классификации изображений. Глава 4, посвященная метрикам оценки, зависит от главы 3: мы будем оценивать качество модели прогнозирования оттока, представленной в главе 3. В главах 8 и 9 мы развернем модель классификации изображений, поэтому полезно пропустить главу 7, прежде чем переходить к главам 8 или 9.

Каждая глава содержит упражнения. Выполнять их очень важно — они помогут вам намного лучше запомнить материал.

## О КОДЕ

В этой книге приведено множество примеров исходного кода в виде как пронумерованных, так и обычных листингов. В обоих случаях исходный код оформляется шрифтом **фиксированной ширины**, подобным этому, чтобы отделить его от обычного. Иногда код также выделяется **жирным** шрифтом для той его части, которая изменилась по сравнению с предыдущими шагами в главе, например когда новая функция добавляется к существующей строке кода.

Во многих случаях исходный код был переформатирован; мы добавили разрывы строк и переработали отступы, чтобы учесть доступное пространство на странице. В редких случаях даже этого оказалось недостаточно, и такие листинги включают маркеры продолжения строки (➡). Кроме того, комментарии в исходном коде часто удалялись, если код описывался в тексте. Важные концепции представлены также в пояснениях к коду листингов.

Код из книги доступен на GitHub по адресу <https://github.com/alexeygrigorev/mlbookcamp-code>. Этот репозиторий также содержит множество полезных ссылок, которые вам пригодятся в ходе изучения темы машинного обучения.

## ДРУГИЕ ОНЛАЙН-РЕСУРСЫ

- Сайт книги <https://mlbookcamp.com/> содержит полезные статьи и курсы, основанные на книге.
- Сообщество энтузиастов обработки данных: <https://datatalks.club>. Там вы можете задать любой вопрос о данных или машинном обучении.
- Существует также канал для обсуждения вопросов, связанных с книгами: #ml-bookcamp.

## ОТ ИЗДАТЕЛЬСТВА

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

## *Об авторе*

---

**Алексей Григорьев** проживает в Берлине со своей женой и сыном. Он опытный инженер-программист, специализирующийся на машинном обучении. Трудится в OLX Group главным специалистом по обработке данных, помогая своим коллегам внедрять машинное обучение в производство.

В свободное от работы время Алексей ведет DataTalks.Club, сообщество людей, которые любят науку о данных и машинное обучение. Кроме того, он является автором еще двух книг: *Mastering Java for Data Science* и *TensorFlow Deep Learning Projects*.

## *Иллюстрация на обложке*

---

Фигура на обложке — *Femme de Brabant*, или женщина из Брабанта. Иллюстрация взята из коллекции парадных костюмов из разных стран Жака Грассе де Сен-Совера (1757–1810) под названием *Costumes de Différents Pays*, опубликованной во Франции в 1797 году. Каждая иллюстрация превосходно прорисована и раскрашена вручную. Богатое разнообразие коллекции Грассе де Сен-Совера живо напоминает нам о том, насколько культурно обособленными были города и регионы мира всего 200 лет назад. Изолированные друг от друга, люди говорили на разных диалектах и языках. На улицах или в сельской местности одежда позволяла легко определить, где проживают люди, чем занимаются или каково их положение в обществе.

С тех пор наша манера одеваться изменилась, и региональное разнообразие, столь заметное в то время, практически исчезло. Сейчас трудно различить жителей разных континентов, не говоря уже о разных городах, регионах или странах. Возможно, мы обменяли культурное разнообразие на более разнообразную личную жизнь и уж точно на более разнообразную и быстро развивающуюся технологическую.

Во времена, когда трудно отличить одну компьютерную книгу от другой, издательство Manning подчеркивает изобретательность и находчивость компьютерного бизнеса, создавая обложки книг, основанные на богатом разнообразии региональной жизни двухвековой давности, оживленной благодаря работам Грассе де Сен-Совера.

# 1

## *Введение в машинное обучение*

### **В этой главе**

- ✓ Что такое машинное обучение и какие задачи оно может решать.
- ✓ Организация успешного проекта машинного обучения.
- ✓ Обучение и выбор моделей машинного обучения.
- ✓ Проверка модели.

В этой главе мы познакомимся с машинным обучением и опишем случаи, в которых оно наиболее полезно. Мы покажем, чем проекты машинного обучения отличаются от традиционной разработки программного обеспечения (решений, основанных на правилах), и проиллюстрируем различия на примере системы обнаружения спама.

Чтобы решать реальные проблемы с помощью машинного обучения, нам понадобится способ организации проектов машинного обучения. В этой главе мы поговорим о CRISP-DM — пошаговой методологии для реализации эффективных проектов машинного обучения.

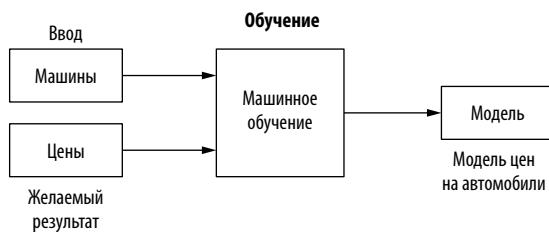
Наконец мы более подробно рассмотрим один из этапов CRISP-DM — моделирование. На данном этапе мы обучаем различные модели и выбираем ту, которая лучше всего решает нашу задачу.

## 1.1. МАШИННОЕ ОБУЧЕНИЕ

Машинное обучение — часть прикладной математики и информатики. Закономерности в данных выявляются в нем с помощью инструментов из математических дисциплин, таких как теория вероятностей, статистика и теория оптимизации.

Основная идея машинного обучения заключается в обучении на примерах: мы готовим набор данных с примерами, а система машинного обучения «учится» на нем. Другими словами, мы даем системе входные данные и желаемый результат, а она пытается выяснить, как выполнять такое преобразование автоматически, не спрашивая человека.

Например, мы можем собрать набор данных с описаниями автомобилей и их ценами. Затем мы предоставляем модель машинного обучения с этим набором данных и «обучаем» ее, демонстрируя ей автомобили и их цены. Этот процесс называется *обучением* или иногда *подгонкой* (рис. 1.1).



**Рис. 1.1.** Алгоритм машинного обучения принимает входные данные (описания автомобилей) и желаемый результат (цены на автомобили). На основе этих данных он создает модель

Когда обучение завершится, мы можем использовать модель, попросив ее спрогнозировать цены на автомобили, которые мы еще не встречали (рис. 1.2).



**Рис. 1.2.** По завершении обучения у нас имеется модель, которую можно применить к новым входным данным (автомобили без цен), чтобы получить выходные данные (прогноз цен)

Все, что нам нужно для машинного обучения, — это набор данных, в котором с каждым входным элементом (автомобилем) сопоставлен желаемый результат (цена).

## 28 Глава 1. Введение в машинное обучение

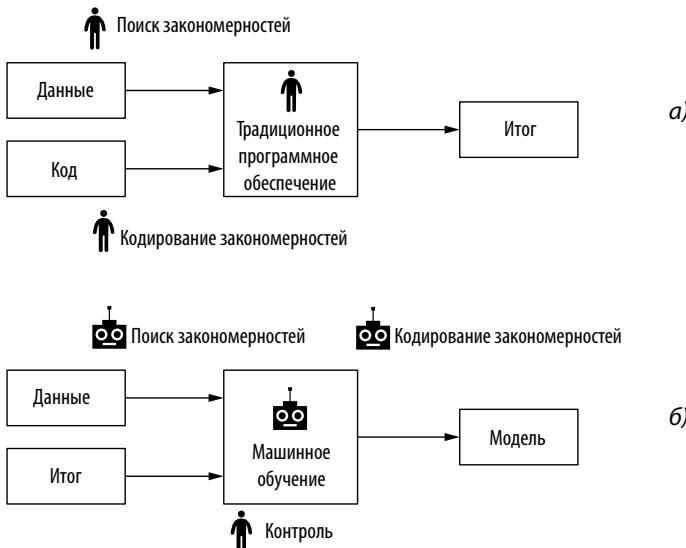
Этот процесс сильно отличается от традиционной разработки программного обеспечения. При отсутствии машинного обучения аналитики и разработчики просматривают имеющиеся у них данные и пытаются отыскать закономерности вручную. После этого они придумывают некую логику: набор правил для преобразования входных данных в желаемый результат. Затем они явно кодируют эти правила с помощью языка программирования, такого как Java или Python, а результат называется программным обеспечением.

Таким образом, в отличие от машинного обучения, всю сложную работу выполняет человек (рис. 1.3).



**Рис. 1.3.** В традиционном программном обеспечении закономерности обнаруживаются вручную, а затем кодируются с помощью языка программирования. Всю работу делает человек

Итоговая разница между традиционной программной системой и системой, основанной на машинном обучении, отражена на рис. 1.4. В машинном обучении



**Рис. 1.4.** Разница между традиционной программной системой и системой машинного обучения. В традиционной разработке ПО мы выполняем всю работу, в то время как в машинном обучении мы делегируем обнаружение закономерностей машине

мы даем системе входные и выходные данные и в результате получаем модель (код), которая может преобразовывать входные данные в выходные. Сложную работу выполняет машина; нам нужно лишь контролировать процесс обучения, чтобы убедиться, что модель получилась достаточно хорошей (рис. 1.4, б). И на-против, в традиционных системах мы сначала самостоятельно ищем закономерности в данных, а затем пишем код, который преобразует данные в желаемый результат, используя эти обнаруженные вручную закономерности (рис. 1.4, а).

### **1.1.1. Машинное обучение в сравнении с системами, основанными на правилах**

Чтобы проиллюстрировать разницу между этими двумя подходами и показать, почему машинное обучение полезно, рассмотрим конкретный случай. Поговорим о системе обнаружения спама.

Предположим, мы запускаем сервис электронной почты, и пользователи начинают жаловаться на нежелательные электронные письма с рекламой. Чтобы решить эту проблему, мы хотим создать систему, которая помечает нежелательные сообщения как спам и переправляет их в соответствующую папку.

Очевидный способ решить проблему — самим просмотреть эти электронные письма, чтобы увидеть, есть ли в них какая-либо закономерность. Например, мы можем проверить отправителя и содержимое.

Если мы обнаружим, что в спам-сообщениях действительно есть закономерность, то запишем обнаруженные закономерности и зададим простые правила для перехвата этих сообщений:

- если отправитель — `promotions@online.com`, то это спам;
- если заголовок содержит «купить сейчас со скидкой 50 %», а домен отправителя — `online.com`, то это спам;
- иначе — «полезное электронное письмо».

Мы пишем эти правила на Python и создаем сервис обнаружения спама, который успешно развертываем. Вначале система работает хорошо и отлавливает весь спам, но некоторое время спустя сквозь нее начинают просачиваться новые спам-сообщения. Существующие правила больше не позволяют помечать эти сообщения как спам.

Чтобы решить эту проблему, мы анализируем содержимое новых сообщений и обнаруживаем, что в большинстве из них есть слово «вклад». Итак, мы добавляем новое правило:

- если отправитель — `promotions@online.com`, то это спам;
- если заголовок содержит «купить сейчас со скидкой 50 %», а домен отправителя — `online.com`, то это спам;

## 30 Глава 1. Введение в машинное обучение

- если текст содержит слово «вклад», то это спам;
- иначе — «полезное электронное письмо».

Обнаружив это правило, мы внедряем исправление в наш сервис Python и начинаем ловить больше спама на радость нашим пользователям.

Однако какое-то время спустя пользователи снова начинают жаловаться: некоторые люди используют слово «вклад» с благими намерениями, но наша система не распознает этот факт и помечает сообщения как спам. Чтобы решить эту проблему, мы смотрим на полезные сообщения и пытаемся понять, чем они отличаются от спама. Через некоторое время мы обнаруживаем ряд закономерностей и снова изменяем правила:

- если отправитель — `promotions@online.com`, то это спам;
- если заголовок содержит «купить сейчас со скидкой 50 %», а домен отправителя — `online.com`, то это спам;
- если тело письма содержит «вклад», то:
  - если домен отправителя — `test.com`, то это спам;
  - если длина описания  $\geq 100$  слов, то это спам;
- иначе — «полезное электронное письмо».

В этом примере мы просмотрели входные данные вручную и проанализировали их в попытке извлечь из них закономерности. В результате анализа мы получили набор правил, которые преобразуют входные данные (электронные письма) в один из двух возможных вариантов выходных данных: спам или не спам.

Теперь представьте, что мы повторяем этот процесс несколько сотен раз. В результате мы получаем код, который довольно сложно поддерживать и даже понимать. В какой-то момент становится невозможным добавлять в код новые правила, не нарушая существующую логику. Таким образом, в долгосрочной перспективе довольно сложно сопровождать и корректировать существующие правила так, чтобы спам-фильтр по-прежнему хорошо работал, сводя к минимуму жалобы на спам.

Это именно та ситуация, в которой может помочь машинное обучение. В ходе данного процесса мы обычно не пытаемся извлекать эти закономерности вручную. Вместо этого мы делегируем задачу статистическим методам, предоставляя системе набор данных с электронными письмами, помеченными как спам или не спам, и описывая каждый объект (электронное письмо) с набором его характеристик (признаков). Основываясь на этой информации, система пытается отыскать закономерности в данных уже без помощи человека. В конце концов, она узнает, как комбинировать признаки таким образом, чтобы спам-сообщения помечались как спам, а нужные сообщения — нет.

Благодаря машинному обучению проблема поддержания созданного вручную набора правил исчезает. Когда появляется новая закономерность — например,

новый тип спама, — мы, вместо того чтобы вручную корректировать существующий набор правил, просто предоставляем алгоритму машинного обучения новые данные. В итоге он извлекает новые важные закономерности из новых данных, не повреждая уже существующие — при условии, что они все еще актуальны и присутствуют в новых.

Посмотрим, как мы можем использовать машинное обучение для решения проблемы классификации спама. Для этого нам сначала придется представить каждое электронное письмо с помощью набора признаков. Для начала мы можем выбрать следующие:

- длина заголовка  $> 10?$  `true/false`;
- длина тела письма  $> 10?$  `true/false`;
- отправитель `promotions@online.com?` `true/false`;
- отправитель `hpYOSKmL@test.com?` `true/false`;
- домен отправителя `test.com?` `true/false`;
- описание содержит «вклад»? `true/false`.

В данном конкретном случае мы описываем все электронные письма с помощью набора из шести признаков. Так совпало, что эти признаки являются производными от рассмотренных ранее правил.

С помощью этого набора мы можем закодировать любое электронное письмо в виде вектора признаков: последовательности чисел, содержащей все значения признаков для конкретного электронного письма.

Теперь представьте, что у нас есть электронное письмо, которое пользователи пометили как спам (рис. 1.5). Мы можем задать это электронное письмо в виде вектора  $[1, 1, 0, 0, 1, 1]$ , и для каждого из шести признаков мы кодируем значение как 1 для `true` или 0 для `false` (рис. 1.6). Поскольку наши пользователи пометили сообщение как спам, целевая переменная равна 1 (`true`).

<b>Тема:</b> Жду вашего ответа <b>От:</b> prince1@test.com
Мы рады сообщить вам, что вы выиграли 1.000.000 (один миллион) долларов США. Чтобы получить выигрыш, вам необходимо оплатить небольшой сбор за обработку заявки. Пожалуйста, переведите 10 долларов США на наш счет PayPal по адресу prince@test.com. Как только мы получим деньги, мы начнем перевод. Еще раз поздравляем!
<b>Спам:</b> true

**Рис. 1.5.** Электронное письмо, помеченное пользователем как спам



**Рис. 1.6.** Шестимерный вектор признаков для спам-сообщения электронной почты. Каждый из шести признаков представлен числом. В этом случае мы используем 1, если признак истинный, и 0, если ложный

Таким образом, мы можем создать векторы признаков для всех электронных писем в нашей базе данных и прикрепить метку к каждому из них. Эти векторы станут входными данными для модели. Затем она берет все эти числа и объединяет признаки таким образом, чтобы прогноз для спам-сообщений был близок к 1 (спам) и равен 0 (не спам) для обычных сообщений (рис. 1.7).



**Рис. 1.7.** Входные данные для алгоритма машинного обучения состоят из нескольких векторов признаков и целевой переменной для каждого вектора

В результате у нас появляется более гибкий инструмент, нежели набор жестко прописанных правил. Если что-то изменится в будущем, то нам не придется пересматривать все правила вручную и пытаться их реорганизовать. Вместо этого мы используем только самые последние данные и заменим старую модель новой.

Этот пример демонстрирует лишь один из способов, с помощью которого машинное обучение может облегчить нашу жизнь.

Другие области применения машинного обучения включают в себя:

- прогнозирование цены автомобиля;
- прогнозирование того, прекратит ли клиент пользоваться услугами компании;
- упорядочивание документов по релевантности по отношению к запросу;

- показ пользователям объявлений, на которые они с большей вероятностью перейдут, вместо нерелевантных;
- классификация вредных и некорректных правок в «Википедии». Подобная система может помочь модераторам «Википедии» расставить приоритеты при проверке предлагаемых правок;
- рекомендации товаров, которые могут купить клиенты;
- классификация изображений по категориям.

Само собой, применение машинного обучения не ограничивается этими примерами. Мы можем использовать буквально все, что можно выразить как (**входные данные, желаемый результат**), для обучения модели машинного обучения.

### **1.1.2. Когда машинное обучение бесполезно**

Машинное обучение полезно и помогает решить множество задач, однако в некоторых случаях в нем нет необходимости.

Для ряда простых задач часто хорошо работают правила и эвристика, поэтому лучше начать с них, а уже затем рассмотреть возможность использования машинного обучения. В нашем примере со спамом мы начали с создания набора правил, но после того, как поддерживать этот набор стало сложно, переключились на машинное обучение. Однако мы использовали некоторые правила в качестве признаков и просто переработали их в модель.

В некоторых случаях просто невозможно применить машинное обучение. Чтобы его использовать, нам нужны данные. Если они отсутствуют, то машинное обучение невозможно.

### **1.1.3. Контролируемое машинное обучение**

Проблема классификации электронной почты, которую мы только что рассмотрели, выступает примером контролируемого обучения: мы предоставляем модели признаки и целевую переменную, а она выясняет, как с их помощью достигать цели. Этот тип обучения называется **контролируемым**, поскольку мы контролируем или обучаем модель, показывая ей примеры, точно так же, как мы учили бы ребенка, показывая ему изображения различных объектов, а затем называя их.

Немного более формально мы можем выразить модель контролируемого машинного обучения математически как

$$y \approx g(X),$$

## 34 Глава 1. Введение в машинное обучение

где:

- $g$  — функция, которую мы хотим изучить с помощью машинного обучения;
- $X$  — матрица признаков, в которой строки являются векторами признаков;
- $y$  — целевая переменная: вектор.

Цель машинного обучения состоит в том, чтобы изучить эту функцию  $g$  таким образом, чтобы, когда она получает матрицу  $X$ , выходные данные были близки к вектору  $y$ . Другими словами, функция  $g$  должна быть способна принимать  $X$  и производить  $y$ . Процесс изучения  $g$  обычно называют *обучением* или *подгонкой*. Мы «подгоняем»  $g$  к набору данных  $X$  таким образом, чтобы он получал  $y$  (рис. 1.8).



**Рис. 1.8.** Когда мы обучаем модель, алгоритм принимает матрицу  $X$ , в которой векторы признаков являются строками, а желаемый результат — вектором  $y$  со всеми значениями, которые мы хотим спрогнозировать. Результатом обучения является  $g$ , модель. После обучения  $g$  должна производить  $y$  при применении к  $X$  — или, если коротко,  $g(X) \approx y$

Существуют различные типы задач контролируемого обучения, и тип зависит от целевой переменной  $y$ . Основные типы представлены ниже.

- Регрессия — целевая переменная  $y$  является числовой, например цена автомобиля или завтрашняя температура. Мы рассмотрим регрессионные модели в главе 2.
- Классификация — целевая переменная  $y$  является категориальной, например спам, не спам или марка автомобиля. Далее мы можем разделить классификацию на две подкатегории: 1) *двоичная классификация*, которая имеет только два возможных результата, таких как спам или не спам; и 2) *многоклассовая классификация*, которая имеет более двух возможных исходов, таких как марка автомобиля (Toyota, Ford, Volkswagen и т. д.). Классификация, особенно двоичная, — пример наиболее распространенного применения машинного обучения. Мы рассматриваем ее в нескольких главах на протяжении всей книги, начиная с главы 3. В текущей же главе мы построим модель, которая позволит спрогнозировать, прекратит ли клиент пользоваться услугами нашей компании.

- Ранжирование — целевая переменная  $y$  представляет собой порядок элементов внутри группы, например порядок страниц на странице результатов поиска. Проблема ранжирования часто возникает в таких областях, как поиск и рекомендации, но эта тема выходит за рамки данной книги, и мы не будем рассматривать ее подробно.

Каждую задачу контролируемого обучения можно решить с помощью различных алгоритмов. Нам доступно множество типов моделей. Эти модели определяют, как именно функция  $g$  учится прогнозировать  $y$  на основе  $X$ . Модели включают в себя:

- линейную регрессию для решения задачи регрессии (описывается в главе 2);
- логистическую регрессию для решения задачи классификации (описывается в главе 3);
- древовидные модели для решения задач как регрессии, так и классификации (описываются в главе 6);
- нейронные сети для решения как регрессионных задач, так и задач классификации (описываются в главе 7).

Глубокому обучению и нейронным сетям в последнее время уделяется особое внимание, в основном благодаря прорыву в методах компьютерного зрения. Эти сети решают такие задачи, как классификация изображений, намного лучше, чем это делали более ранние методы. *Глубокое обучение* — подобласть машинного обучения, в которой функция  $g$  представляет собой нейронную сеть со многими слоями. Мы узнаем больше о нейронных сетях и глубоком обучении, начиная с главы 7, где обучаем модель глубокого обучения для классификации изображений.

## 1.2. ПРОЦЕСС МАШИННОГО ОБУЧЕНИЯ

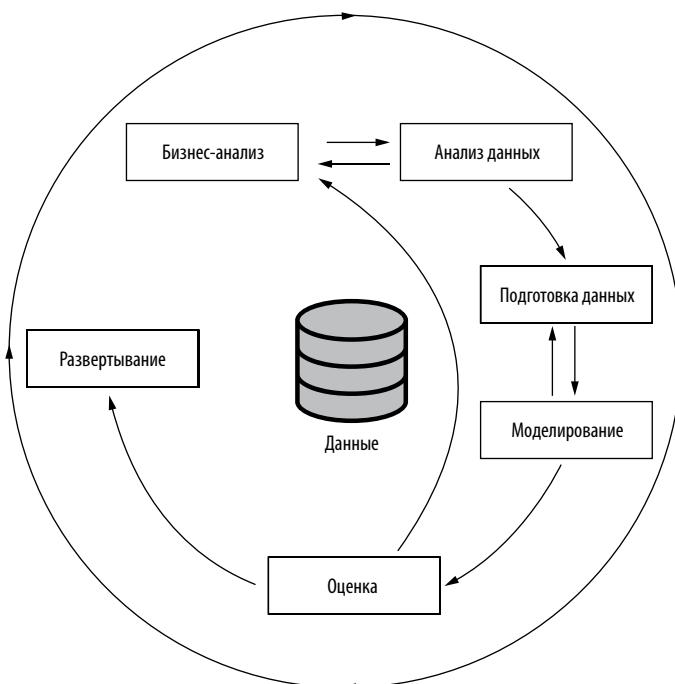
Создание системы машинного обучения включает в себя больше, чем просто выбор модели, ее обучение и применение к новым данным. Обучение модели — лишь часть, небольшой шаг в этом процессе.

Будет много других шагов, таких как определение проблемы, которую может решить машинное обучение, и использование прогнозов модели для воздействия на конечных пользователей. Более того, процесс является итеративным. Обучая модель и применяя ее к новому набору данных, мы часто выявляем случаи, в которых модель работает недостаточно хорошо. Мы используем их для переобучения модели таким образом, чтобы новая версия лучшеправлялась с подобными ситуациями.

Определенные методы и фреймворки помогают нам организовать проект машинного обучения так, чтобы он не выходил из-под контроля. Одним из таких фреймворков служит CRISP-DM, который расшифровывается как Cross-Industry Standard Process for Data Mining – *межотраслевой стандартный процесс интеллектуального анализа данных*. Он был изобретен довольно давно, в 1996 году, но, несмотря на возраст, все еще применим к сегодняшним задачам.

Согласно CRISP-DM (рис. 1.9) процесс машинного обучения состоит из шести этапов:

1. Бизнес-анализ.
2. Анализ данных.
3. Подготовка данных.
4. Моделирование.
5. Оценка.
6. Разворачивание.



**Рис. 1.9.** Процесс CRISP-DM. Проект машинного обучения начинается с понимания проблемы, а затем переходит к подготовке данных, обучению модели и оценке результатов. Наконец модель добирается до этапа развертывания. Процесс является итеративным, и на каждом шаге можно вернуться к предыдущему

Каждый этап охватывает типичные задачи:

- на этапе бизнес-анализа мы пытаемся выразить задачу, понять, как мы можем ее решить, и определить, поможет ли нам в этом машинное обучение;
- на этапе анализа данных мы анализируем доступные наборы данных и решаем, нужно ли нам собирать больше данных;
- на этапе подготовки данных мы преобразуем данные в табличную форму, которую можно использовать в качестве входных данных для модели машинного обучения;
- когда данные подготовлены, мы переходим к этапу моделирования, на котором обучаем модель;
- после определения наилучшей модели наступает этап оценки, на котором мы оцениваем модель, чтобы понять, решает ли она исходную бизнес-задачу, и оцениваем ее успешность на этом поприще;
- наконец на этапе развертывания мы развертываем модель в производственной среде.

### **1.2.1. Бизнес-анализ**

Рассмотрим пример обнаружения спама для поставщика услуг электронной почты. Мы видим больше спам-сообщений, чем когда-либо прежде, и наша нынешняя система не может с этим справиться. К данной задаче мы обращаемся на этапе бизнес-анализа: анализируем проблему и существующее решение, после чего пытаемся определить, поможет ли внедрение машинного обучения в эту систему остановить спам-сообщения. Мы также определяем цель и способы ее измерения.

Целью может быть, например, «уменьшить количество полученных спам-сообщений» или «уменьшить количество жалоб на спам, которые служба поддержки клиентов получает за день». На данном этапе мы также можем решить, что машинное обучение не поможет, и предложить более простой способ решения задачи.

### **1.2.2. Анализ данных**

Следующий шаг — анализ данных. Здесь мы попытаемся определить источники данных, которые можем использовать для решения задачи. Например, если на нашем сайте есть кнопка «Сообщить о спаме», то мы можем получить данные, сгенерированные пользователями, которые отметили свои входящие электронные письма как спам. Затем мы смотрим на данные и пытаемся понять, подходят ли они для решения нашей проблемы.

Однако эти данные могут не в полной мере подходить по целому ряду причин. Одной из них может быть то, что набор слишком мал, чтобы извлечь какие-либо полезные закономерности. Другой причиной может быть то, что данные слишком зашумлены. Пользователи могут неправильно использовать кнопку, поэтому она будет бесполезна для обучения модели машинного обучения, или же процесс сбора данных может быть нарушен, и в итоге будет собрана лишь небольшая часть нужных нам данных.

Если мы приедем к выводу, что имеющихся у нас в настоящее время данных недостаточно, то нам потребуется найти способ получить более качественные данные, независимо от того, получаем мы их из внешних источников или совершенствуем способ их сбора внутри компании. К тому же открытия, которые мы совершим на данном этапе, могут повлиять на цель, поставленную на этапе, повлияют на цель, которую мы поставили на этапе бизнес-анализа, поэтому нам, возможно, придется вернуться к этому шагу и скорректировать цель в соответствии с выводами.

Когда у нас есть надежные источники данных, мы переходим к этапу подготовки данных.

### **1.2.3. Подготовка данных**

Здесь мы очищаем данные, преобразуя их так, чтобы использовать в качестве входных для модели машинного обучения. В примере со спамом мы преобразуем набор данных в набор признаков, которые позже вводим в модель.

После того как данные подготовлены, мы переходим к этапу моделирования.

### **1.2.4. Моделирование**

На этом этапе мы решаем, какую модель машинного обучения использовать и как убедиться, что мы извлекаем из нее максимум пользы. Например, чтобы решить проблему спама, мы можем попробовать логистическую регрессию и глубокую нейронную сеть.

Нам нужно знать, как измерить производительность моделей и выбрать najlepshuyu iz nich. Чto kасается модели спам-фильтра, то мы можем посмотреть, насколько хорошо модель прогнозирует спам-сообщения, и выбрать ту, которая делает это лучше других. Для этой цели важно установить надлежащую структуру проверки, поэтому несколько позже мы рассмотрим данную задачу более подробно.

Весьма вероятно, что на текущем этапе нам придется вернуться и скорректировать способ подготовки данных. Возможно, мы выделили отличный признак, поэтому возвращаемся к этапу подготовки данных, чтобы написать некий код для вычисления этого признака. Когда код готов, мы снова обучаем модель, чтобы проверить, подходит ли признак. Например, мы могли бы добавить признак «длина темы письма», переобучить модель и проверить, улучшает ли это изменение производительность модели.

Выбрав наилучшую из возможных моделей, мы переходим к этапу оценки.

### **1.2.5. Оценка**

На данном этапе мы проверяем, соответствует ли модель ожиданиям. Ставя цель на этапе бизнес-анализа, мы также продумываем способ определения того, будет ли цель достигнута. Как правило, мы делаем это, просматривая некую важную бизнес-метрику и убеждаясь, что модель перемещает метрику в нужном направлении. В случае обнаружения спама метрикой может служить количество людей, которые нажимают кнопку «Сообщить о спаме», или количество жалоб на решаемую проблему, полученных службой поддержки. В обоих случаях мы надеемся, что использование модели сократит их количество.

В настоящее время этот шаг тесно связан со следующим — развертыванием.

### **1.2.6. Развёртывание**

Лучший способ оценить модель — устроить ей боевое крещение: протестировать ее на небольшом количестве пользователей, а затем проверить, изменилась ли наша бизнес-метрика для этих пользователей. Например, если мы хотим, чтобы наша модель уменьшила количество зарегистрированных спам-сообщений, то ожидаем увидеть меньше сообщений от этой группы по сравнению с остальными пользователями.

После развертывания модели мы используем все, что узнали на предыдущих этапах, и возвращаемся к первому шагу, чтобы поразмышлять о достигнутом (или недостигнутом). Может оказаться, что наша первоначальная цель была неправильной и что на самом деле мы хотим добиться *не* сокращения количества жалоб, а повышения вовлеченности клиентов за счет уменьшения количества спама. Поэтому мы возвращаемся к этапу бизнес-анализа и переопределяем нашу цель. Затем, при повторной оценке модели, мы уже используем другую бизнес-метрику для измерения ее качества.

### 1.2.7. Повтор

Как видите, CRISP-DM делает упор на итеративный характер процессов машинного обучения: по окончании последнего шага от нас ожидается возвращение к первому, уточнение исходной задачи и изменение ее на основе полученной информации. Мы никогда не останавливаемся на последнем шаге; вместо этого мы переосмысливаем проблему и стараемся понять, что можно улучшить на следующей итерации.

Распространенное заблуждение состоит в том, что инженеры по машинному обучению и специалисты по обработке данных целыми днями только и делают, что обучаются модели машинного обучения. В действительности это не так, что можем легко заметить на схеме CRISP-DM (см. рис. 1.9). До и после этапа моделирования выполняется множество шагов, и все они важны для успешного проекта машинного обучения.

## 1.3. МОДЕЛИРОВАНИЕ И ПРОВЕРКА МОДЕЛИ

Как мы видели ранее, обучение моделей (этап моделирования) — только один шаг во всем процессе. Но он важен, поскольку именно здесь мы на самом деле используем машинное обучение для обучения моделей.

Собрав все необходимые данные и убедившись, что они достаточно хороши, мы ищем способ обработки данных, а затем приступаем к обучению модели машинного обучения.

В нашем примере со спамом это происходит после того, как мы получим все отчеты о спаме, обработаем электронные письма и подготовим матрицу для использования в модели.

На этом этапе может встать вопрос, что использовать — логистическую регрессию или нейронную сеть. Если мы решим использовать нейронную сеть, поскольку где-то услышали, что это лучшая модель, то как нам убедиться, что она действительно лучше любой другой?

Цель данного этапа — создать модель таким образом, чтобы она обеспечивала наилучшие прогностические характеристики. Для этого нам нужен способ надежно измерить производительность каждой возможной модели-кандидата, а затем выбрать наилучшую.

Один из возможных подходов заключается в обучении модели, запуске ее в живой системе и наблюдении за тем, что произойдет. В примере со спамом мы решили использовать нейронную сеть для обнаружения спама, поэтому мы обучаем ее и внедряем в нашу производственную систему. Затем наблюдаем,

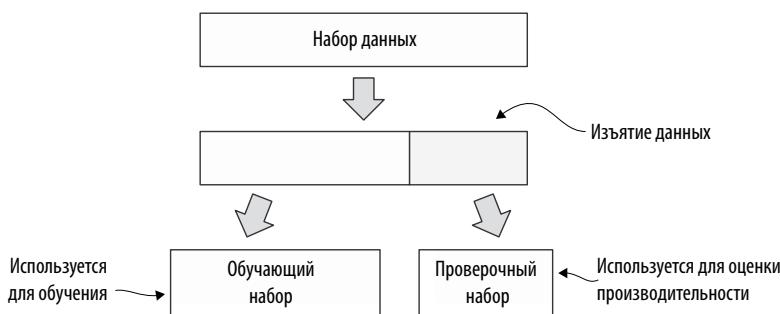
как модель ведет себя при новых сообщениях, и регистрируем случаи, когда система работает некорректно.

Однако такой подход в нашем случае не идеален: мы не можем проделать это для каждой имеющейся у нас модели-кандидата. Что еще хуже, мы можем не нарочком развернуть действительно плохую модель и увидеть, что она плохая, только после того, как она опробована на живых пользователях нашей системы.

### ПРИМЕЧАНИЕ

Тестирование модели в живой системе называется онлайн-тестированием и служит важным этапом оценки качества модели на реальных данных. Однако этот подход относится к этапам оценки и развертывания процесса, а не к этапу моделирования.

Лучший способ выбрать наилучшую модель перед развертыванием — эмуляция сценария запуска в эксплуатацию. Мы получаем наш полный набор данных, отбираем из него часть и обучаем модель на остатке. Когда обучение завершено, мы делаем вид, что сохраненный набор данных — это новые данные, и используем его для оценки производительности наших моделей. Эту часть данных часто называют *проверочным набором*, а процесс удаления части набора данных и использования его для оценки производительности называется *проверкой* (рис. 1.10).

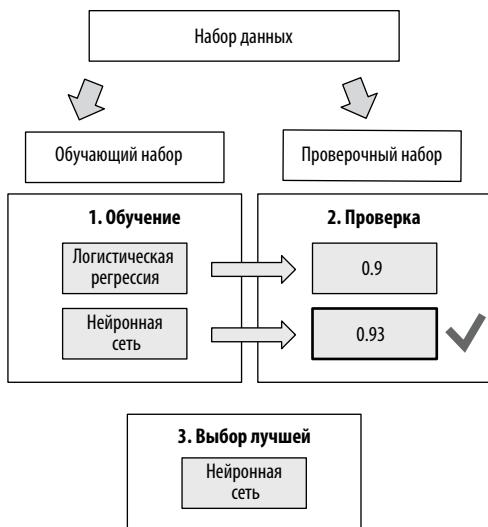


**Рис. 1.10.** Чтобы оценить производительность модели, мы отделяем часть данных и используем их только для проверки

В наборе данных о спаме мы можем изъять каждое десятое сообщение. Таким образом, мы сохраняем 10 % данных, которые используем только для проверки моделей, а остальные 90 % применяем для обучения.

Далее на основе обучающих данных мы обучаем как логистическую регрессию, так и нейронную сеть. Когда модели обучены, мы применяем их к проверочному набору данных и выясняем, какая из них более точно прогнозирует спам.

Если после применения моделей к проверочному набору мы видим, что логистическая регрессия справляется с прогнозированием спама только в 90 % случаев, тогда как нейронная сеть — в 93 % случаев, то приходим к выводу, что модель нейронной сети более результативна, чем логистическая регрессия (рис. 1.11).

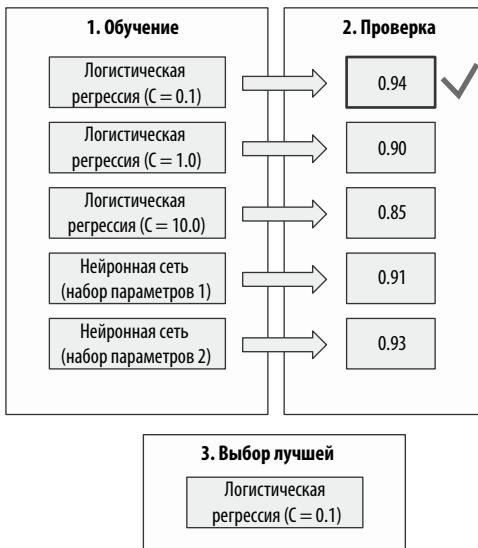


**Рис. 1.11.** Процесс проверки. Мы разделяем набор данных на две части, обучаем модели на обучающей части и оцениваем производительность на проверочной. Используя результаты оценки, мы можем выбрать наилучшую модель

Часто у нас в наличии оказываются не две модели, а гораздо больше. Логистическая регрессия, например, имеет параметр  $C$ , и в зависимости от заданного значения, результаты могут сильно различаться. Аналогично нейронная сеть имеет множество параметров, и каждый из них может очень сильно повлиять на прогностические характеристики конечной модели. Кроме того, есть и другие модели, каждая со своим набором параметров. Как нам выбрать лучшую модель с наилучшими параметрами?

Для этого мы используем ту же схему оценки. Мы обучаем модели с различными параметрами на обучающих данных, применяем их к проверочным данным, а затем выбираем модель и ее параметры на основе лучших результатов этапа проверки (рис. 1.12).

Однако у этого подхода есть один нюанс. Если мы раз за разом повторяем процесс оценки модели и используем для этой цели один и тот же проверочный набор данных, то хорошие цифры, которые мы наблюдаем в наборе данных проверки, могут оказаться лишь следствием случайности. Другими словами, «лучшей» модели могло просто повезти с прогнозированием результатов для этого конкретного набора данных.

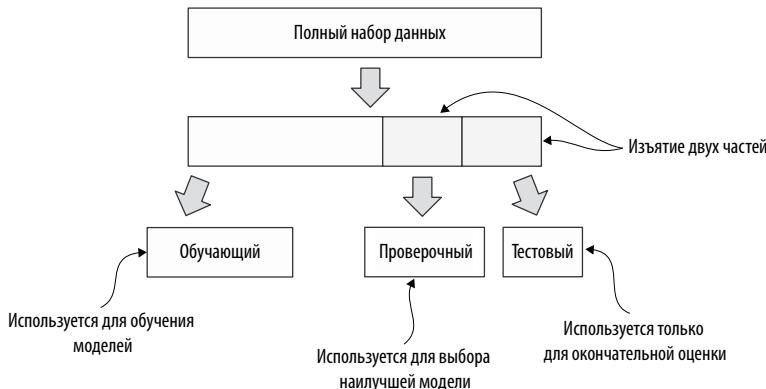


**Рис. 1.12.** Использование проверочного набора данных для выбора наилучшей модели с наилучшими параметрами

### ПРИМЕЧАНИЕ

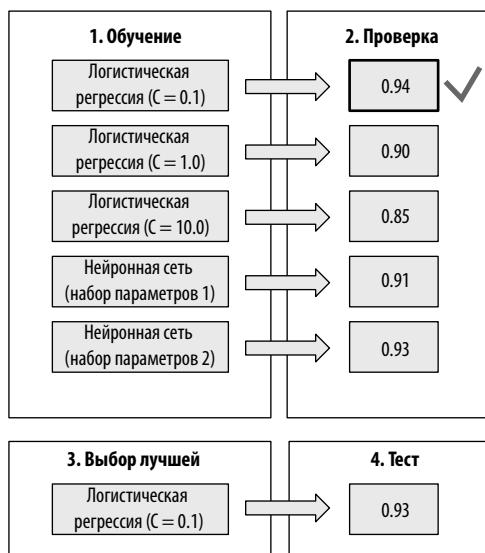
В статистике и других областях эта проблема известна как проблема множественных сравнений или проблема множественных тестов. Чем больше раз мы выполняем прогнозы на одном и том же наборе данных, тем больше вероятность того, что мы случайно увидим хорошую производительность.

Чтобы избежать этой проблемы, мы используем ту же идею: снова изынем часть данных. Эту часть назовем *тестовым* набором данных. Мы будем изредка использовать его для тестирования модели, которую выбрали как лучшую (рис. 1.13).



**Рис. 1.13.** Разделение данных на обучающую, тестовую и проверочную части

Чтобы применить этот подход к примеру со спамом, сначала мы сохраним 10 % данных в качестве тестового набора, после чего сохраним еще 10 % в качестве проверочного. Мы опробуем несколько моделей на проверочном наборе данных, выберем лучшую и применим ее уже к тестовому набору. Если при этом разница в производительности между проверкой и тестированием невелика, то можно утверждать, что эта модель действительно наилучшая (рис. 1.14).



**Рис. 1.14.** Использование тестового набора данных для подтверждения того, что производительность наилучшей модели на проверочном наборе является хорошей

### ВАЖНО

Настройка процесса проверки — наиболее важный шаг в машинном обучении. Без этого нет надежного способа выяснить, является ли только что обученная модель хорошей, бесполезной или даже вредной.

Процесс выбора наилучшей модели и наилучших параметров для модели называется *выбором модели*. Мы можем обобщить выбор модели следующим образом (рис. 1.15).

1. Мы разделяем данные на части для обучения, проверки и тестирования.
2. Сначала мы обучаем каждую модель на обучающей части, а затем оцениваем на проверочной.

3. Каждый раз, обучая новую модель, мы записываем результаты оценки, используя проверочную часть.
4. В конце мы определяем, какая модель является лучшей, и тестируем ее на тестовом наборе данных.



**Рис. 1.15.** Сначала мы разбиваем набор данных, выбираем модель и обучаем ее только на обучающей части данных. Затем, на этапе проверки, мы оцениваем модель. Мы повторяем этот процесс многократно, пока не обнаружим лучшую модель

Важно использовать процесс выбора модели, прежде всего проверив и протестировав модели в автономном режиме, чтобы убедиться, что обучаемые модели вполне пригодны. Если модель хорошо работает в автономном режиме, то мы можем принять решение о переходе к следующему шагу и развернуть ее, чтобы оценить ее производительность на реальных пользователях.

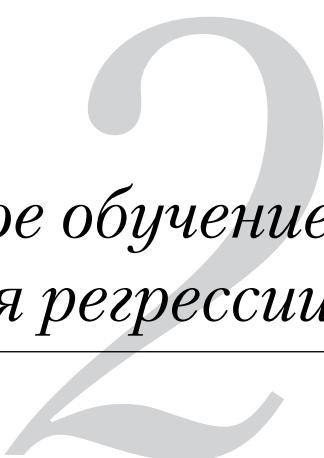
## РЕЗЮМЕ

- В отличие от традиционных систем разработки программного обеспечения на основе правил, в которых правила извлекаются и кодируются вручную, системы машинного обучения можно научить извлекать значимые закономерности из данных автоматически. Это в разы увеличивает уровень гибкости и облегчает адаптацию к изменениям.
- Для успешной реализации проекта машинного обучения требуется структура и набор руководящих принципов. CRISP-DM — платформа для организации проекта машинного обучения, которая разбивает процесс на шесть этапов, от бизнес-анализа до развертывания. Фреймворк ориенти-

## **46** Глава 1. Введение в машинное обучение

рован на итеративный характер машинного обучения и помогает нам с его организацией.

- Моделирование — важный шаг в проекте машинного обучения: та часть, где мы фактически используем машинное обучение для обучения модели. На данном этапе рождаются модели, которые обеспечивают наилучшую прогностическую производительность.
- Выбор модели представляет собой процесс выбора наилучшей модели для решения задачи. Мы разделили все доступные данные на три части: обучающую, проверочную и тестовую. Мы обучаем модели на обучающем наборе и выбираем лучшую с помощью проверочного. Когда лучшая модель выбрана, мы используем этап тестирования, чтобы выполнить окончательную проверку и убедиться, что лучшая модель работает хорошо. Этот процесс помогает создавать полезные модели, которые хорошо работают, не преподнося неприятных сюрпризов.



# *Машинное обучение для регрессии*

---

## **В этой главе**

- ✓ Создание проекта прогнозирования цен на автомобили с использованием линейной регрессионной модели.
- ✓ Выполнение первоначального исследовательского анализа данных с помощью блокнотов Jupyter.
- ✓ Настройка системы проверки.
- ✓ Реализация модели линейной регрессии с нуля.
- ✓ Выполнение простого конструирования признаков для модели.
- ✓ Контролирование модели с помощью регуляризации.
- ✓ Использование модели для прогнозирования цен на автомобили.

В главе 1 мы говорили о контролируемом машинном обучении, в котором мы обучаем модели выявлять закономерности в данных, давая им примеры. Предположим, у нас имеется набор данных с описаниями автомобилей, включающими марку, модель и возраст, и мы хотели бы с помощью машинного обучения спрогнозировать цены на эти автомобили. Эти характеристики автомобилей называются *признаками*, а цена — *целевая переменная*, которую мы хотим спрогнозировать. Затем модель получает признаки и объединяет их, чтобы вывести цену.

Это пример контролируемого обучения: у нас есть определенная информация о цене некоторых автомобилей и мы можем использовать ее для прогнозирования

цен на остальные. В главе 1 мы также говорили о различных типах контролируемого обучения: регрессии и классификации. Когда целевая переменная является числовой, перед нами задача регрессии, а когда целевая переменная является категориальной, это задача классификации.

В текущей главе мы построим регрессионную модель и начнем с самой простой — линейной регрессии. Мы сами реализуем алгоритмы, что достаточно просто сделать в нескольких строках кода. В то же время код очень нагляден, и он научит вас работать с массивами NumPy и выполнять основные матричные операции, такие как умножение и обращение. Мы также обсудим проблемы неустойчивости численного решения при обращении матрицы и узнаем, как в этом помогает регуляризация.

## 2.1. ПРОЕКТ ПО ПРОГНОЗИРОВАНИЮ ЦЕН НА АВТОМОБИЛИ

Задача этой главы состоит в прогнозировании цены на автомобиль. Предположим, что у нас имеется сайт, на котором люди могут продавать и покупать подержанные автомобили. При размещении объявления на нашем сайте продавцам часто трудно определиться с приемлемой ценой. Мы хотим помочь нашим пользователям с помощью автоматической рекомендации цены. Мы просим продавцов указать модель, марку, год выпуска, пробег и другие важные характеристики автомобиля и на основе этой информации предлагаем лучшую цену.

Один из менеджеров по продукту в компании случайно наткнулся на открытый набор данных с ценами на автомобили и попросил нас взглянуть на него. Мы проверили данные и убедились, что они содержат все важные признаки, а также рекомендуемую цену — все то, что нужно для нашего случая. Таким образом, мы решили использовать этот набор данных для построения алгоритма рекомендации цены.

План проекта состоит в следующем.

1. Сначала мы загрузим набор данных.
2. Далее проведем некий предварительный анализ данных.
3. После этого настроим стратегию проверки, позволяющую убедиться, что наша модель дает правильные прогнозы.
4. Затем реализуем модель линейной регрессии на Python и NumPy.
5. Далее рассмотрим конструирование признаков для извлечения важных функций из данных с целью улучшения модели.
6. Наконец разберемся, как сделать нашу модель стабильной с помощью регуляризации и использовать ее для прогнозирования цен на автомобили.

### 2.1.1. Загрузка набора данных

Первое, что мы сделаем для этого проекта, — это установим все необходимые библиотеки: Python, NumPy, Pandas и Jupyter Notebook. Самый простой способ — использовать дистрибутив Python под названием Anaconda (<https://www.anaconda.com>). Инструкции по установке можно найти в приложении А.

После того как библиотеки будут установлены, нам нужно скачать набор данных. У нас есть несколько вариантов того, как это сделать. Вы можете скачать его вручную через веб-интерфейс Kaggle, доступный по адресу <https://www.kaggle.com/CooperUnion/cardataset>. (Больше информации о наборе данных и о том, как он был собран, можно найти на <https://www.kaggle.com/jszhih7/car-price-prediction>.) Перейдите туда, откройте его и нажмите ссылку для загрузки. Другой вариант — использовать интерфейс командной строки Kaggle (command-line interface, CLI), который представляет собой инструмент для программного доступа ко всем наборам данных, доступным через Kaggle. В этой главе мы будем использовать второй вариант. Описание настройки интерфейса командной строки Kaggle можно найти в приложении А.

#### ПРИМЕЧАНИЕ

Kaggle — онлайн-сообщество людей, интересующихся машинным обучением. В основном оно известно тем, что проводит соревнования по машинному обучению; кроме того, это платформа для обмена данными, где любой желающий может поделиться набором данных. На ней доступны более 16 000 наборов данных. Это отличный источник проектных идей и очень полезный ресурс для проектов машинного обучения.

В этой главе, как и во всей остальной книге, мы будем активно использовать NumPy. По ходу работы мы рассмотрим все необходимые операции NumPy, но более подробную информацию можно найти в приложении В.

Исходный код этого проекта доступен в репозитории книги на GitHub по адресу <https://github.com/alexeygrigorev/mlbookcamp-code> в папке chapter-02-car-price.

В качестве первого шага мы создадим папку проекта. Мы можем дать ей любое название, например `chapter-02-car-price`:

```
mkdir chapter-02-car-price  
cd chapter-02-car-price
```

Затем скачаем сам набор данных:

```
kaggle datasets download -d CooperUnion/cardataset
```

Эта команда скачивает файл `cardataset.zip`, представляющий собой ZIP-архив. Распакуем его:

```
unzip cardataset.zip
```

Внутри всего один файл: `data.csv`.

Теперь, когда у нас есть набор данных, перейдем к следующему шагу — анализу.

## **2.2. ИССЛЕДОВАТЕЛЬСКИЙ АНАЛИЗ ДАННЫХ**

Анализ данных — важный шаг в процессе машинного обучения. Прежде чем мы сможем обучить какую-либо модель, нам нужно узнать, какие данные у нас есть и насколько они полезны. Это делается с помощью исследовательского анализа данных (exploratory data analysis, EDA).

Мы рассматриваем набор данных, чтобы изучить:

- распределение целевой переменной;
- признаки в этом наборе данных;
- распределение значений в этих признаках;
- качество данных;
- количество пропущенных значений.

### **2.2.1. Набор инструментов для исследовательского анализа данных**

Основными инструментами для такого анализа служат Jupyter Notebook, Matplotlib и Pandas:

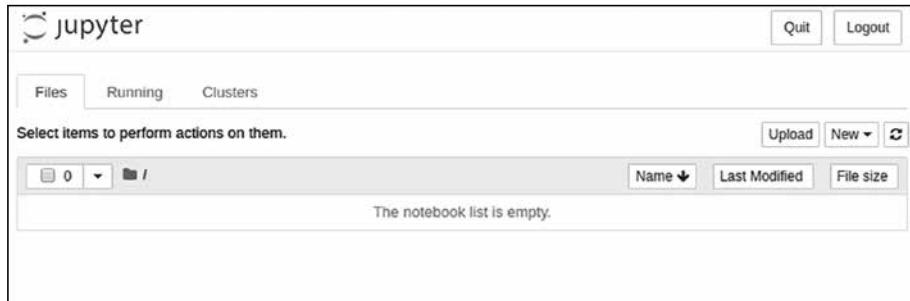
- Jupyter Notebook — инструмент для интерактивного выполнения кода на Python. Это позволяет исполнить фрагмент кода и тут же увидеть результат. Кроме того, мы можем отображать диаграммы и свободно добавлять заметки с комментариями. Он также поддерживает и другие языки, такие как R или Julia (отсюда и название: Jupyter расшифровывается как Julia, Python, R), но мы будем использовать его только для Python;
- Matplotlib — библиотека для построения графиков. Это эффективный инструмент, позволяющий создавать различные типы визуализаций, такие как линейные диаграммы, столбчатые диаграммы и гистограммы;
- Pandas — библиотека для работы с табличными данными. Она позволяет считывать данные из любого источника, будь то файл CSV, файл JSON или база данных.

Мы также будем использовать Seaborn, еще один инструмент для построения графиков, являющийся надстройкой Matplotlib и упрощающий рисование диаграмм.

Запустим Jupyter Notebook, выполнив следующую команду:

```
jupyter notebook
```

Эта команда запускает сервер Jupyter Notebook в текущем каталоге и открывает его в браузере по умолчанию (рис. 2.1).



**Рис. 2.1.** Начальный экран сервиса Jupyter Notebook

Если Jupyter запущен на удаленном сервере, то ему потребуется дополнительная настройка. Подробную информацию о настройке можно найти в приложении А.

Теперь создадим блокнот для этого проекта. Нажмите **New**, затем выберите Python 3 в разделе **Notebooks**. Мы можем назвать ее **chapter-02-car-price-project** — щелкните на текущем названии (**Untitled**) и замените его новым.

В первую очередь нам нужно импортировать все библиотеки, необходимые для этого проекта. Напишите в первой ячейке следующее:

```
Импортирует NumPy — библиотека  
для числовых операций ①
import numpy as np ←
import pandas as pd ← ② Импортирует Pandas —  
библиотеку для табличных данных

from matplotlib import pyplot as plt | ③ Импортирует библиотеки построения  
import seaborn as sns ←
%matplotlib inline ← ④ Обеспечивает правильную визуализацию  
графиков в блокнотах Jupyter
```

Первые две строки, ① и ②, отвечают за импорт необходимых библиотек: NumPy для числовых операций и Pandas для табличных данных. Существует соглашение о том, чтобы импортировать эти библиотеки, используя более короткие псевдонимы (например, `pd` в `import pandas as pd`). Это соглашение распространено в сообществе машинного обучения Python, и ему следуют буквально все.

Следующие две строки, ③, отвечают за импорт графических библиотек. Первая из них, Matplotlib, — библиотека для создания высококачественных визуализаций. Это не самая легкая для использования библиотека. Некоторые библиотеки упрощают использование Matplotlib, и Seaborn — как раз одна из них.

Наконец `%matplotlib inline` в строке ❸ сообщает Jupyter, что в блокноте ожидаются графики и необходимо отображать их, как только они нам понадобятся.

Нажмите `Shift+Enter` или кнопку Run, чтобы исполнить содержимое выбранной ячейки.

Мы не будем углубляться в подробности работы с Jupyter Notebook. Чтобы узнать об этом больше, посетите официальный сайт (<https://jupyter.org>). Там можно найти большой объем документации и примеров, которые помогут вам освоить данный инструмент.

### 2.2.2. Считывание и подготовка данных

Теперь ознакомимся с нашим набором данных. Для этой цели мы можем использовать функцию `read_csv` из Pandas. Введите следующий код в следующую ячейку и снова нажмите `Shift+Enter`:

```
df = pd.read_csv('data.csv')
```

Эта строка кода считывает CSV-файл и записывает результаты в переменную с именем `df`, что является сокращением от *DataFrame* (датафрейм). Теперь мы можем узнать, сколько там строк. Воспользуемся функцией `len`:

```
len(df)
```

Функция выводит `11914`, и это означает, что в данном наборе данных насчитывается почти 12 000 автомобилей (рис. 2.2).

```
jupyter car-price-project
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

In [1]: import pandas as pd
       import numpy as np

       import seaborn as sns
       from matplotlib import pyplot as plt
       %matplotlib inline

In [2]: df = pd.read_csv('data.csv')

In [3]: len(df)
Out[3]: 11914

In [ ]:
```

**Рис. 2.2.** Блокноты Jupyter являются интерактивными. Мы можем ввести в ячейку код, выполнить его и сразу же увидеть результат, что идеально подходит для исследовательского анализа данных

Теперь воспользуемся функцией `df.head()`, чтобы взглянуть на первые пять строк нашего датафрейма `DataFrame` (рис. 2.3).

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors
0	BMW	Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0 T
1	BMW	Series 1	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0 Lux
2	BMW	Series 1	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0
3	BMW	Series 1	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0 Lu
4	BMW	Series 1	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0

**Рис. 2.3.** Вывод функции `head()` из датафрейма Pandas показывает первые пять строк набора данных. Этот вывод позволяет нам понять, как выглядят данные

Это дает нам представление о том, как выглядят данные. Мы уже видим, что в наборе данных содержатся кое-какие несоответствия: имена столбцов иногда содержат пробелы, а иногда подчеркивания (`_`). То же самое верно и для значений признаков: иногда они пишутся с заглавной буквы, а иногда представляют собой короткие строки с пробелами. Это неудобно и может запутать, но мы можем решить проблему с помощью нормализации, заменив все пробелы символами подчеркивания и переведя все буквы в нижний регистр:

```
Переводит все имена столбцов в нижний регистр ①  
и заменяет пробелы символами подчеркивания  
  
df.columns = df.columns.str.lower().str.replace(' ', '_') ← ②  
  
string_columns = list(df.dtypes[df.dtypes == 'object'].index) ← ③  
Выбирает  
только столбцы  
со строковыми  
значениями  
  
for col in string_columns:  
    df[col] = df[col].str.lower().str.replace(' ', '_') ← ④  
  
Понижает регистр и заменяет пробелы символами  
подчеркивания для значений во всех строковых  
столбцах датафрейма
```

В ① и ③ мы используем специальный атрибут `str`. Он позволяет нам применять строковые операции ко всему столбцу одновременно, не прибегая к написанию циклов `for`. Мы используем его для понижения регистра имен столбцов и содержимого этих столбцов, а также для замены пробелов подчеркиванием.

## 54 Глава 2. Машинное обучение для регрессии

Мы можем использовать данный атрибут только для столбцов со строковыми значениями внутри. Именно поэтому мы сначала выбираем такие столбцы в ❷.

### ПРИМЕЧАНИЕ

В этой и последующих главах мы рассматриваем соответствующие операции Pandas в рабочем порядке, но на довольно высоком уровне. В приложении Г можно найти материалы, которые позволяют более последовательно и углубленно ознакомиться с Pandas.

После этой начальной предварительной обработки датафрейм выглядит уже более однородным (рис. 2.4).

In [6]:	df.head()								
Out[6]:	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	n
0	bmw	1_series_m	2011	premium_unleaded_(required)	335.0	6.0	manual	rear_wheel_drive	
1	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	rear_wheel_drive	
2	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	rear_wheel_drive	
3	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	rear_wheel_drive	
4	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	rear_wheel_drive	

**Рис. 2.4.** Результат предварительной обработки данных. Имена и значения столбцов нормализованы: они записаны в нижнем регистре, а пробелы преобразованы в символы подчеркивания

Как мы видим, этот набор данных содержит несколько столбцов:

- `make` — марка автомобиля (BMW, Toyota и т. д.);
- `model` — модель автомобиля;
- `year` — год выпуска автомобиля;
- `engine_fuel_type` — тип топлива, необходимого двигателю (дизельный, электрический и т. д.);
- `engine_hp` — мощность двигателя в лошадиных силах;
- `engine_cylinders` — количество цилиндров в двигателе;
- `transmission_type` — тип коробки передач (автоматическая или ручная);
- `driven_wheels` — привод (передний, задний, полный);
- `number_of_doors` — количество дверей в автомобиле;
- `market_category` — премиальный, кроссовер и т. д.;
- `vehicle_size` — компактный, средний или большой;
- `vehicle_style` — седан или кабриолет;
- `highway_mpg` — миль на галлон (miles per gallon, mpg) на шоссе;
- `city_mpg` — миль на галлон по городу;

- *popularity* — количество упоминаний автомобиля в Twitter;
- *msrp* — рекомендованная производителем розничная цена.

Для нас наиболее интересным здесь является последний столбец: MSRP (рекомендованная производителем розничная цена или просто цена). Мы используем ее для прогнозирования цен на автомобиль.

### 2.2.3. Анализ целевых переменных

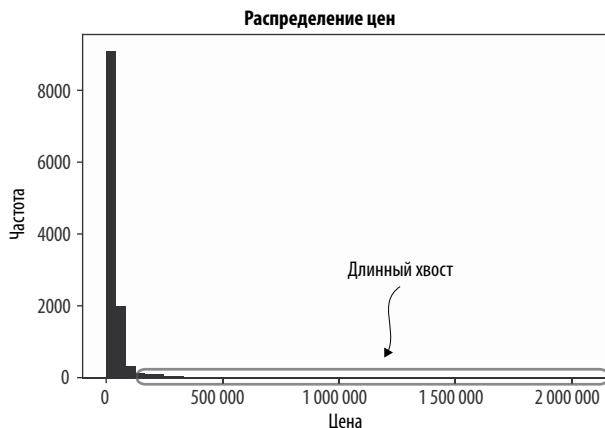
Столбец MSRP содержит важную информацию — это наша целевая переменная  $y$ , которая и является значением, которое мы будем учиться прогнозировать.

Одним из первых шагов исследовательского анализа данных всегда должно быть выяснение того, что представляют собой значения  $y$ . Обычно мы делаем это, проверяя распределение  $y$ : визуальное описание того, какими могут быть возможные значения  $y$  и как часто они встречаются. Этот тип визуализации называется *гистограммой*.

Для построения гистограммы мы используем Seaborn, поэтому введите в блокнот Jupyter следующее:

```
sns.histplot(df.msrp, bins=40)
```

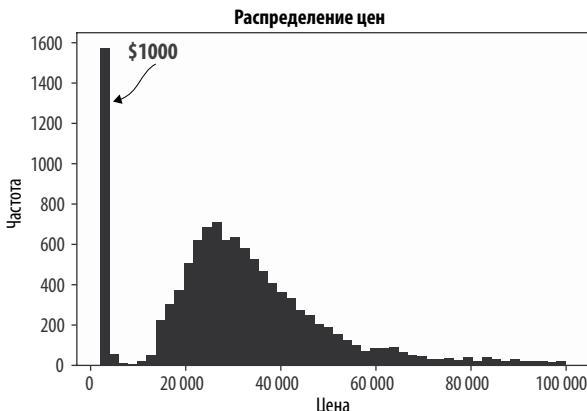
Построив этот график, мы сразу же отмечаем, что распределение цен имеет очень длинный «хвост». В левой части много автомобилей с низкими ценами, но их количество быстро уменьшается, после чего остается длинный «хвост» из очень немногих автомобилей с высокими ценами (рис. 2.5).



**Рис. 2.5.** Распределение цен в наборе данных. Мы видим множество значений на нижнем конце ценовой оси и почти ничего на верхнем. Это распределение с длинным «хвостом», что является типичной ситуацией, когда есть множество товаров с низкими ценами и очень небольшое количество дорогих

Мы можем рассмотреть подробности, немного увеличив масштаб и отобрав значения ниже 100 000 долларов (рис. 2.6):

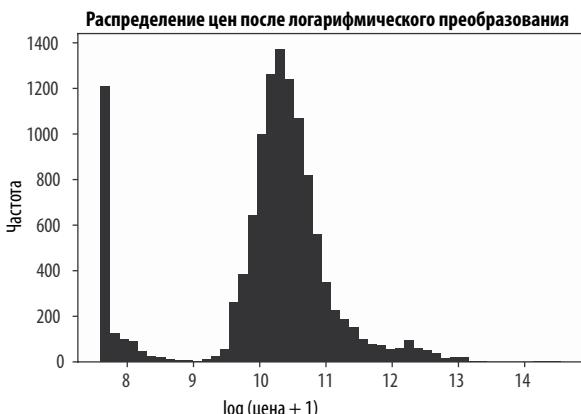
```
sns.histplot(df.msrp[df.msrp < 100000])
```



**Рис. 2.6.** Распределение цен на автомобили стоимостью менее 100 000 долларов США. Рассмотрение только цен ниже 100 000 долларов позволяет нам лучше увидеть начало распределения. Мы также видим много автомобилей стоимостью 1000 долларов

Из-за длинного «хвоста» нам довольно сложно увидеть распределение, но еще большее влияние он оказывает на модель: такое распределение может сильно ее запутать, что приведет к недостаточно хорошему обучению. Одним из способов решения этой проблемы служит логарифмическое преобразование. Применив функцию  $\log$  к ценам, мы устраним этот нежелательный эффект (рис. 2.7).

$$y_{\text{new}} = \log(y + 1)$$



**Рис. 2.7.** Логарифм цены. Эффект длинного «хвоста» устранен, и мы можем наблюдать все распределение на едином графике

Часть + 1 важна в случаях, когда в данных имеются нули. Логарифм нуля равен минус бесконечности, но логарифм единицы равен нулю. Если все наши значения неотрицательны, то, добавив 1, мы можем быть уверены, что преобразованные значения не опускаются ниже нуля.

В нашем конкретном случае нулевые значения не являются проблемой — все цены, которые у нас есть, начинаются с 1000 долларов — но это то соглашение, которому мы все равно следуем. В NumPy имеется функция, которая выполняет такое преобразование:

```
log_price = np.log1p(df.msrp)
```

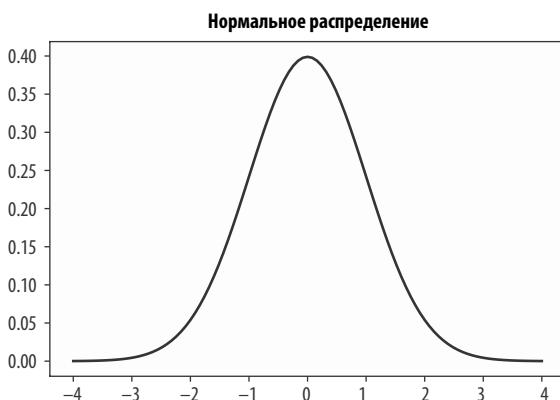
Чтобы взглянуть на распределение цен после преобразования, мы можем использовать ту же функцию `histplot` (см. рис. 2.7):

```
sns.histplot(log_price)
```

Как видите, данное преобразование удаляет длинный «хвост», и теперь распределение напоминает колокол. Конечно, такое распределение не будет нормальным из-за большого пика более низких цен, но теперь модель вполне может справиться с задачей.

### ПРИМЕЧАНИЕ

Как правило, хорошо, если целевое распределение выглядит как нормальное (рис. 2.8). При этом условии такие модели, как линейная регрессия, работают приемлемо.



**Рис. 2.8.** Нормальное распределение, также известное как гауссово, следует колоколообразной кривой, которая симметрична и имеет пик в центре

**Упражнение 2.1**

Голова распределения — диапазон, в котором содержится много значений. Что такое длинный «хвост» распределения?

- А. Большой пик в районе 1000 долларов США.
- Б. Случай, когда множество значений распределены очень далеко от головы, и эти значения визуально отображаются в виде «хвоста» на гистограмме.
- В. Множество очень схожих значений собраны вместе в пределах короткого диапазона.

## 2.2.4. Проверка на наличие пропущенных значений

Более внимательно мы рассмотрим другие признаки чуть позже, но есть что-то, что мы должны сделать сейчас, — проверить данные на наличие пропущенных значений. Этот шаг важен, поскольку модели машинного обучения, как правило, не умеют автоматически обрабатывать пропущенные значения. Нам нужно выяснить, не потребуется ли нам делать что-то особенное для обработки этих значений.

Pandas имеет удобную функцию, которая проверяет наличие пропущенных значений:

```
df.isnull().sum()
```

Эта функция показывает

make	0
model	0
year	0
engine_fuel_type	3
engine_hp	69
engine_cylinders	30
transmission_type	0
driven_wheels	0
number_of_doors	6
market_category	3742
vehicle_size	0
vehicle_style	0
highway_mpg	0
city_mpg	0
popularity	0
msrp	0

Во-первых, мы выясняем, что MSRP (наша целевая переменная) не содержит никаких пропущенных значений. Это хороший результат, поскольку в противном случае такие записи для нас бесполезны: нам всегда нужно знать целевое значение наблюдения, чтобы использовать его для обучения. Кроме того, в ряде столбцов отсутствуют значения, особенно в `market_category`, где имеется почти 4000 строк с отсутствующими значениями.

Нам придется разобраться с недостающими значениями позже, когда мы будем обучать модель, поэтому мы должны помнить об этой проблеме. На данный момент больше ничего не делаем с этими признаками и переходим к следующему шагу: настройке платформы проверки, позволяющей обучать и тестировать модели машинного обучения.

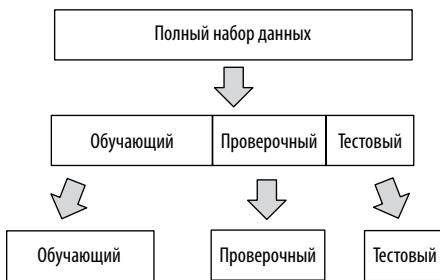
### 2.2.5. Платформа проверки

Мы уже выяснили, что важно настроить систему проверки как можно раньше, чтобы удостовериться в том, что обучаемые модели не только хорошо работают, но и могут быть обобщены, то есть применяться к новым, неизвестным данным. Чтобы это проделать, мы отберем часть данных и будем обучать модель только на оставшейся части. Затем мы используем отложенный набор данных — тот, который мы не использовали для обучения, — чтобы убедиться, что предсказания модели имеют смысл.

Этот шаг важен, поскольку мы обучаем модель, используя методы оптимизации, которые подгоняют функцию  $g(X)$  к данным  $X$ . Иногда эти методы обнаруживают ложные закономерности, которые кажутся модели реальными, но на самом деле представляют собой случайные флуктуации. Если у нас есть небольшой обучающий набор данных, в котором, например, все автомобили BMW стоят всего 10 000 долларов, то модель посчитает, что это верно для всех автомобилей BMW в мире.

Чтобы подобного не произошло, мы используем проверку. Поскольку набор проверочных данных не используется в обучении, для метода оптимизации эти данные будут новыми. Применив модель к этим данным, мы имитируем случай применения модели к новым данным, которые не встречались ранее. Если в проверочном наборе данных содержатся автомобили BMW с ценами выше 10 000 долларов, но наша модель будет прогнозировать их стоимость как 10 000 долларов, то мы поймем, что модель плохо работает на этих примерах.

Как мы уже знаем, нам следует разделить набор данных на три части: обучающую, проверочную и тестовую (рис. 2.9).



**Рис. 2.9.** Весь набор данных разделен на три части: обучающую, проверочную и тестовую

Разделим датафрейм таким образом, чтобы:

- 20 % данных шло на проверку;
- 20 % — на тестирование;
- остальные 60 % — на обучение (листинг 2.1).

#### Листинг 2.1. Разделение данных на наборы для проверки, тестирования и обучения

```

n = len(df)           ❶ Возвращает количество
                      строк в датафрейме

n_val = int(0.2 * n)
n_test = int(0.2 * n)
n_train = n - (n_val + n_test) ❷ Вычисляет, сколько строк должно быть
                                использовано для обучения, проверки
                                и тестирования

np.random.seed(2)      ❸ Фиксирует случайное начальное значение,
                      обеспечивая воспроизводимость результатов

idx = np.arange(n)    ❹ Создает массив NumPy с индексами от 0 до (n-1)
np.random.shuffle(idx) | и перетасовывает его

df_shuffled = df.iloc[idx] ❺ Использует массив с индексами для получения
                           перетасованного датафрейма

df_train = df_shuffled.iloc[:n_train].copy()
df_val = df_shuffled.iloc[n_train:n_train+n_val].copy()
df_test = df_shuffled.iloc[n_train+n_val:].copy() ❻ Разбивает перетасованный
                                                датафрейм на обучающие,
                                                проверочные и тестовые
  
```

Рассмотрим этот код более внимательно и проясним кое-какие нюансы.

В ❹ мы создаем массив, а затем перетасовываем его. Посмотрим, что здесь происходит. Мы можем взять меньший массив из пяти элементов и перетасовать его:

```

idx = np.arange(5)
print('before shuffle', idx)
np.random.shuffle(idx)
print('after shuffle', idx)
  
```

Если мы это запустим, то на выходе получим что-то вроде

```
before shuffle [0 1 2 3 4]
after shuffle [2 3 0 4 1]
```

Однако если мы запустим код снова, то результаты будут другими:

```
before shuffle [0 1 2 3 4]
after shuffle [4 3 0 2 1]
```

Чтобы обеспечить одинаковые результаты при каждом запуске, в ❸ мы фиксируем случайное начальное значение:

```
np.random.seed(2)
idx = np.arange(5)
print('before shuffle', idx)
np.random.shuffle(idx)
print('after shuffle', idx)
```

Функция `np.random.seed` принимает любое число и использует его в качестве начального числа для всех сгенерированных данных внутри библиотеки случайных чисел NumPy.

Выполнив этот код, мы получим

```
before shuffle [0 1 2 3 4]
after shuffle [2 4 1 3 0]
```

В этом случае результаты по-прежнему являются случайными, но при повторном запуске результат оказывается таким же, как и при предыдущем:

```
before shuffle [0 1 2 3 4]
after shuffle [2 4 1 3 0]
```

Это хорошо для воспроизводимости. Если требуется, чтобы кто-то другой запустил данный код и получил те же результаты, то нам нужно убедиться, что зафиксировано все, даже «случайный» компонент нашего кода.

## ПРИМЕЧАНИЕ

Это делает результаты воспроизводимыми на том же компьютере. При использовании другой операционной системы и другой версии NumPy результат может оказаться другим.

Создав массив с индексами `idx`, мы можем использовать его для получения перетасованной версии нашего исходного датафрейма. Для этой цели в ❹ мы используем функцию `iloc`, дающую доступ к строкам датафрейма по их номерам:

```
df_shuffled = df.iloc[idx]
```

## 62 Глава 2. Машинное обучение для регрессии

Если `idx` содержит перетасованные последовательные числа, то этот код создаст перетасованный датафрейм (рис. 2.10).

	make	model	year	msrp
0	lotus	evora_400	2017	91900
1	aston_martin	v8_vantage	2014	136900
2	hyundai	genesis	2015	38000
3	suzuki	samurai	1993	2000
4	mitsubishi	outlander	2015	26195

`df.iloc[idx]`

`idx = [2, 4, 1, 3, 0]`

	make	model	year	msrp
2	hyundai	genesis	2015	38000
4	mitsubishi	outlander	2015	26195
1	aston_martin	v8_vantage	2014	136900
3	suzuki	samurai	1993	2000
0	lotus	evora_400	2017	91900

**Рис. 2.10.** Использование функции `iloc` для перетасовки датафрейма.

При использовании с перетасованным массивом индексов функция создает перетасованный датафрейм

В примере мы использовали `iloc` со списком индексов. Оператор двоеточия (`:`) также позволяет нам использовать диапазоны. Именно это мы делаем в ❶, чтобы разделить перетасованный датафрейм на обучающий, проверочный и тестовый наборы:

```
df_train = df_shuffled.iloc[:n_train].copy()
df_val = df_shuffled.iloc[n_train:n_train+n_val].copy()
df_test = df_shuffled.iloc[n_train+n_val: ].copy()
```

Теперь датафрейм разделен на три части и мы можем двигаться дальше. Наш первоначальный анализ показал длинный «хвост» в распределении цен, и для устранения данного эффекта нам следует применить логарифмическое преобразование. Мы можем проделать это отдельно для каждого датафрейма:

```
y_train = np.log1p(df_train.msrp.values)
y_val = np.log1p(df_val.msrp.values)
y_test = np.log1p(df_test.msrp.values)
```

Чтобы избежать случайного использования целевой переменной в дальнейшем, удалим ее из датафреймов:

```
del df_train['msrp']
del df_val['msrp']
del df_test['msrp']
```

### ПРИМЕЧАНИЕ

Удаление целевой переменной — необязательный шаг. Но стоит убедиться, что мы не задействуем ее при обучении модели: если подобное произойдет, то мы будем использовать цену для прогнозирования цены и наша модель будет показывать идеальную достоверность.

Когда проверочное разделение завершено, мы можем переходить к следующему шагу — обучению.

## 2.3. МАШИННОЕ ОБУЧЕНИЕ ДЛЯ РЕГРЕССИИ

Выполнив первоначальный анализ данных, мы готовы обучать модель. Проблема, которую мы решаем, — это задача регрессии: цель состоит в том, чтобы предсказать число, а именно цену автомобиля. Для этого проекта мы будем использовать простейшую регрессионную модель: линейную регрессию.

### 2.3.1. Линейная регрессия

Чтобы спрогнозировать цену автомобиля, нам нужно использовать какую-то модель машинного обучения. В нашем случае мы будем использовать линейную регрессию, которую реализуем самостоятельно. Как правило, вручную это не делается, и подобную работу выполняет какая-либо библиотека. Однако в этой главе мы хотим показать, что внутри таких фреймворков нет ничего волшебного: это просто код. Линейная регрессия — идеальная модель, поскольку она относительно проста и может быть реализована с помощью всего нескольких строк кода NumPy.

Для начала разберемся, как работает линейная регрессия. Как мы знаем из главы 1, модель контролируемого машинного обучения имеет определенную форму:

$$y \approx g(X).$$

Это матричная форма.  $X$  — матрица, где признаки наблюдений являются строками матрицы, а  $y$  — вектор со значениями, которые требуется спрогнозировать.

Эти матрицы и векторы могут запутывать, так что сделаем шаг назад и рассмотрим, что происходит с одним наблюдением  $x_i$  и значением  $y_i$ , которое мы хотим спрогнозировать. Индекс  $i$  здесь означает, что это номер наблюдения  $i$ , одно из  $m$  наблюдений, которые содержатся в нашем обучающем наборе данных.

Тогда для этого единственного наблюдения предыдущая формула выглядит следующим образом:

$$y_i \approx g(x_i).$$

Если у нас есть  $n$  признаков, то наш вектор  $x_i$  является  $n$ -мерным, поэтому он содержит  $n$  компонентов:

$$x_i = (x_{i1}, x_{i2} \dots x_{in}).$$

Поскольку он имеет  $n$  компонентов, мы можем записать функцию  $g$  как функцию с  $n$  параметрами, что совпадает с предыдущей формулой:

$$y_i = g(x_i) = g(x_{i1}, x_{i2} \dots x_{in}).$$

## 64 Глава 2. Машинное обучение для регрессии

В нашем случае у нас в обучающем наборе данных 7150 автомобилей. Это означает, что  $m = 7150$ , а  $i$  может быть любым числом от 0 до 7149. Для  $i = 10$ , например, мы получим такой автомобиль:

make	rolls-royce
model	phantom_drophead_coupe
year	2015
engine_fuel_type	premium_unleaded_(required)
engine_hp	453
engine_cylinders	12
transmission_type	automatic
driven_wheels	rear_wheel_drive
number_of_doors	2
market_category	exotic, luxury, performance
vehicle_size	large
vehicle_style	convertible
highway_mpg	19
city_mpg	11
popularity	86
msrp	479775

Выберем несколько числовых признаков и пока проигнорируем остальные. Мы можем начать с лошадиных сил, миль на галлон по городу и популярности:

engine_hp	453
city_mpg	11
popularity	86

Затем присвоим эти признаки  $x_{i1}$ ,  $x_{i2}$  и  $x_{i3}$  соответственно. Таким образом, мы получаем вектор признаков  $x_i$  с тремя компонентами:

$$x_i = (x_{i1}, x_{i2}, \dots, x_{in}) = (453, 11, 86).$$

Чтобы лучше все понять описываемое, мы можем перевести эту математическую нотацию на Python. В нашем случае функция  $g$  имеет следующую сигнатуру:

```
def g(xi):
    # xi – это список из n элементов
    # делаем что-нибудь с xi
    # вернуть результат
    pass
```

В этом коде переменная  $xi$  — это наш вектор  $x_i$ . В зависимости от реализации  $xi$  может быть списком с  $n$  элементами или массивом NumPy, имеющим размер  $n$ .

Для автомобиля, описанного ранее,  $xi$  представляет собой список из трех элементов:

```
xi = [453, 11, 86]
```

Когда мы применяем функцию  $g$  к вектору  $xi$ , она выдает  $y_{pred}$  в качестве вывода, который является прогнозом  $g$  для  $xi$ :

```
y_pred = g(xi)
```

Мы ожидаем, что прогноз окажется как можно ближе к  $y_i$ , реальной цене автомобиля.

### ПРИМЕЧАНИЕ

Чтобы проиллюстрировать идеи, лежащие в основе математических формул, в этом разделе мы будем использовать Python. Нам не требуется использовать эти фрагменты кода для самого проекта. С другой стороны, запуск данного кода в Jupyter может помочь понять концепции.

Функция  $g$  может выглядеть по-разному, и выбор алгоритма машинного обучения определяет способ ее работы.

Если  $g$  — модель линейной регрессии, то она получит следующий вид:

$$g(x_i) = g(x_{i1}, x_{i2} \dots x_{in}) = w_0 + x_{i1}w_1 + x_{i2}w_2 + \dots + x_{in}w_n.$$

Переменные  $w_0, w_1, w_2 \dots w_n$  служат параметрами модели:

- $w_0$  — составляющая *смещения*;
- $w_1, w_2 \dots w_n$  — *веса* каждого признака  $x_{i1}, x_{i2} \dots x_{in}$ .

Эти параметры точно определяют, как модель должна комбинировать признаки, чтобы прогнозы в итоге получились максимально хорошими. Ничего страшного, если значение этих параметров пока не до конца ясно, поскольку мы рассмотрим их несколько позже.

Чтобы формула была короче, используем обозначение суммы:

$$g(x_i) = g(x_{i1}, x_{i2} \dots x_{in}) = w_0 + \sum_{j=1}^n x_{ij}w_j.$$

### Упражнение 2.2

Для контролируемого обучения мы используем модель машинного обучения для единственного наблюдения  $y_i \approx g(x_i)$ . Что такое  $x_i$  и  $y_i$  в этом проекте?

- А.  $x_i$  — вектор признаков, содержащий ряд чисел, описывающих объект (автомобиль), а  $y_i$  — логарифм цены этого автомобиля.
- Б.  $y_i$  — вектор признаков, содержащий ряд чисел, описывающих объект (автомобиль), а  $x_i$  — логарифм цены этого автомобиля.

## 66 Глава 2. Машинное обучение для регрессии

Указанные веса — это то, что модель усваивает, когда мы ее обучаем. Чтобы лучше понять, как модель использует веса, рассмотрим следующие значения (табл. 2.1).

**Таблица 2.1.** Пример весов, полученных с помощью модели линейной регрессии

$w_0$	$w_1$	$w_2$	$w_3$
7,17	0,01	0,04	0,002

Итак, если мы захотим перевести эту модель на Python, то она будет выглядеть следующим образом:

```
w0 = 7.17
# [w1    w2    w3    ]
w = [0.01, 0.04, 0.002]
n = 3

def linear_regression(xi):
    result = w0
    for j in range(n):
        result = result + xi[j] * w[j]
    return result
```

Мы помещаем все веса объектов в один список  $w$  — точно так же, как мы ранее поступили с  $xi$ . Все, что нам теперь нужно сделать, — перебрать эти веса и умножить их на соответствующие значения признаков. Это и будет не чем иным, как прямым переводом предыдущей формулы на Python.

Разобраться в этом достаточно легко. Взгляните еще раз на формулу:

$$w_0 + \sum_{j=1}^n x_{ij} w_j.$$

В нашем примере имеются три признака, поэтому  $n = 3$ , и мы получаем

$$g(x_i) = g(x_{i1}, x_{i2}, x_{i3}) = w_0 + \sum_{j=1}^3 x_{ij} w_j = w_0 + x_{i1} w_1 + x_{i2} w_2 + x_{i3} w_3.$$

Это именно то, что мы видим в коде

```
result = w0 + xi[0] * w[0] + xi[1] * w[1] + xi[2] * w[2]
```

за простым исключением, что индексация в Python начинается с 0,  $x_{i1}$  превращается в  $xi[0]$ , а  $w_1$  — в  $w[0]$ .

Теперь посмотрим, что произойдет, когда мы применим модель к нашему наблюдению  $x_i$  и заменим веса их значениями:

$$g(x_i) = 7,17 + 453 \times 0,01 + 11 \times 0,04 + 86 \times 0,002 = 12,31.$$

Прогноз, который мы получим для этого наблюдения, будет равен 12,31. Вспомните, что во время предварительной обработки мы применили к нашей целевой переменной  $y$  логарифмическое преобразование. Вот почему модель, которую мы обучили на этих данных, также предсказывает логарифм цены. Чтобы отменить преобразование, нам нужно взять экспоненту логарифма. В нашем случае прогноз становится равным 603 000 долларов:

$$\exp(12,31 + 1) = 603\,000.$$

Смещение (7,17) — это значение, которое мы бы получили, если бы ничего не знали об автомобиле; оно служит базовой линией.

Однако мы кое-что знаем об этом автомобиле: мощность, миль на галлон по городу (MPG) и популярность. Это признаки  $x_{i1}$ ,  $x_{i2}$  и  $x_{i3}$ , каждый из которых что-то говорит нам об автомобиле. Мы используем данную информацию для корректировки базовой линии.

Рассмотрим первый признак: лошадиные силы. Вес для этого признака равен 0,01, это значит, для каждой дополнительной единицы лошадиной силы мы корректируем базовую линию, добавляя 0,01. Поскольку у нас в двигателе 453 лошади, мы добавляем 4,53 к базовому показателю:  $453 \text{ л/с} \cdot 0,01 = 4,53$ .

То же самое происходит и с MPG. Каждая дополнительная миля на галлон увеличивает цену на 0,04, поэтому мы добавляем 0,44:  $11 \text{ м/г} \cdot 0,04 = 0,44$ .

Наконец мы принимаем во внимание популярность. В нашем примере каждое упоминание в ленте Twitter приводит к увеличению на 0,002. В общей сложности популярность вносит 0,172 в окончательный прогноз.

Именно поэтому мы получаем 12,31, когда сводим все воедино (рис. 2.11).

$$g(x_i) = 7.17 + 453 \cdot 0.01 + 11 \cdot 0.04 + 86 \cdot 0.002 = 12.31$$

Смещение	Лошадиные силы	MPG	Популярность
4.53	0.44	0.172	

**Рис. 2.11.** Прогноз линейной регрессии представляет собой базовую линию 7,17 (смещение), скорректированную с учетом информации, полученной от признаков. Доля лошадиных сил в окончательном прогнозе составляет 4,53, миль на галлон — 0,44, а популярности — 0,172

Теперь вспомним, что на самом деле мы имеем дело с векторами, а не с отдельными числами. Мы знаем, что  $x_i$  — это вектор с  $n$  компонентами:

$$x_i = (x_{i1}, x_{i2}, \dots, x_{in}).$$

Мы также можем объединить все веса в один вектор  $w$ :

$$w = (w_0, w_1, w_2, \dots, w_n).$$

Фактически мы уже делали это в примере Python, когда помещали все веса в список, который представлял собой вектор размерности 3 с весами для каждого отдельного признака. Вот как выглядят векторы в нашем примере:

$$\begin{aligned}x_i &= (x_{i1}, x_{i2}, x_{i3}) = (453, 11, 86); \\w &= (0,01, 0,04, 0,002).\end{aligned}$$

Поскольку теперь мы думаем о признаках и весах как о векторах  $x_i$  и  $w$  соответственно, мы можем заменить сумму элементов этих векторов их скалярным произведением:

$$x_i^T w = \sum_{j=1}^n x_{ij} w_j = x_{i1} w_1 + x_{i2} w_2 + \dots + x_{in} w_n.$$

Скалярное произведение — это способ умножения двух векторов: мы умножаем соответствующие элементы векторов, после чего суммируем результаты. В приложении B можно найти более подробную информацию об умножении вектора на вектор.

Перевод формулы для скалярного произведения в код прост:

```
def dot(xi, w):
    n = len(w)
    result = 0.0
    for j in range(n):
        result = result + xi[j] * w[j]
    return result
```

Используя новую нотацию, мы можем переписать все уравнение для линейной регрессии как

$$g(x_i) = w_0 + x_i^T w,$$

где

- $w_0$  — компонент смещения;
- $w$  —  $n$ -мерный вектор весов.

Теперь мы можем использовать новую функцию `dot`, поэтому функция линейной регрессии в Python становится очень короткой:

```
def linear_regression(xi):
    return w0 + dot(xi, w)
```

В качестве альтернативы, если `xi` и `w` являются массивами NumPy, мы можем использовать для умножения встроенный метод `dot`:

```
def linear_regression(xi):
    return w0 + xi.dot(w)
```

Чтобы сделать его еще короче, мы можем объединить  $w_0$  и  $w$  в один  $(n + 1)$ -мерный вектор, добавив  $w_0$  к  $w$  прямо перед  $w_1$ :

$$w = (w_0, w_1, w_2 \dots w_n).$$

Таким образом, мы получаем вектор весов  $w$ , состоящий из компонента смещения  $w_0$ , за которым следуют веса  $w_1, w_2, \dots$  из исходного вектора весов  $w$ .

В Python это проделать очень легко. Если у нас уже есть старые веса в списке  $w$ , то нам нужно лишь выполнить следующую операцию:

```
w = [w0] + w
```

Помните, что оператор `+` в Python объединяет списки, поэтому `[1] + [2, 3, 4]` создаст новый список из четырех элементов: `[1, 2, 3, 4]`. В нашем случае  $w$  уже является списком, поэтому мы создаем новый  $w$  с одним дополнительным элементом в начале: `w0`.

Поскольку теперь  $w$  становится  $(n + 1)$ -мерным вектором, нам также нужно настроить вектор объектов  $x_i$  так, чтобы скалярное произведение по-прежнему работало. Это легко сделать, добавив фиктивный признак  $x_{i0}$ , который всегда принимает значение 1. Затем мы добавим этот новый фиктивный признак к  $x_i$  прямо перед  $x_{ii}$ :

$$x_i = (x_{i0}, x_{i1}, x_{i2} \dots x_{in}) = (1, x_{i1}, x_{i2} \dots x_{in}).$$

Или в коде:

```
xi = [1] + xi
```

Мы создаем новый список  $xi$  с 1 в качестве первого элемента, за которым следуют все элементы из старого списка  $xi$ .

С помощью этих модификаций мы можем выразить модель как скалярное произведение между новым  $x_i$  и новым  $w$ :

$$g(x_i) = x_i^T w.$$

В коде это выразить просто:

```
w0 = 7.17
w = [0.01, 0.04, 0.002]
w = [w0] + w

def linear_regression(xi):
    xi = [1] + xi
    return dot(xi, w)
```

Эти формулы для линейных регрессий эквивалентны, поскольку первый признак нового  $x_i$  равен 1, поэтому, умножая первый компонент  $x_i$  на первый компонент  $w$ , мы получаем компонент смещения, поскольку  $w_0 \times 1 = w_0$ .

Теперь мы готовы вернуться к общей картине и поговорить о матричной форме. В данных содержится много наблюдений, и  $x_i$  — одно из них. Таким образом, у нас есть  $m$  векторов признаков  $x_1, x_2, \dots, x_i, \dots, x_m$ , и каждый из этих векторов состоит из  $n + 1$  признаков:

$$x_1 = (1, x_{11}, x_{12} \dots x_{1n})$$

$$x_2 = (1, x_{21}, x_{22} \dots x_{2n})$$

...

$$x_i = (1, x_{i1}, x_{i2} \dots x_{in})$$

$$x_m = (1, x_{m1}, x_{m2} \dots x_{mn})$$

Мы можем сложить эти векторы вместе в виде строк матрицы. Назовем эту матрицу  $X$  (рис. 2.12).

$X =$	$=$	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>вектор <math>x_1</math></td></tr> <tr><td>вектор <math>x_2</math></td></tr> <tr><td>вектор <math>x_3</math></td></tr> <tr><td>...</td></tr> <tr><td>вектор <math>x_m</math></td></tr> </table>	вектор $x_1$	вектор $x_2$	вектор $x_3$	...	вектор $x_m$																				
вектор $x_1$																											
вектор $x_2$																											
вектор $x_3$																											
...																											
вектор $x_m$																											
<table border="1" style="display: inline-table; vertical-align: middle; border-collapse: collapse;"> <tr><td>1</td><td><math>x_{11}</math></td><td><math>x_{12}</math></td><td>...</td><td><math>x_{1n}</math></td></tr> <tr><td>1</td><td><math>x_{21}</math></td><td><math>x_{22}</math></td><td>...</td><td><math>x_{2n}</math></td></tr> <tr><td>1</td><td><math>x_{31}</math></td><td><math>x_{32}</math></td><td>...</td><td><math>x_{3n}</math></td></tr> <tr><td>...</td><td>...</td><td>...</td><td>...</td><td>...</td></tr> <tr><td>1</td><td><math>x_{m1}</math></td><td><math>x_{m2}</math></td><td>...</td><td><math>x_{mn}</math></td></tr> </table>	1	$x_{11}$	$x_{12}$	...	$x_{1n}$	1	$x_{21}$	$x_{22}$	...	$x_{2n}$	1	$x_{31}$	$x_{32}$	...	$x_{3n}$	...	...	...	...	...	1	$x_{m1}$	$x_{m2}$	...	$x_{mn}$	$=$	
1	$x_{11}$	$x_{12}$	...	$x_{1n}$																							
1	$x_{21}$	$x_{22}$	...	$x_{2n}$																							
1	$x_{31}$	$x_{32}$	...	$x_{3n}$																							
...	...	...	...	...																							
1	$x_{m1}$	$x_{m2}$	...	$x_{mn}$																							

**Рис. 2.12.** Матрица  $X$ , в которой наблюдения  $x_1, x_2, \dots, x_m$  являются строками

Посмотрим, как это выглядит в коде. Мы можем взять несколько строк из обучающего набора данных, например первую, вторую и десятую:

```
x1 = [1, 148, 24, 1385]
x2 = [1, 132, 25, 2031]
x10 = [1, 453, 11, 86]
```

Теперь объединим строки в другой список:

```
X = [x1, x2, x10]
```

Список  $X$  теперь содержит три списка. Мы можем думать об этом как о матрице  $3 \times 4$  — матрице с тремя строками и четырьмя столбцами:

```
X = [[1, 148, 24, 1385],
      [1, 132, 25, 2031],
      [1, 453, 11, 86]]
```

Каждый столбец этой матрицы представляет собой признак:

- 1) первый столбец — фиктивный признак с «1»;
- 2) второй столбец — мощность двигателя;

- 3) третий — MPG в городе;
- 4) и последний — популярность, или количество упоминаний в Twitter.

Вы уже знаете, что, для того чтобы сделать прогноз для одного вектора признаков, нам нужно вычислить скалярное произведение этого вектора признаков и вектора весов. Теперь у нас есть матрица  $X$ , которая в Python представляет собой список векторов признаков. Чтобы сделать прогнозы для всех строк матрицы, мы можем просто перебрать все строки  $X$  и вычислить скалярное произведение:

```
predictions = []

for xi in X:
    pred = dot(xi, w)
    predictions.append(pred)
```

В линейной алгебре это умножение матрицы на вектор: мы умножаем матрицу  $X$  на вектор  $w$ . Формула для линейной регрессии превращается в

$$g(X) = w_0 + Xw.$$

Результатом будет массив с прогнозами для каждой строки  $X$ . Более подробную информацию о матрично-векторном умножении можно найти в приложении В.

При такой формулировке матрицы код для применения линейной регрессии для составления прогнозов становится очень простым, как и перевод на NumPy:

```
predictions = X.dot(w)
```

### Упражнение 2.3

Когда мы умножаем матрицу  $X$  на вектор весов  $w$ , что мы получаем?

- A. Вектор  $y$  с фактической ценой.
- B. Вектор  $y$  с прогнозами цен.
- C. Одно число  $y$  с прогнозами цен.

### 2.3.2. Обучающая модель линейной регрессии

До сих пор мы рассматривали только прогнозирование. Чтобы иметь возможность это сделать, нам нужно знать веса  $w$ . Как мы их получим?

Мы узнаем веса из данных: используем целевую переменную  $y$ , чтобы найти такую  $w$ , которая наилучшим образом сочетает в себе признаки  $X$ . «Наилучшим образом» в случае линейной регрессии означает, что ошибка между прогнозами  $g(X)$  и фактическим целевым значением  $y$  сведена к минимуму.

У нас для этого есть несколько способов. Мы используем нормальное уравнение, которое служит самым простым методом реализации. Весовой вектор  $w$  может быть вычислен по следующей формуле:

$$w = (X^T X)^{-1} X^T y.$$

#### ПРИМЕЧАНИЕ

Выведение нормального уравнения выходит за рамки этой книги. Мы даем некоторое представление о том, как оно работает, в приложении B, но более подробно познакомиться с темой вам поможет учебник по машинному обучению. Книга *The Elements of Statistical Learning, 2nd edition* Г. Фридмана, Р. Тибширани и Т. Хасти (Friedman, Tibshirani, Hastie) будет весьма полезной на начальном этапе.

Эта математическая часть может напугать или запутать, но ее довольно легко перевести на NumPy:

- $X^T$  — транспонирование  $X$ . В NumPy это `X.T`;
- $X^T X$  — умножение матрицы на матрицу, которое мы можем выполнить с помощью метода `dot` из NumPy: `X.T.dot(X)`;
- $X^{-1}$  — величина, обратная  $X$ . Для обращения мы можем использовать функцию `np.linalg.inv`.

Таким образом, приведенная выше формула превращается непосредственно в `inv(X.T.dot(X)).dot(X.T).dot(y)`

Более подробную информацию об этом уравнении можно найти в приложении B.

Чтобы реализовать нормальное уравнение, нам нужно проделать следующее:

1. Создать функцию, которая принимает матрицу  $X$  с признаками и вектор  $y$  с целью.
2. Добавить фиктивный столбец (признак, который всегда содержит значение 1) в матрицу  $X$ .
3. Обучить модель: вычислить веса  $w$ , используя нормальное уравнение.
4. Разделить полученный  $w$  на смещение  $w_0$  и остальные веса и вернуть их.

Последний шаг — разделение  $w$  на компонент смещения и остаток — необязателен и в основном нужен для удобства; в противном случае нам пришлось бы добавлять фиктивный столбец каждый раз, когда мы захотим получить прогноз, вместо того чтобы сделать это один раз во время обучения.

Реализуем все это (листинг 2.2).

**Листинг 2.2.** Линейная регрессия, реализованная с помощью NumPy

```
def train_linear_regression(X, y):
    # добавление фиктивного столбца
    ones = np.ones(X.shape[0]) ← ❶ Создает массив, содержащий
    X = np.column_stack([ones, X]) ← только единицы
    # формула нормального уравнения
    XTX = X.T.dot(X) ← ❷ Добавляет массив из единиц
    XTX_inv = np.linalg.inv(XTX) ← в качестве первого столбца X
    w = XTX_inv.dot(X.T).dot(y) ← ❸ Вычисляет  $X^T X$ 
    ← ❹ Вычисляет обратную величину  $X^T X$ 
    ← ❺ Вычисляет остальную часть
    return w[0], w[1:] ← ❻ Нормального уравнения
                           ← ❼ Разбивает вектор весов на смещение и остальные веса
```

С помощью шести строк кода мы внедрили наш первый алгоритм машинного обучения. В ❶ мы создаем вектор, содержащий только единицы, который мы добавляем к матрице  $X$  в качестве первого столбца; это фиктивный признак в ❷. Далее мы вычисляем  $X^T X$  в ❸ и его обратное значение в ❹ и объединяем их, чтобы вычислить  $w$  в ❺. Наконец мы разделяем веса на смещение  $w_0$  и остальные веса  $w$  в ❻.

Функция `column_stack` в NumPy, которую мы использовали для добавления столбца, поначалу может смутить, поэтому рассмотрим ее более пристально:

```
np.column_stack([ones, X])
```

Она принимает список массивов NumPy, который в нашем случае содержит `ones` и `X`, и складывает их (рис. 2.13).

```
ones = np.array([1, 1])
ones
```

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

```
X = np.array([[2, 3], [4, 5]])
X
```

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

```
np.column_stack([ones, X])
```

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

Единица       $X$

**Рис. 2.13.** Функция `column_stack` принимает список массивов NumPy и складывает их в столбцы. В нашем случае функция добавляет массив с единицами в качестве первого столбца матрицы

Если разделить веса на компонент смещения и остаток, то формула линейной регрессии для составления прогнозов немного изменится:

$$g(X) = w_0 + Xw.$$

Это по-прежнему очень легко перевести на NumPy:

```
y_pred = w0 + X.dot(w)
```

Давайте используем ее для нашего проекта!

## 2.4. ПРОГНОЗИРОВАНИЕ ЦЕНЫ

Мы рассмотрели уже много теоретических вопросов, поэтому пора вернуться к нашему проекту, прогнозированию цены на автомобиль. Теперь в нашем распоряжении есть функция для обучения модели линейной регрессии, поэтому воспользуемся ею для построения простого базового решения.

### 2.4.1. Базовое решение

Однако для того чтобы иметь возможность использовать эту функцию, нам потребуются некоторые данные: матрица  $X$  и вектор с целевой переменной  $y$ . Мы уже подготовили  $y$ , но у нас все еще нет  $X$ : то, что у нас есть прямо сейчас, — датафрейм, а не матрица. Итак, чтобы создать эту матрицу  $X$ , нам потребуется извлечь ряд признаков из нашего набора данных.

Мы начнем с очень наивного способа создания признаков — выберем несколько числовых признаков и сформируем из них матрицу  $X$ . В предыдущем примере мы использовали лишь три признака. На этот раз мы добавим еще несколько и используем следующие столбцы:

- `engine_hp`;
- `engine_cylinders`;
- `highway_mpg`;
- `city_mpg`;
- `popularity`.

Выберем признаки из датафрейма и запишем их в новую переменную `df_num`:

```
base = ['engine_hp', 'engine_cylinders', 'highway_mpg', 'city_mpg',
        'popularity']
df_num = df_train[base]
```

Как обсуждалось в разделе 2.2, посвященном исследовательскому анализу данных, в наборе данных отсутствуют значения. Нам придется что-то с этим делать, поскольку модель линейной регрессии не может автоматически обрабатывать пропущенные значения.

Один из вариантов — удалить все строки, содержащие хотя бы одно пропущенное значение. Этот подход, однако, имеет некоторые недостатки. Самое главное — мы потеряем информацию, которая содержится в других столбцах. Даже если не знать количество дверей, мы все равно можем получить много информации об автомобиле, например марку, модель, возраст и другие характеристики, которые не хотим потерять.

Другой вариант — заполнить недостающие значения каким-либо другим значением. Таким образом, мы не теряем информацию в других столбцах и по-прежнему можем делать прогнозы, даже если в строке отсутствуют значения. Самый простой из возможных подходов — заполнить недостающие значения нулями. Мы можем использовать метод Pandas под названием `fillna`:

```
df_num = df_num.fillna(0)
```

Этот метод неидеально справляется с пропущенными значениями, но часто работает достаточно хорошо. Если мы установим значение отсутствующего признака равным нулю, то соответствующий признак будет просто проигнорирован.

### **ПРИМЕЧАНИЕ**

Альтернативный вариант — замена отсутствующих значений средними. Для некоторых переменных, например количества цилиндров, нулевое значение не имеет особого смысла: автомобиль не может не иметь хотя бы одного цилиндра. Однако это усложнит наш код и не окажет существенного влияния на результат. Вот почему мы остановимся на более простом подходе и заменим недостающие значения нулями.

Нетрудно понять, почему установка для признака нулевого значения — то же самое, что его игнорирование. Вспомним формулу для линейной регрессии. В нашем случае имеется пять признаков, поэтому формула выглядит так:

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2 + x_{i3}w_3 + x_{i4}w_4 + x_{i5}w_5.$$

Если третий элемент отсутствует и мы замещаем его нулем, то  $x_{i3}$  становится нулевым:

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2 + 0 \cdot w_3 + x_{i4}w_4 + x_{i5}w_5.$$

В этом случае независимо от веса  $w_3$  для данного признака произведение  $x_{i3}w_3$  всегда будет равно нулю. Другими словами, этот признак не внесет никакого

вклада в окончательный прогноз, и мы будем основывать наш прогноз только на тех признаках, которые не пропущены:

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2 + x_{i4}w_4 + x_{i5}w_5.$$

Теперь нам нужно преобразовать этот датафрейм в массив NumPy. Самый простой способ — использовать его свойство `values`:

```
X_train = df_num.values
```

`X_train` — это матрица, двумерный массив NumPy. Ее мы можем использовать в качестве входных данных для нашей функции `linear_regression`. Назовем это:

```
w_0, w = train_linear_regression(X_train, y_train)
```

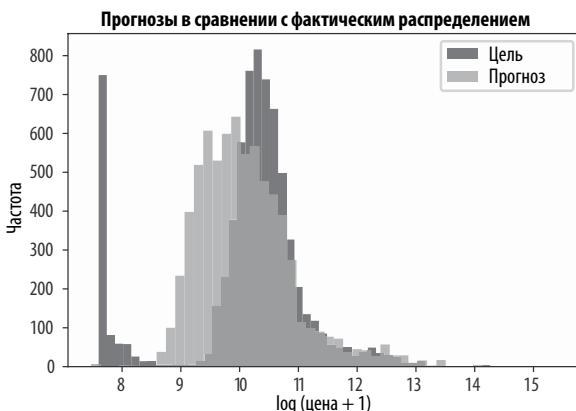
Мы только что обучили первую модель! Теперь мы можем применить ее к обучающим данным, чтобы увидеть, насколько хороший прогноз она дает:

```
y_pred = w_0 + X_train.dot(w)
```

Чтобы увидеть, насколько хороши прогнозы, мы можем использовать `histplot` — функцию от Seaborn для построения гистограмм, которую мы использовали ранее — для построения прогнозируемых значений и сравнения их с фактическими ценами:

```
sns.histplot(y_pred, label='prediction')
sns.histplot(y_train, label='target')
plt.legend()
```

Из графика (рис. 2.14) видно, что распределение значений, которое мы предсказали, сильно отличается от фактических значений. Результат может



**Рис. 2.14.** Распределение прогнозируемых значений (светло-серые) и фактических значений (темно-серые). Мы видим, что наши прогнозы не так уж хороши; они сильно отличаются от фактического распределения

указывать на то, что модель недостаточно эффективна, чтобы уловить распределение целевой переменной. Это не должно удивлять: модель, которую мы использовали, довольно проста и включает в себя всего пять очень простых признаков.

### 2.4.2. RMSE: оценка качества модели

Просмотр графиков и сравнение распределений фактической целевой переменной с прогнозами — хороший способ оценить качество, но у нас не получится проделывать это каждый раз, когда мы что-то меняем в модели. Вместо этого следует использовать метрику, которая количественно определяет качество модели. Существует множество показателей для оценки того, насколько хорошо ведет себя регрессионная модель. Наиболее часто используемым из них является *корень среднеквадратичной ошибки* — сокращенно RMSE (root mean squared error).

RMSE сообщает нам, насколько велики ошибки, допускаемые моделью. Он вычисляется с помощью следующей формулы:

$$\text{RMSE} = \sqrt{\frac{1}{m} \sum_{i=1}^m (g(x_i) - y_i)^2}.$$

Попробуем разобраться, что здесь происходит. Для начала рассмотрим состав суммы. Мы имеем:

$$(g(x_i) - y_i)^2.$$

Это разница между прогнозом, который мы делаем для наблюдения, и фактическим целевым значением для этого наблюдения (рис. 2.15).

$g(x_1)$	$g(x_2)$	$g(x_3)$	$\dots$	$g(x_m)$
9.6	7.3	9.6	$\dots$	10.8

—

9.5	10.3	9.8	$\dots$	10.7
$y_1$	$y_2$	$y_3$		$y_m$

=

0.1	-3.0	-0.2	$\dots$	0.1
$g(x_1) - y_1$	$g(x_3) - y_3$	$g(x_m) - y_m$		

**Рис. 2.15.** Разница между прогнозами  $g(x)$  и фактическими значениями  $y_i$

Затем мы используем квадрат разницы, что придает гораздо больший вес большим разницам. Например, если мы прогнозируем 9,5, а фактическое значение равно 9,6, то разница составит 0,1, поэтому ее квадрат равен 0,01, что довольно немного. Но если мы прогнозируем 7,3, а фактическое значение равно 10,3, то разница равна 3, а квадрат разницы равен 9 (рис. 2.16).

$$\left( \begin{array}{ccccc} 0.1 & -3.0 & -0.2 & \dots & 0.1 \end{array} \right)^2 = \begin{array}{ccccc} 0.01 & 9.0 & 0.04 & \dots & 0.01 \end{array}$$

**Рис. 2.16.** Квадрат разницы между прогнозируемыми и фактическими значениями. Для больших разниц квадрат довольно большой

Это часть SE (*квадратичная ошибка*) RMSE.

Далее, у нас есть сумма:

$$\sum_{i=1}^m (g(x_i) - y_i)^2.$$

Это суммирование проходит по всем  $t$  наблюдениям и объединяет все квадратичные ошибки (рис. 2.17) в одно число.

$$\sum_{i=1}^m \left( \begin{array}{|c|c|c|c|c|} \hline 0.01 & 9.0 & 0.04 & \dots & 0.01 \\ \hline \end{array} \right) = \boxed{9.06}$$

**Рис. 2.17.** Результат суммирования всех квадратов разниц представляет собой одно число

Если мы разделим эту сумму на  $m$ , то получим среднеквадратичную ошибку:

$$\frac{1}{m} \sum_{i=1}^m (g(x_i) - y_i)^2$$

Это среднеквадратичная ошибка, которую наша модель допускает в среднем, — компонент  $M$  (*mean*) сокращения RMSE, или *среднеквадратичная ошибка* (MSE). MSE является хорошей метрикой и сама по себе (рис. 2.18).

$$\frac{1}{m} \sum_{i=1}^m \left( \underbrace{\begin{bmatrix} 0.01 & 9.0 & 0.04 & \dots & 0.01 \end{bmatrix}}_{\text{Среднее}} \right) = \frac{1}{m} \begin{bmatrix} 9.06 \end{bmatrix} = \boxed{2.26}$$

Квадратичная ошибка

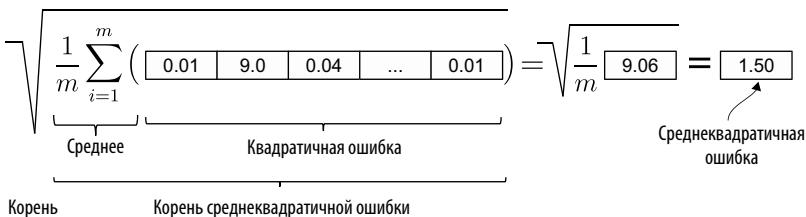
Среднеквадратичная ошибка

**Рис. 2.18.** MSE определяется путем вычисления среднего значения квадратов ошибок

Наконец мы берем из всего этого квадратный корень:

$$\text{RMSE} = \sqrt{\frac{1}{m} \sum_{i=1}^m (g(x_i) - y_i)^2}.$$

Это компонент R (*root*) сокращения RMSE (рис. 2.19).



**Рис. 2.19.** RMSE: сначала мы вычисляем MSE, а затем вычисляем ее квадратный корень

При использовании NumPy для реализации RMSE мы можем воспользоваться *векторизацией*: процессом применения одной и той же операции ко всем элементам одного или нескольких массивов NumPy. Векторизация дает нам множество преимуществ. Во-первых, код более лаконичен: нам не нужно использовать циклы для применения одной и той же операции к каждому элементу массива. Во-вторых, векторизованные операции выполняются намного быстрее, чем простые циклы `for` из Python.

Рассмотрим следующую реализацию (листинг 2.3).

### Листинг 2.3. Реализация вычисления корня среднеквадратичной ошибки

```
def rmse(y, y_pred):
    error = y_pred - y
    mse = (error ** 2).mean()
    return np.sqrt(mse)
```

- ❶ Вычисляет разницу между прогнозом
- ❷ Вычисляет MSE: сначала вычисляет квадратичную ошибку, а затем вычисляет среднее значение
- ❸ Извлекает квадратный корень, чтобы получить RMSE

В ❶ мы вычисляем поэлементную разницу между вектором с прогнозами и вектором с целевой переменной. Результатом является новый массив NumPy `error`, содержащий разницы. В ❷ мы выполняем две операции в одной строке: вычисляем квадрат каждого элемента массива `error`, а затем получаем среднее значение результата, что дает нам MSE. В ❸ мы вычисляем квадратный корень, чтобы получить RMSE.

Поэлементные операции в NumPy и Pandas довольно удобны. Мы можем применить операцию ко всему массиву NumPy (или серии в Pandas), не прибегая к написанию циклов.

Например, в первой строке нашей функции `rmse` мы вычисляем разницу между прогнозируемыми и фактическими ценами:

```
error = y_pred - y
```

Здесь для каждого элемента `y_pred` мы вычитаем соответствующий элемент `y`, а затем помещаем результат в новый массив `error` (рис. 2.20).

y_pred	9.55	9.36	9.67	8.65	10.87
-					
y	9.58	9.89	9.89	7.6	10.94
=					
error	-0.03	-0.5	-0.22	1.05	-0.07

**Рис. 2.20.** Разница между `y_pred` и `y` по элементам.

Результат записывается в массив `error`

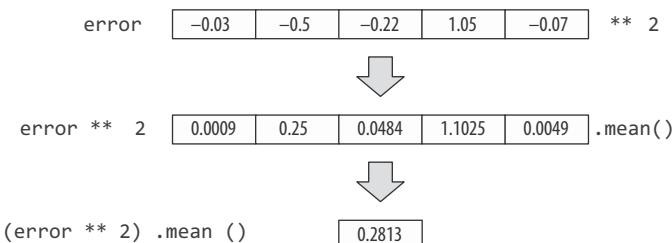
Далее мы вычисляем квадрат каждого элемента массива `error`, после чего вычисляем его среднее значение, чтобы получить среднеквадратичную ошибку нашей модели (рис. 2.21).

```
mse = (error ** 2).mean()
```

**Рис. 2.21.** Чтобы вычислить MSE, мы сначала вычисляем квадрат каждого элемента в массиве ошибок, а затем вычисляем среднее значение результата

Чтобы понять, что здесь происходит, требуется знать, что оператор степени (`**`) также применяется поэлементно, поэтому результатом является новый массив, в котором все элементы исходного массива возведены в квадрат.

Когда у нас есть этот новый массив с квадратами элементов, мы просто вычисляем его среднее значение с помощью метода `mean()` (рис. 2.22).



**Рис. 2.22.** Оператор степени (\*\*) применен поэлементно к массиву ошибок.

В результате получается еще один массив, в котором каждый элемент возведен в квадрат. Далее мы вычисляем среднее значение массива с квадратичной ошибкой для вычисления MSE

Наконец мы вычисляем квадратный корень из среднего значения и получаем RMSE:

```
np.sqrt(mse)
```

Теперь мы можем использовать RMSE для оценки качества модели:

```
rmse(y_train, y_pred)
```

Код выдает результат 0,75. Это число говорит нам о том, что в среднем прогнозы модели отклоняются на 0,75. Сам по себе этот результат может показаться не очень полезным, но мы можем использовать его для сравнения данной модели с другими. Если одна модель имеет лучший (более низкий) RMSE, чем другая, это указывает на то, что она лучше.

### 2.4.3. Проверка модели

В примере выше мы вычисляли RMSE на обучающем наборе. Результат полезен для понимания, но не отражает того, как модель будет использоваться в дальнейшем. Она будет применяться для прогнозирования цен на автомобили, которые раньше не встречала. Для этой цели мы ранее выделили проверочный набор данных. Мы намеренно не используем его для обучения и сохраняем для проверки модели.

Мы уже разделили наши данные на несколько частей: `df_train`, `df_val` и `df_test`. Мы также создали матрицу `X_train` из `df_train` и использовали `X_train` и `y_train` для обучения модели. Теперь нам нужно выполнить те же шаги, чтобы получить `X_val` — матрицу с признаками, вычисленными из проверочного набора данных. Затем мы можем применить модель к `X_val`, чтобы получить прогнозы и сравнить их с `y_val`.

Сначала мы создаем матрицу `X_val`, следуя тем же шагам, что и в случае `X_train`:

```
df_num = df_val[base]
df_num = df_num.fillna(0)
X_val = df_num.values
```

Мы готовы применить модель к `X_val` для получения прогнозов:

```
y_pred = w_0 + X_val.dot(w)
```

Массив `y_pred` содержит прогнозы для проверочного набора данных. Теперь мы используем `y_pred` и сравниваем его с фактическими ценами из `y_val` с помощью функции RMSE, которую реализовали ранее:

```
rmse(y_val, y_pred)
```

Код выдает значение `0.76`, и это число мы должны использовать для сравнения моделей.

В предыдущем коде можно наблюдать некоторое дублирование: обучающие и проверочные тесты требуют одинаковой предварительной обработки, и мы дважды писали один и тот же код. Таким образом, имеет смысл перенести эту логику в отдельную функцию и избежать дублирования кода.

Можно назвать эту функцию `prepare_X`, поскольку она создает матрицу `X` из датафрейма (листинг 2.4).

**Листинг 2.4.** Функция `prepare_X` для преобразования датафрейма в матрицу

```
def prepare_X(df):
    df_num = df[base]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X
```

Обучение и оценка упрощаются и теперь выглядят следующим образом:

```
X_train = prepare_X(df_train)           | Обучает модель
w_0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)               | Применяет модель к проверочному
y_pred = w_0 + X_val.dot(w)            | набору данных
print('validation:', rmse(y_val, y_pred)) ← | Вычисляет RMSE на основе
                                              | проверочных данных
```

Это дает нам возможность проверить, приводят ли какие-либо корректировки модели к улучшению качества прогнозирования. В качестве следующего шага добавим дополнительные признаки и проверим, приведет ли это к более низкому RMSE.

## 2.4.4. Простое конструирование признаков

У нас уже есть базовая модель с простыми признаками. Чтобы ее улучшить, мы можем добавить дополнительные признаки: создаем их и добавляем к уже существующим. Как мы уже знаем, этот процесс называется *конструированием признаков*.

Поскольку мы уже настроили систему проверки, можем легко определить, улучшает ли добавление новых признаков качество модели. Наша цель — улучшить RMSE, рассчитанную на основе проверочных данных.

Сначала мы создаем новый признак `age` из признака `year`. Возраст автомобиля должен быть очень полезен при прогнозировании цены: интуитивно понятно, что чем новее автомобиль, тем дороже он должен стоить.

Поскольку набор данных был создан в 2017 году (что мы можем проверить, проверив `df_train.year.max()`), можно рассчитать возраст, вычтя год выпуска автомобиля из 2017:

```
df_train['age'] = 2017 - df_train.year
```

Эта операция является поэлементной. Мы вычисляем разницу между 2017 годом и каждым элементом серии значений года. Результатом является новая серия Pandas, содержащая разницы, которые мы записываем обратно в датафрейм в качестве столбца возраста.

Мы уже знаем, что нам придется использовать одну и ту же предварительную обработку дважды: к наборам обучения и проверки. Поскольку мы не хотим повторять код извлечения признаков несколько раз, переместим эту логику в функцию `prepare_X` (листинг 2.5).

**Листинг 2.5.** Создание признака возраста в функции `prepare_X`

```
def prepare_X(df):
    df = df.copy()          ❶ Создает копию входного параметра для
    features = base.copy()  ❷ Создает копию базового списка
    df['age'] = 2017 - df.year
    features.append('age')  ❸ Вычисляет признак возраста
    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X               ❹ Добавляет возраст к списку имен
                            признаков, которые мы используем
                            для модели
```

Способ реализации функции на этот раз несколько отличается от предыдущей версии. Остановимся на этих различиях. Сначала в ❶ мы создаем копию датафрейма `df`, который передаем в функцию. Позже в коде мы модифицируем `df`, добавив дополнительные строки в ❸. Такое поведение известно как *побочный эффект*:

вызывающий функцию может не ожидать, что функция изменит датафрейм. Чтобы предотвратить такой неприятный сюрприз, мы вместо этого модифицируем копию исходного датафрейма. В ❷ мы создаем копию списка базовых признаков по той же причине. Позже мы расширим этот список новыми признаками ❸, но не хотим изменять исходный список. Остальная часть кода остается прежней.

Проверим, приведет ли добавление признака `age` к каким-либо улучшениям:

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

Код выводит следующее:

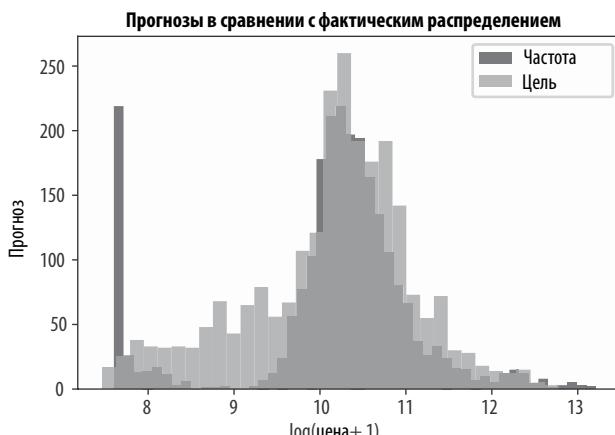
```
validation: 0.517
```

Ошибка проверки составляет 0,517, что является хорошим улучшением по сравнению с 0,76 — значением, которое мы имели в базовом решении. Таким образом, мы приходим к выводу, что добавление возраста автомобиля действительно полезно при составлении прогнозов.

Мы также можем взглянуть на распределение прогнозируемых значений:

```
sns.histplot(y_pred, label='prediction')
sns.histplot(y_val, label='target')
plt.legend()
```

Мы видим (рис. 2.23), что распределение прогнозов теперь намного точнее соответствует целевому. Это подтверждает и проверочный балл RMSE.



**Рис. 2.23.** Распределение прогнозируемых (светло-серые) и фактических (темно-серые) значений. Благодаря новым признакам модель оказывается гораздо ближе к исходному распределению

## 2.4.5. Обработка категориальных переменных

Мы увидели, что добавление признака `age` оказалось весьма полезно для модели. Продолжим добавлять новые признаки. Один из столбцов, который мы можем использовать далее, — это количество дверей. Данная переменная представлена числом и может принимать три значения: 2, 3 и 4 двери. Несмотря на соблазн поместить переменную в модель как есть, на самом деле это не числовая переменная: мы не можем сказать, что при добавлении еще одной двери цена автомобиля вырастет (или упадет) на определенную сумму. Скорее всего, это категориальная переменная.

*Категориальные переменные* описывают характеристики объектов и могут принимать одно из нескольких возможных значений. Марка автомобиля — категориальная переменная; например, это может быть Toyota, BMW, Ford или любая другая марка. Категориальную переменную легко распознать по ее значениям, которые обычно представляют собой строки, а не числа. Однако это не всегда так. Количество дверей, например, категориально: оно может принимать только одно из трех возможных значений (2, 3 и 4).

Мы можем использовать категориальные переменные в модели машинного обучения по-разному. Один из простейших способов — кодирование таких переменных набором двоичных признаков с отдельным признаком для каждого отдельного значения.

В нашем случае мы создадим три двоичных объекта: `num_doors_2`, `num_doors_3` и `num_doors_4`. Если у автомобиля две двери, то `num_doors_2` будет установлено в 1, а остальные в 0. Если у автомобиля три двери, то `num_doors_3` получит значение 1, и то же самое относится к `num_doors_4`.

Этот метод кодирования категориальных переменных называется *прямым кодированием*. Мы очень подробно рассмотрим его в главе 3. А пока остановимся на самом простом способе реализовать это кодирование: переберем возможные значения (2, 3 и 4) и для каждого проверим, соответствует ли ему значение наблюдения.

Добавим эти строки в функцию `prepare_X`:

```
Перебирает возможные значения
переменной «количество дверей»
for v in [2, 3, 4]:
    ❶ Дает признаку значимое имя, например
    num_doors_2 для v=2
    feature = 'num_doors_%s' % v
    value = (df['number_of_doors'] == v).astype(int)
    df[feature] = value
    features.append(feature)
    ❷ Добавляет признак обратно
    в датафрейм, используя имя из ❶
❸ Осуществляет прямое
кодирование признака
```

Код может показаться трудным для понимания, поэтому уделим больше внимания тому, что здесь происходит. Самая сложная строка — это ❸:

```
(df['number_of_doors'] == v).astype(int)
```

Сначала мы рассмотрим выражение внутри круглых скобок, где мы используем оператор равенства (`==`). Эта операция также является поэлементной, подобной тем, которые мы использовали ранее при вычислении RMSE. В нашем случае операция создает новую серию Pandas. Если элементы исходной серии равны `v`, то соответствующие элементы в результате получают значение `True`; в противном случае элементы получают значение `False`. Операция создает серию значений `True/False`.

Поскольку `v` имеет три значения (2, 3 и 4) и мы применяем эту операцию к каждому значению `v`, мы создаем три серии (рис. 2.24).

Series	<code>v = 2</code>	<code>v = 3</code>	<code>v = 4</code>
2	True	False	False
4	False	False	True
2	True	False	False
4	False	True	False
3	False	True	False

**Рис. 2.24.** Мы используем оператор `==`, чтобы создать новую серию из исходной: одну для двух дверей, одну для трех и одну для четырех

Далее мы преобразуем серию `Boolean` в целые числа таким образом, чтобы значение `True` стало 1, а значение `False` — 0, что легко проделать с помощью метода `astype(int)` (рис. 2.25). Теперь мы можем использовать результаты в качестве признаков и поместить их в линейную регрессию.

True	False	False	<code>astype(int)</code>	1	0	0
False	False	True		0	0	1
True	False	False		1	0	0
False	False	True		0	0	1
False	True	False		0	1	0

**Рис. 2.25.** Использование `astype(int)` для преобразования серий с логическими значениями в целые числа

Количество дверей, как мы уже обсуждали, является категориальной переменной, которая лишь кажется числовой, поскольку ее значения являются целыми числами (2, 3 и 4). Все остальные категориальные переменные, которые у нас есть в наборе данных, выражены строками.

Мы можем использовать тот же подход для кодирования других категориальных переменных. Начнем с `make`. Для наших целей должно быть достаточно получить и использовать только наиболее часто встречающиеся значения. Выясним, каковы пять наиболее частых значений:

```
df['make'].value_counts().head(5)
```

Код выводит следующее:

chevrolet	1123
ford	881
volkswagen	809
toyota	746
dodge	626

Мы берем эти значения и используем их для кодирования `make` таким же образом, как и в случае с переменной количества дверей.

Далее мы создаем пять новых переменных с именами `is_make_chevrolet`, `is_make_ford`, `is_make_volkswagen`, `is_make_toyota` и `is_make_dodge`:

```
for v in ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge']:
    feature = 'is_make_%s' % v
    df[feature] = (df['make'] == v).astype(int)
    features.append(feature)
```

Теперь весь `prepare_X` должен выглядеть следующим образом (листинг 2.6).

#### Листинг 2.6. Обработка категориальных переменных `number_of_doors` и `make`

```
def prepare_X(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')                                Кодирует переменную
                                                        количества дверей

    for v in [2, 3, 4]::                                ←
        feature = 'num_doors_%s' % v
        df[feature] = (df['number_of_doors'] == v).astype(int)
        features.append(feature)                         Кодирует
                                                        переменную
                                                        марки

    for v in ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge']: ←
        feature = 'is_make_%s' % v
        df[feature] = (df['make'] == v).astype(int)
        features.append(feature)

    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X
```

Выясним, улучшает ли этот код RMSE модели:

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

Код выводит следующее:

```
validation: 0.507
```

Предыдущее значение составляло 0,517, так что нам удалось еще больше улучшить показатель RMSE.

Мы можем использовать еще несколько переменных: `engine_fuel_type`, `transmission_type`, `driven_wheels`, `market_category`, `vehicle_size` и `vehicle_style`. Проделаем то же самое и для них. После внесения изменений `prepare_X` выглядит немного сложнее (листинг 2.7).

### Листинг 2.7. Обработка большего количества категориальных переменных в функции `prepare_X`

```
def prepare_X(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')

    for v in [2, 3, 4]:
        feature = 'num_doors_%s' % v
        df[feature] = (df['number_of_doors'] == v).astype(int)
        features.append(feature)

    for v in ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge']:
        feature = 'is_make_%s' % v
        df[feature] = (df['make'] == v).astype(int)
        features.append(feature)

    for v in ['regular_unleaded', 'premium_unleaded_(required)', 'premium_unleaded_(recommended)', 'flex-fuel_(unleaded/e85)']:
        feature = 'is_type_%s' % v
        df[feature] = (df['engine_fuel_type'] == v).astype(int)
        features.append(feature)

    for v in ['automatic', 'manual', 'automated_manual']:
        feature = 'is_transmission_%s' % v
        df[feature] = (df['transmission_type'] == v).astype(int)
        features.append(feature)
```

```

for v in ['front_wheel_drive', 'rear_wheel_drive',
          'all_wheel_drive', 'four_wheel_drive']:
    feature = 'is_driven_wheels_%s' % v
    df[feature] = (df['driven_wheels'] == v).astype(int)
    features.append(feature)

for v in ['crossover', 'flex_fuel', 'luxury',
          'luxury,performance', 'hatchback']:
    feature = 'is_mc_%s' % v
    df[feature] = (df['market_category'] == v).astype(int)
    features.append(feature)

for v in ['compact', 'midsize', 'large']:
    feature = 'is_size_%s' % v
    df[feature] = (df['vehicle_size'] == v).astype(int)
    features.append(feature)

for v in ['sedan', '4dr_suv', 'coupe', 'convertible',
          '4dr_hatchback']:
    feature = 'is_style_%s' % v
    df[feature] = (df['vehicle_style'] == v).astype(int)
    features.append(feature)
    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X

```

Кодирует количество ведущих колес

Кодирует категорию маркетинга

Кодирует размер

Кодирует стиль

Проведем тест:

```

X_train = prepare_X(df_train)
w_0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))

```

Число, которое мы видим, значительно ухудшилось по сравнению с прежним. Мы получаем 34,2, что намного больше, чем число 0,5, которое было у нас раньше.

### ПРИМЕЧАНИЕ

Полученное вами число может отличаться в зависимости от версий Python, NumPy, версий зависимостей NumPy, операционной системы и других факторов. Но скачок показателя проверки с 0,5 до чего-то значительно большего всегда должен настороживать.

Вместо улучшения новые функции значительно ухудшили результат. К счастью, у нас есть этап проверки, который помогает нам выявлять эту проблему. В следующем подразделе мы увидим, почему подобное происходит и как с этим бороться.

## 2.4.6. Регуляризация

Мы увидели, что добавление новых признаков не всегда помогает, а в нашем случае все вообще намного ухудшилось. Причиной такого поведения служит неустойчивость численного решения. Вспомним формулу нормального уравнения:

$$\boldsymbol{w} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}.$$

Одно из слагаемых в уравнении – обращение матрицы  $\boldsymbol{X}^T \boldsymbol{X}$ :

$$(\boldsymbol{X}^T \boldsymbol{X})^{-1}.$$

В нашем случае проблема таится в обращении. Иногда при добавлении новых столбцов в  $\boldsymbol{X}$  мы можем случайно добавить столбец, представляющий собой комбинацию других столбцов. Например, если у нас уже есть признак «миль на галлон по городу» и мы решаем добавить «километры на литр по городу», то второй признак будет таким же, как и первый, но умноженным на некую константу.

Когда подобное происходит,  $\boldsymbol{X}^T \boldsymbol{X}$  становится *неопределенной* или *сингулярной*; это значит, невозможно получить обращение данной матрицы. Если мы попытаемся обратить сингулярную матрицу, то NumPy сообщит нам об этом, выдав `LinAlgError`:

```
LinAlgError: Singular matrix
```

Однако наш код не вызвал никаких исключений. Это произошло потому, что у нас обычно не бывает столбцов, которые представляют собой идеальные линейные комбинации других столбцов. Реальные данные часто зашумлены, имеют ошибки измерения (например, запись 1,3 вместо 13 для миль на галлон), ошибки округления (например, сохранение 0,0999999 вместо 0,1) и многие другие. Технически такие матрицы не являются сингулярными, поэтому NumPy на них не реагирует.

Однако по этой причине некоторые значения в весах становятся чрезвычайно большими — намного больше, чем должны.

Если мы посмотрим на значения наших  $w_0$  и  $\boldsymbol{w}$ , то увидим, что это действительно так. Например, компонент смещения  $w_0$  имеет значение 5788519290303866,0 (значение может варьироваться в зависимости от компьютера, операционной системы и версии NumPy), а некоторые компоненты  $\boldsymbol{w}$ , в свою очередь, содержат чрезвычайно большие отрицательные значения.

В численной линейной алгебре такие проблемы называются *проблемами неустойчивости численного решения* и обычно решаются с помощью методов регуляризации. Цель *регуляризации* состоит в том, чтобы убедиться, что обращение существует, вследствие чего матрица становится обратимой. Регуляризация является важной концепцией машинного обучения и означает контроль над весами модели, обеспечивая их должное поведение и не позволяя им слишком увеличиваться, как произошло в нашем случае.

Один из способов выполнить регуляризацию — добавить небольшое число к каждому диагональному элементу матрицы. Тогда мы получаем следующую формулу для линейной регрессии:

$$w = (X^T X + \alpha I)^{-1} X^T y.$$

### **ПРИМЕЧАНИЕ**

Регуляризованную линейную регрессию часто называют гребневой. Многие библиотеки, включая Scikit-learn, используют термины ridge для обозначения регуляризованной линейной регрессии, а linear — для обозначения нерегуляризованной модели.

Рассмотрим ту часть, которая изменилась: матрицу, которую нам нужно обратить. Вот как она выглядит:

$$X^T X + \alpha I.$$

Эта формула говорит, что нам нужна *I — единичная матрица*, которая представляет собой матрицу с единицами на главной диагонали и нулями в остальных случаях. Мы умножаем эту единичную матрицу на число  $\alpha$ . Таким образом, все значения, находящиеся на диагонали *I*, становятся  $\alpha$ . Затем мы суммируем  $\alpha I$  и  $X^T X$ , что добавляет  $\alpha$  ко всем диагональным элементам  $X^T X$ .

Эту формулу можно непосредственно перевести в код NumPy:

```
XTX = X_train.T.dot(X_train)
XTX = XTX + 0.01 * np.eye(XTX.shape[0])
```

Функция `np.eye` создает двумерный массив NumPy, который также является единичной матрицей. Когда мы умножаем на 0,01, единицы по диагонали становятся 0,01, поэтому, добавляя эту матрицу в `XTX`, мы добавляем только 0,01 к ее основной диагонали (рис. 2.26).

```
np.eye(4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

```
np.eye(4) * 0.01
array([[0.01, 0. , 0. , 0. ],
       [0. , 0.01, 0. , 0. ],
       [0. , 0. , 0.01, 0. ],
       [0. , 0. , 0. , 0.01]])
```

```
XTX = np.array([
    [0, 1, 2, 3],
    [0, 1, 2, 3],
    [0, 1, 2, 3],
    [0, 1, 2, 3],
])
XTX + 0.01 * np.eye(4)
array([[0.01, 1. , 2. , 3. ],
       [0. , 1.01, 2. , 3. ],
       [0. , 1. , 2.01, 3. ],
       [0. , 1. , 2. , 3.01]])
```

А. Функция `eye` из NumPy создает единичную матрицу

Б. Когда мы умножаем единичную матрицу на число, это число переходит в главную диагональ результата

В. Эффект добавления единичной матрицы, умноженной на 0,01, к другой матрице соответствует добавлению 0,01 к главной диагонали этой матрицы

**Рис. 2.26.** Использование единичной матрицы для добавления 0,01 к главной диагонали квадратной матрицы

Создадим новую функцию, которая использует эту идею и реализует линейную регрессию с регуляризацией (листинг 2.8).

#### Листинг 2.8. Линейная регрессия с регуляризацией

```
def train_linear_regression_reg(X, y, r=0.0):
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    XTX = X.T.dot(X)
    reg = r * np.eye(XTX.shape[0])
    XTX = XTX + reg
    XTX_inv = np.linalg.inv(XTX)
    w = XTX_inv.dot(X.T).dot(y)

    return w[0], w[1:]
```

← Управляет величиной регуляризации с помощью параметра `r`

Добавляет `r` к основной диагонали `XTX`

Функция очень похожа на линейную регрессию, но несколько строк все же отличаются. Для начала есть дополнительный параметр `r`, который управляет величиной регуляризации, — это соответствует числу  $\alpha$  в формуле, которое мы добавляем к главной диагонали  $XTX$ .

Регуляризация влияет на окончательное решение, уменьшая компоненты  $w$ . Мы можем увидеть, что чем больше регуляризации добавляем, тем меньше становятся веса.

Проверим, что происходит с нашими весами для разных значений  $r$ :

```
for r in [0, 0.001, 0.01, 0.1, 1, 10]:
    w_0, w = train_linear_regression_reg(X_train, y_train, r=r)
    print('%5s, %.2f, %.2f, %.2f' % (r, w_0, w[13], w[21]))
```

Код выводит следующее:

```
0, 5788519290303866.00, -9.26, -5788519290303548.00
0.001, 7.20, -0.10, 1.81
0.01, 7.18, -0.10, 1.81
0.1, 7.05, -0.10, 1.78
1, 6.22, -0.10, 1.56
10, 4.39, -0.09, 1.08
```

Мы начинаем с 0, что является нерегуляризованным решением, и получаем очень большие числа. Затем мы пробуем 0,001 и увеличиваем его в 10 раз на каждом шаге: 0,01, 0,1, 1 и 10. Мы видим, что выбранные нами значения уменьшаются по мере роста  $r$ .

Теперь выясним, помогает ли регуляризация решить нашу задачу и какую RMSE мы в итоге получим. Запустим код с  $r=0.001$ :

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression_reg(X_train, y_train, r=0.001)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

Код выводит следующее:

```
Validation: 0.460
```

Этот результат выглядит гораздо лучше по сравнению с предыдущим: 0,507.

## ПРИМЕЧАНИЕ

Иногда, когда добавление нового признака приводит к снижению производительности, для решения проблемы бывает достаточно просто удалить этот признак. Наличие проверочного набора данных важно для принятия решения о том, следует ли добавить регуляризацию, удалить признак или сделать и то и другое: мы используем оценку на основе проверочных данных, чтобы выбрать наилучший вариант. В нашем конкретном случае мы видим, что добавление регуляризации нам помогает: этот шаг улучшает оценку, которую мы получали ранее.

Мы попытались использовать  $r=0.001$ , но следует попробовать и другие значения. Протестируем несколько разных вариантов и выберем наилучший параметр  $r$ :

```
X_train = prepare_X(df_train)
X_val = prepare_X(df_val)
```

```
for r in [0.000001, 0.0001, 0.001, 0.01, 0.1, 1, 5, 10]:  
    w_0, w = train_linear_regression_reg(X_train, y_train, r=r)  
    y_pred = w_0 + X_val.dot(w)  
    print('%6s' %r, rmse(y_val, y_pred))
```

Мы видим, что наилучшая производительность достигается при меньшем  $r$ :

```
1e-06 0.460225  
0.0001 0.460225  
0.001 0.460226  
0.01 0.460239  
0.1 0.460370  
1 0.461829  
5 0.468407  
10 0.475724
```

Мы также можем заметить, что производительность при значениях ниже 0,1 меняется не так сильно, за исключением несущественного шестого знака.

Примем модель с  $r=0.01$  в качестве окончательной. Теперь мы можем проверить ее на тестовом наборе данных, чтобы убедиться, что она работает:

```
X_train = prepare_X(df_train)  
w_0, w = train_linear_regression_reg(X_train, y_train, r=0.01)  
  
X_val = prepare_X(df_val)  
y_pred = w_0 + X_val.dot(w)  
print('validation:', rmse(y_val, y_pred))  
  
X_test = prepare_X(df_test)  
y_pred = w_0 + X_test.dot(w)  
print('test:', rmse(y_test, y_pred))
```

Код выводит следующее:

```
validation: 0.460  
test: 0.457
```

Поскольку эти два числа довольно близки, мы приходим к выводу, что модель может хорошо обобщаться на новые неизвестные данные.

### Упражнение 2.4

Назовите причины, по которым регуляризация необходима.

- А. Она помогает контролировать веса модели и не позволяет им стать слишком большими.
- Б. Данные реального мира зашумлены.
- В. У нас часто возникают проблемы с неустойчивостью численного решения.

Возможны несколько ответов.

## 2.4.7. Использование модели

Поскольку теперь у нас есть модель, мы можем использовать ее для прогнозирования цены на автомобиль.

Предположим, что пользователь размещает на нашем сайте следующее объявление:

```
ad = {
    'city_mpg': 18,
    'driven_wheels': 'all_wheel_drive',
    'engine_cylinders': 6.0,
    'engine_fuel_type': 'regular_unleaded',
    'engine_hp': 268.0,
    'highway_mpg': 25,
    'make': 'toyota',
    'market_category': 'crossover,performance',
    'model': 'venza',
    'number_of_doors': 4.0,
    'popularity': 2031,
    'transmission_type': 'automatic',
    'vehicle_size': 'midsize',
    'vehicle_style': 'wagon',
    'year': 2013
}
```

Мы хотели бы предложить ему цену на этот автомобиль. Для этого мы используем нашу модель:

```
df_test = pd.DataFrame([ad])
X_test = prepare_X(df_test)
```

Сначала мы создаем небольшой датафрейм с единственной строкой. Эта строка содержит все значения словаря ad, который мы создали ранее. Далее мы преобразуем этот датафрейм в матрицу. Теперь мы можем применить нашу модель к матрице, чтобы получить прогноз цены этого автомобиля:

```
y_pred = w_0 + X_test.dot(w)
```

Однако данный прогноз не является окончательной ценой; это логарифм цены. Чтобы получить фактическую цену, нам нужно отменить логарифмирование, применив функцию экспоненты:

```
suggestion = np.expml(y_pred)
suggestion
```

Результат составляет 28 294,13. Реальная цена этого автомобиля — 31 120 долларов, так что наша модель недалеко ушла от фактической цены.

## 2.5. СЛЕДУЮЩИЕ ШАГИ

### 2.5.1. Упражнения

Вы можете попробовать выполнить следующие действия, чтобы еще больше улучшить модель.

- *Напишите функцию для двоичного кодирования.* В этой главе мы реализовали кодировку категорий вручную: просмотрели пять первых значений, занесли их в список, а затем прошлись по списку, чтобы создать двоичные признаки. Это слишком сложный подход, вследствие чего рекомендуется написать функцию, которая будет делать это автоматически. Она должна принимать несколько аргументов: датафрейм, имя категориальной переменной и количество наиболее частых значений, которые нужно учитывать. Эта функция также должна помочь нам и с предыдущим упражнением.
- *Чуть больше погрузитесь в конструирование признаков.* При реализации кодирования категорий мы включили только первые пять значений для каждой категориальной переменной. Включение большего количества значений в процесс кодирования вполне может улучшить модель. Попробуйте сделать это и переоцените качество модели с точки зрения RMSE.

### 2.5.2. Другие проекты

Есть и другие проекты, которыми вы можете заняться прямо сейчас.

- Спрогнозируйте цену дома. Вы можете взять набор открытых данных Airbnb в Нью-Йорке из <https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data> или набор данных о жилье в Калифорнии из [https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch\\_california\\_housing.html](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_california_housing.html).
- Попробуйте и другие наборы данных, такие как <https://archive.ics.uci.edu/ml/datasets.php?task=reg>, которые имеют числовые целевые значения. Например, мы можем использовать наборы данных успеваемости учащихся (<http://archive.ics.uci.edu/ml/datasets/Student+Performance>) для обучения модели определения успеваемости студентов.

## РЕЗЮМЕ

- Важно проводить простой первоначальный исследовательский анализ. Помимо прочего, это помогает выяснить, есть ли в данных недостающие значения. Невозможно обучить модель линейной регрессии, если значения

отсутствуют, поэтому важно проверить наши данные и при необходимости заполнить недостающие значения.

- В рамках исследовательского анализа данных нам необходимо проверить распределение целевой переменной. Если целевое распределение имеет длинный «хвост», то нам следует применить логарифмическое преобразование. Без этого мы можем получить от модели линейной регрессии неточные и вводящие в заблуждение прогнозы.
- Разделение данных на обучающие/проверочные/тестовые — лучший способ проверить наши модели. Он позволяет достоверно измерять производительность модели, а такие явления, как проблемы с неустойчивостью численного решения, не останутся без внимания.
- Модель линейной регрессии основана на простой математической формуле, и понимание этой формулы служит ключом к успешному применению модели. Знание этих деталей помогает нам понять, как работает модель, еще до кодирования.
- Используя Python и NumPy, достаточно легко реализовать линейную регрессию с нуля. Это помогает осознать, что в машинном обучении нет ничего волшебного: это простая математика, переведенная в код.
- RMSE предоставляет нам возможность измерить прогностическую производительность нашей модели на проверочном наборе. Это позволяет удостовериться, что модель работает хорошо, а также помогает сравнить несколько моделей и выбрать лучшую.
- Конструирование признаков — процесс создания новых признаков. Добавление позволяет повышать производительность модели. При добавлении новых признаков нам всегда нужно использовать проверочный набор, чтобы убедиться, что наша модель действительно улучшается. Без постоянного контроля мы рискуем получить посредственную или очень плохую производительность.
- Иногда мы сталкиваемся с проблемами неустойчивости численного решения, которые можем преодолеть с помощью регуляризации. Наличие хорошего способа проверки моделей имеет решающее значение для выявления проблемы до того, как станет слишком поздно.
- После того как модель обучена и проверена, мы можем использовать ее для прогнозов, например применив ее к автомобилям с неизвестными ценами, чтобы получить их возможную стоимость.

В главе 3 мы узнаем, как выполнять классификацию с помощью машинного обучения и используем логистическую регрессию для прогнозирования оттока клиентов.

## ОТВЕТЫ К УПРАЖНЕНИЯМ

- Упражнение 2.1. Ответ Б. Значения распределены далеко от головы.
- Упражнение 2.2. Ответ А.  $x_i$  — вектор признаков, а  $y_i$  — логарифм цены.
- Упражнение 2.3. Ответ Б. Вектор  $y$  с прогнозами цен.
- Упражнение 2.4. Ответы А, Б и В. Все три ответа верны.

# 3

## *Машинное обучение для классификации*

### **В этой главе**

- ✓ Выполнение исследовательского анализа данных для выявления важных признаков.
- ✓ Кодирование категориальных переменных для использования в моделях машинного обучения.
- ✓ Использование логистической регрессии для классификации.

В этой главе мы собираемся использовать машинное обучение для прогнозирования оттока.

*Отток* — это явление, когда клиенты перестают пользоваться услугами какой-либо компании. Таким образом, прогнозирование оттока заключается в выявлении клиентов, которые, скорее всего, в ближайшем будущем расторгнут свои контракты. Если компания может это делать, то может и предлагать скидки на эти услуги, чтобы удержать пользователей.

Конечно же, мы можем применить для этого машинное обучение: использовать прошлые данные о клиентах, которые уже ушли, и на их основе построить модель для выявления текущих клиентов, которые готовы уйти. Это задача бинарной классификации. Целевая переменная, которую мы хотим спрогнозировать, является категориальной и имеет только два возможных результата: уйдет или не уйдет.

В главе 1 мы узнали, что существует множество моделей контролируемого машинного обучения, а также специально упомянули те, которые можно использовать для бинарной классификации, включая логистическую регрессию, деревья решений и нейронные сети. В этой главе мы начнем с самой простой из них — логистической регрессии. Несмотря на свою явную простоту, она по-прежнему остается эффективной. Кроме того, эта модель имеет множество преимуществ перед другими: она быстрая и понятная, а результаты ее работы легко интерпретировать. Это своего рода рабочая лошадка машинного обучения и наиболее широко используемая модель во всей отрасли.

## 3.1. ПРОЕКТ ПО ПРОГНОЗИРОВАНИЮ ОТТОКА КЛИЕНТОВ

Проект, который я подготовил для этой главы, — прогнозирование оттока клиентов телекоммуникационной компании. Для его разработки мы используем логистическую регрессию и Scikit-learn.

Представим, что мы сотрудники компании, которая предлагает услуги телефонной связи и Интернета, и у нас есть проблема: наблюдается отток некоторых из наших клиентов. Они перестают пользоваться нашими услугами и переходят к конкурентам. Мы хотели бы пресечь это явление, поэтому разрабатываем систему для выявления таких клиентов, чтобы предложить им какой-либо стимул, который позволит им остаться. Мы хотим нацелить на них свои рекламные сообщения и предоставить скидку. Мы также хотели бы понять, почему именно модель считает, что наши клиенты вот-вот уйдут, а для этого нам нужно уметь интерпретировать ее прогнозы.

Мы собрали набор данных, в который внесли определенную информацию о наших клиентах: какими услугами они пользовались, сколько платили и как долго оставались с нами. Мы также знаем, кто расторг контракты и перестал пользоваться нашими услугами (в результате оттока). Будем использовать эту информацию в качестве целевой переменной в модели машинного обучения и прогнозировать, используя всю остальную доступную информацию.

План проекта представлен ниже.

1. Сначала мы загрузим набор данных и выполним некую первоначальную подготовку: переименуем столбцы и изменим значения внутри столбцов, чтобы они были согласованы по всему набору данных.
2. Затем мы разделим данные на обучающие, проверочные и тестовые, чтобы иметь возможность проверять наши модели.
3. В рамках первоначального анализа данных мы рассмотрим важность признаков, чтобы определить, какие из них важны в наших данных.

4. Преобразуем категориальные переменные в числовые и, таким образом, сможем использовать их в модели.
5. Наконец, обучим модель логистической регрессии.

В предыдущей главе мы все реализовывали самостоятельно, используя Python и NumPy. Однако в этом проекте уже начнем использовать Scikit-learn, библиотеку Python для машинного обучения. Если быть точными, то будем использовать ее для:

- разделения набора данных на обучающий и тестовый;
- кодирования категориальных переменных;
- обучения логистической регрессии.

### **3.1.1. Набор данных об оттоке телекоммуникационной компании**

Как и в предыдущей главе, мы будем использовать наборы данных из Kaggle. На этот раз мы получим данные из <https://www.kaggle.com/blastchar/telco-customer-churn>.

Согласно описанию, набор содержит следующую информацию:

- услуги клиентов — телефон; несколько линий; Интернет; техническая поддержка и дополнительные услуги, такие как онлайн-безопасность, резервное копирование, защита устройств и потоковое ТВ;
- информация об учетной записи — как долго оставались клиентами, тип контракта, способ оплаты;
- сборы — сколько клиент заплатил за последний месяц и в общей сложности;
- демографическая информация — пол, возраст, имеются ли иждивенцы или партнер;
- отток — да/нет, покинул ли клиент компанию в течение последнего месяца.

Сначала мы скачаем набор данных. Чтобы все было организовано должным образом, для начала создадим папку `chapter-03-churn-prediction`. Затем перейдем в нее и используем Kaggle CLI для скачивания данных:

```
kaggle datasets download -d blastchar/telco-customer-churn
```

После того как данные скачиваются, распакуем архив, чтобы получить оттуда CSV-файл:

```
unzip telco-customer-churn.zip
```

Теперь можно начинать.

### 3.1.2. Подготовка исходных данных

Первый шаг — создание нового блокнота в Jupyter. Если он еще не запущен — сделайте это:

```
jupyter notebook
```

Мы назовем блокнот `chapter-03-churn-project` (но вы можете назвать его, как вам нравится).

Как и ранее, мы начнем с добавления обычного импорта:

```
import pandas as pd
import numpy as np

import seaborn as sns
from matplotlib import pyplot as plt
%matplotlib inline
```

И теперь мы можем прочитать набор данных:

```
df = pd.read_csv('WA_Fn-UseC_-Telco-Customer-Churn.csv')
```

Мы используем функцию `read_csv` для считывания данных, а затем записываем результаты в датафрейм с именем `df`. Чтобы увидеть, сколько строк он содержит, воспользуемся функцией `len`:

```
len(df)
```

Она выводит число 7043, так что в этом наборе данных содержится 7043 строки. Набор данных невелик, но его должно быть достаточно для обучения годной модели.

Далее взглянем на первые пару строк с помощью `df.head()` (рис. 3.1). По умолчанию функция отображает первые пять строк датафрейма.

df.head()														
	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	... DeviceProtection	Tech Support	StreamingTV	StreamingMovies
0	7590-VHVEG	Female	0	Yes	No	1	No	No phone service	DSL	No	...	No	Yes	Yes
1	5575-GNVDE	Male	0	No	No	34	Yes	No	DSL	Yes	...	Yes	Yes	Yes
2	3668-QPYBK	Male	0	No	No	2	Yes	No	DSL	Yes	...	No	Yes	Yes
3	7795-CFOCW	Male	0	No	No	45	No	No phone service	DSL	Yes	...	Yes	Yes	Yes
4	9237-HQITU	Female	0	No	No	2	Yes	No	Fiber optic	No	...	No	Yes	Yes

**Рис. 3.1.** Выходные данные команды `df.head()`, показывающей первые пять строк набора данных об оттоке из телекоммуникационных компаний

В этом датафрейме довольно много столбцов, поэтому все они не помещаются на экране. Вместо этого мы можем транспонировать датафрейм с помощью функции `T`, поменяв столбцы и строки местами таким образом, чтобы столбцы (идентификатор пользователя, пол и т. д.) стали строками. Так мы можем видеть уже гораздо больше данных (рис. 3.2):

```
df.head().T
```

	0	1	2
<b>customerID</b>	7590-VHVEG	5575-GNVDE	3668-QPYBK
<b>gender</b>	Female	Male	Male
<b>SeniorCitizen</b>	0	0	0
<b>Partner</b>	Yes	No	No
<b>Dependents</b>	No	No	No
<b>tenure</b>	1	34	2
<b>PhoneService</b>	No	Yes	Yes
<b>MultipleLines</b>	No phone service	No	No
<b>InternetService</b>	DSL	DSL	DSL
<b>OnlineSecurity</b>	No	Yes	Yes
<b>OnlineBackup</b>	Yes	No	Yes
<b>DeviceProtection</b>	No	Yes	No
<b>TechSupport</b>	No	No	No
<b>StreamingTV</b>	No	No	No
<b>StreamingMovies</b>	No	No	No
<b>Contract</b>	Month-to-month	One year	Month-to-month
<b>PaperlessBilling</b>	Yes	No	Yes
<b>PaymentMethod</b>	Electronic check	Mailed check	Mailed check
<b>MonthlyCharges</b>	29.85	56.95	53.85
<b>TotalCharges</b>	29.85	1889.5	108.15
<b>Churn</b>	No	No	Yes

**Рис. 3.2.** Выходные данные `df.head().T`, показывающие первые три строки набора данных об оттоке из телекоммуникационной компании. Исходные строки отображаются в виде столбцов: таким образом можно увидеть больше данных, не используя прокрутку

Мы видим, что набор данных состоит из нескольких столбцов:

- **CustomerID** — идентификатор клиента;
- **Gender** — мужчина/женщина;

- **SeniorCitizen** — является ли клиент пожилым (0/1);
- **Partner** — проживает ли он с партнером (да/нет);
- **Dependents** — имеются ли у него иждивенцы (да/нет);
- **Tenure** — количество месяцев с момента начала действия контракта;
- **PhoneService** — есть ли у него телефонная служба (да/нет);
- **MultipleLines** — имеется ли у него несколько телефонных линий (да/нет/нет телефонной связи);
- **InternetService** — тип интернет-сервиса (нет/кабель/оптоволокно);
- **OnlineSecurity** — подключена ли онлайн-безопасность (да/нет/нет Интернета);
- **OnlineBackup** — подключен ли онлайн-сервис резервного копирования (да/нет/нет Интернета);
- **DeviceProtection** — подключен ли сервис защиты устройств (да/нет/нет Интернета);
- **TechSupport** — есть ли у клиента техническая поддержка (да/нет/нет Интернета);
- **StreamingTV** — подключен ли сервис потоковой передачи ТВ (да/нет/нет Интернета);
- **StreamingMovies** — подключен ли сервис потоковой передачи фильмов (да/нет/нет Интернета);
- **Contract** — тип контракта (ежемесячный/годовой/двулетний);
- **PaperlessBilling** — подключено ли электронное выставление счетов (да/нет);
- **PaymentMethod** — способ оплаты (электронный чек, чек по почте, банковский перевод, кредитная карта);
- **MonthlyCharges** — сумма, взимаемая ежемесячно (число);
- **TotalCharges** — общая сумма начислений (число);
- **Churn** — расторг ли клиент контракт (да/нет).

Самый интересный для нас пункт — это **Churn**. В качестве целевой переменной для нашей модели мы хотим научиться прогнозировать именно его. Он принимает два значения: да, если клиент расторг контракт, и нет, если он этого не сделал.

При чтении CSV-файла Pandas пытается автоматически определить правильный тип каждого столбца. Однако иногда это достаточно трудно сделать, и предлагаемые типы не совсем такие, какими мы их ожидаем. Вот почему важно проверять, верны ли фактические типы. Рассмотрим их, используя `df.dtypes`:

```
df.dtypes
```

Мы видим (рис. 3.3), что большинство типов определены правильно. Напомним, что `object` означает строковое значение, которое мы и ожидаем в большинстве столбцов. Однако можно сразу выделить два момента. Во-первых, `SeniorCitizen` определяется как `int64`, так что он имеет тип `integer`, а не `object`. Причина в том, что вместо значений `yes` и `no` (как у в других столбцах) здесь значения `1` и `0`, вследствие чего Pandas интерпретирует его как столбец с целыми числами. На самом деле это не проблема, поэтому нам не нужно как-либо дополнительного предварительно обрабатывать этот столбец.

df.dtypes	
customerID	object
gender	object
SeniorCitizen	int64
Partner	object
Dependents	object
tenure	int64
PhoneService	object
MultipleLines	object
InternetService	object
OnlineSecurity	object
OnlineBackup	object
DeviceProtection	object
TechSupport	object
StreamingTV	object
StreamingMovies	object
Contract	object
PaperlessBilling	object
PaymentMethod	object
MonthlyCharges	float64
TotalCharges	object
Churn	object
dtype:	object

SeniorCitizen имеет тип integer

TotalCharges не идентифицирован  
правильно как числовой тип (float или int)

**Рис. 3.3.** Автоматически определенные типы для всех столбцов датафрейма. `Object` означает строку. `TotalCharges` неправильно идентифицирован как `object`, тогда как он должен быть `float`

Еще один момент, на который следует обратить внимание, — это тип для `TotalCharges`. Ожидается, что данный столбец будет числовым: он содержит общую сумму, списанную с клиента, вследствие чего это должно быть число, а не строка. Тем не менее Pandas определяет его тип как `object`. Причина в том, что в некоторых случаях данный столбец содержит пробел, что позволяет представлять отсутствующее значение. Столкнувшись с нечисловыми символами, Pandas всегда определяет столбец `object`.

### ВАЖНО

Обращайте внимание на случаи, когда ожидаемый числовой тип Pandas заменяет на другой: скорее всего, столбец содержит специальное кодирование для пропущенных значений, которые требуют дополнительной предварительной обработки.

Мы можем сделать этот столбец числовым, преобразовав его в числа с помощью специальной функции в Pandas: `to_numeric`. По умолчанию она вызывает исключение, когда встречает нечисловые данные (например, пробелы), но мы можем заставить ее пропускать их, указав параметр `errors='coerce'`. Таким образом, Pandas заменит все нечисловые значения на `NaN` (not a number, не число):

```
total_charges = pd.to_numeric(df.TotalCharges, errors='coerce')
```

Чтобы подтвердить, что данные действительно содержат нечисловые символы, мы теперь можем использовать функцию `isnull()` столбца `total_charges` для получения всех строк, в которых Pandas не смог разобрать исходную строку:

```
df[total_charges.isnull()][['customerID', 'TotalCharges']]
```

Мы видим, что в столбце `TotalCharges` действительно встречаются пробелы (рис. 3.4).

customerID	TotalCharges
488	4472-LVYGI
753	3115-CZMZD
936	5709-LVOEQ
1082	4367-NUYAO
1340	1371-DWPAZ
3331	7644-OMVMY
3826	3213-VVOLG
4380	2520-SGTAA
5218	2923-ARZLG
6670	4075-WKNIU
6754	2775-SEFEE

```
total_charges = pd.to_numeric(df.TotalCharges, errors='coerce')
df[total_charges.isnull()][['customerID', 'TotalCharges']]
```

customerID	TotalCharges
488	4472-LVYGI
753	3115-CZMZD
936	5709-LVOEQ
1082	4367-NUYAO
1340	1371-DWPAZ
3331	7644-OMVMY
3826	3213-VVOLG
4380	2520-SGTAA
5218	2923-ARZLG
6670	4075-WKNIU
6754	2775-SEFEE

**Рис. 3.4.** Мы можем определить нечисловые данные в столбце, проанализировав их содержимое как числовое и увидев, в каких строках синтаксический анализ завершается ошибкой

Теперь нам предстоит решить, что же делать с этими недостающими значениями. Хотя сделать с ними можно очень многое, мы поступим с ними так же, как и в предыдущей главе, — установим недостающие значения равными нулю:

```
df.TotalCharges = pd.to_numeric(df.TotalCharges, errors='coerce')
df.TotalCharges = df.TotalCharges.fillna(0)
```

Кроме того, можно заметить, что имена столбцов не соответствуют единому соглашению об именовании. Некоторые из них начинаются со строчной буквы, в то время как другие начинаются с заглавной, а в значениях встречаются пробелы.

Приведем все к единообразному виду, понизив регистр и заменив пробелы символами подчеркивания. Так мы устраним все несоответствия в данных. Мы используем точно такой же код, как и в предыдущей главе:

```
df.columns = df.columns.str.lower().str.replace(' ', '_')

string_columns = list(df.dtypes[df.dtypes == 'object'].index)

for col in string_columns:
    df[col] = df[col].str.lower().str.replace(' ', '_')
```

Далее обратимся к нашей целевой переменной: `churn`. В настоящее время она категориальна и принимает два значения: yes и no (рис. 3.5, A). В случае двоичной классификации все модели обычно ожидают число: 0 для no и 1 для yes. Преобразуем все это в числа:

```
df.churn = (df.churn == 'yes').astype(int)
```

Используя `df.churn == 'yes'`, мы создаем серию Pandas типа `boolean`. Позиция в серии равна `True`, если в исходной серии она yes, и `False` в противном случае. Поскольку единственное другое значение, которое переменная может принять, — это no, команда преобразует yes в `True`, а no в `False` (рис. 3.5, B).

```
df.churn.head()
```

```
0      no
1      no
2    yes
3      no
4    yes
Name: churn, dtype: object
```

**A.** Исходный столбец «Churn» — это серия Pandas, которая содержит только значения «yes» и «no»

```
(df.churn == 'yes').head()
```

```
0    False
1    False
2     True
3    False
4     True
Name: churn, dtype: bool
```

**B.** Результат оператора `==`: это серия значений Boolean, где значение `True` устанавливается, когда элементы исходного ряда равны «yes», и `False` в противном случае

```
(df.churn == 'yes').astype(int).head()
```

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

**B.** Результат преобразования серии Boolean в целые числа: значение `True` преобразуется в 1, а значение `False` — в 0

**Рис. 3.5.** Выражение `(df.churn == 'yes').astype(int)` с разбивкой по шагам

Выполняя приведение с помощью функции `astype(int)`, мы преобразуем значение `True` в 1 и значение `False` в 0 (рис. 3.5, B). Это та же идея, которую

мы использовали в предыдущей главе, когда реализовывали кодирование категорий.

Мы уже провели небольшую предварительную обработку, так что отделим часть данных для тестирования. В предыдущей главе мы самостоятельно реализовали это в коде. Это позволяет понять, как все работает, но обычно мы не пишем такие вещи каждый раз с нуля. Вместо этого, как правило, используются существующие реализации из библиотек. В данной главе мы используем Scikit-learn, и в ней есть модуль под названием `model_selection`, который помогает разделить данные. Воспользуемся им.

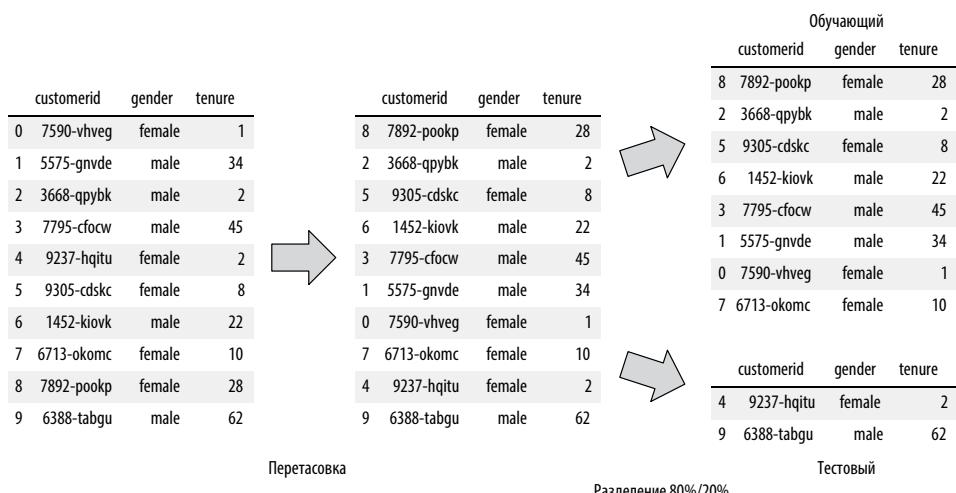
Функция, которую нам нужно импортировать из `model_selection`, называется `train_test_split`:

```
from sklearn.model_selection import train_test_split
```

Когда импортование завершено, она готова к использованию:

```
df_train_full, df_test = train_test_split(df, test_size=0.2, random_state=1)
```

Функция `train_test_split` принимает датафрейм `df` и создает два новых датафрейма: `df_train_full` и `df_test`. Это делается путем перетасовки исходного набора данных с последующим разделением его таким образом, чтобы тестовый набор содержал 20 % данных, а обучающий — оставшиеся 80 % (рис. 3.6). Внутренне это реализовано аналогично тому, что мы делали вручную в предыдущей главе.



**Рис. 3.6.** При использовании `train_test_split` исходный набор данных перетасовывается, а затем разделяется таким образом, что 80 % данных отправляется в обучающий набор, а оставшиеся 20 % — в тестовый

Эта функция содержит ряд параметров.

1. Первый передаваемый параметр — датафрейм, который требуется разделить: `df`.
2. Второй параметр — `test_size` — определяет размер набора данных, который мы хотим выделить для тестирования (20 % в нашем случае).
3. Третий параметр, который мы передаем, — `random_state`. Он необходим для того, чтобы каждый раз, когда мы запускаем данный код, датафрейм разделялся одинаково.

Перетасовка данных выполняется с помощью генератора случайных чисел. Важно зафиксировать начальное случайное значение, чтобы каждый раз конечное расположение строк оказывалось одинаковым.

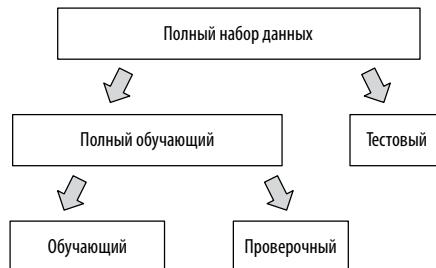
Мы действительно видим побочный эффект перетасовки: если мы посмотрим на датафреймы после разделения, например с помощью метода `head()`, то заметим, что индексы кажутся расставленными случайным образом (рис. 3.7).

```
df_train_full.head()
```

	customerid	gender	seniorcitizen	partner	dependents	tenure	phoneservice
1814	5442-pptjy	male	0	yes	yes	12	yes
5946	6261-rcvns	female	0	no	no	42	yes
3881	2176-osjuv	male	0	yes	no	71	yes
2389	6161-erdgd	male	0	yes	yes	71	yes
3676	2364-ufrom	male	0	no	no	30	yes

**Рис. 3.7.** Побочный эффект `train_test_split`: индексы (первый столбец) перемешиваются в новых датафреймах, поэтому вместо последовательных чисел, таких как 0, 1, 2..., они выглядят уже случайными

В предыдущей главе мы разделили данные на три части: обучающие, проверочные и тестовые. Однако `train_test_split` разбивает данные только на две:



**Рис. 3.8.** Поскольку `train_test_split` разбивает набор данных только на две части, то для получения трех частей мы выполняем разделение дважды. Сначала мы разделяем весь набор данных на полный обучающий и тестовый, а затем разделяем полный обучающий на обучающий и проверочный

обучающую и тестовую. Несмотря на это мы можем разделить исходный набор данных на три части — нужно просто снова разделить одну из частей (рис. 3.8).

Разделим датафрейм `df_train_full` еще раз, теперь уже на обучающий и проверочный наборы:

```
df_train, df_val = train_test_split(df_train_full, test_size=0.33,
                                     random_state=11) ← Устанавливает случайное начальное значение
y_train = df_train.churn.values | при выполнении разделения, что обеспечивает
y_val = df_val.churn.values | одинаковый результат при каждом запуске кода
del df_train['churn'] | Принимает столбец с целевой переменной churn
del df_val['churn'] | и сохраняет его за пределами датафрейма
                     | Удаляет столбцы churn из обоих датафреймов для
                     | гарантии того, что мы случайно не используем
                     | переменную churn в качестве признака при обучении
```

Теперь датафреймы подготовлены, и мы готовы использовать обучающий набор данных для первоначального исследовательского анализа данных.

### 3.1.3. Исследовательский анализ данных

Важно просмотреть данные перед обучением модели. Чем больше мы знаем о данных и их внутренних проблемах, тем лучше окажется модель, которую мы построим впоследствии.

Мы всегда должны проверять данные на отсутствующие значения, поскольку многие модели машинного обучения не могут легко справиться с отсутствием данных. Мы уже обнаружили проблему со столбцом `TotalCharges` и заменили недостающие значения нулями. Теперь посмотрим, нужно ли нам выполнять какую-либо дополнительную обработку `null`:

```
df_train_full.isnull().sum()
```

customerid	0
gender	0
seniorcitizen	0
partner	0
dependents	0
tenure	0
phoneservice	0
multiplelines	0
internetservice	0
onlinesecurity	0
onlinebackup	0
deviceprotection	0
techsupport	0
streamingtv	0
streamingmovies	0
contract	0
paperlessbilling	0
paymentmethod	0
monthlycharges	0
totalcharges	0
churn	0
dtype: int64	

**Рис. 3.9.** Нам не нужно обрабатывать отсутствующие значения в наборе данных: все значения во всех столбцах присутствуют

Команда выводит одни нули (рис. 3.9), вследствие чего мы видим, что у нас нет пропущенных значений в наборе данных и нам не придется делать что-то дополнительно.

Еще одно действие, которое мы должны сделать, — это проверить распределение значений в целевой переменной. Используем для этого метод `value_counts()`:

```
df_train_full.churn.value_counts()
```

Вывод выглядит так:

0	4113
1	1521

Первый столбец — значение целевой переменной, а второй — количество. Как мы видим, большинство клиентов не расторгали контракт.

Мы знаем абсолютные цифры, но проверим еще и долю ушедших пользователей среди всех клиентов. Для этого нам нужно разделить количество клиентов, которые ушли, на их общее количество. Мы знаем, что отток составил 1521 из 5634, поэтому пропорция такова:

$$1521 / 5634 = 0,27$$

Так мы получаем долю оттока пользователей, или вероятность того, что клиент расторгнет контракт. Как мы видим из обучающего набора данных, примерно 27 % клиентов перестали пользоваться нашими услугами.

Доля ушедших пользователей, или вероятность оттока, имеет специальное название: коэффициент оттока.

Еще один способ рассчитать коэффициент оттока — метод `mean()`. Он более удобен в использовании, нежели ручной расчет:

```
global_mean = df_train_full.churn.mean()
```

Используя этот метод, мы также получаем 0,27 (рис. 3.10).

```
global_mean = df_train_full.churn.mean()
round(global_mean, 3)
```

```
0.27
```

**Рис. 3.10.** Вычисление глобального коэффициента оттока в обучающем наборе данных

Причина, по которой он дает сходный результат, заключается в том, как мы вычисляем среднее значение. Если помните, эта формула выглядит так:

$$\frac{1}{n} \sum_{i=1}^n y_i,$$

где  $n$  — количество элементов в наборе данных.

Поскольку  $y_i$  может принимать только нули и единицы, суммировав их все, мы получаем количество единиц, или количество людей, которые ушли. Затем мы делим это значение на общее количество клиентов, что в точности совпадает с формулой, которую мы использовали для расчета коэффициента оттока ранее.

Наш набор данных с оттоком — пример так называемого *несбалансированного* набора данных. В нашем наборе оказалось в три раза больше людей, которые не участвовали в оттоке, чем тех, кто в нем участвовал. Следовательно, можно утверждать, что класс nonchurn доминирует над классом churn. Это вполне очевидно: коэффициент оттока в наших данных составляет 0,27, что является сильным показателем дисбаланса классов. Противоположностью *несбалансированному* набору выступает *сбалансированный*, когда положительные и отрицательные классы равномерно распределены между всеми наблюдениями.

### Упражнение 3.1

Чему равно среднее значение массива Boolean?

- A. Процент элементов `False` в массиве: количество элементов `False`, деленное на длину массива.
- B. Процент элементов `True` в массиве: количество элементов `True`, деленное на длину массива.
- B. Длина массива.

Как категориальные, так и числовые переменные в нашем наборе данных важны, но они различаются и требуют разного же подхода. Поэтому мы рассмотрим их по отдельности.

Создадим два списка:

- `categorical`, который будет содержать имена категориальных переменных;
- `numerical`, который аналогично будет содержать имена числовых переменных.

```
categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',
               'phoneservice', 'multiplelines', 'internetservice',
               'onlinesecurity', 'onlinebackup', 'deviceprotection',
               'techsupport', 'streamingtv', 'streamingmovies',
               'contract', 'paperlessbilling', 'paymentmethod']
numerical = ['tenure', 'monthlycharges', 'totalcharges']
```

Для начала мы можем узнать, сколько уникальных значений содержит каждая переменная. Мы уже знаем, что их должно быть всего несколько для каждого столбца, но проверим это:

```
df_train_full[categorical].nunique()
```

Действительно, мы видим, что большинство столбцов имеют два или три значения, а один (paymentmethod) — четыре (рис. 3.11). Это хорошо. Нам не придется тратить дополнительное время на подготовку и очистку данных; все уже готово к работе.

```
df_train_full[categorical].nunique()
```

gender	2
seniorcitizen	2
partner	2
dependents	2
phoneservice	2
multiplelines	3
internetservice	3
onlinesecurity	3
onlinebackup	3
deviceprotection	3
techsupport	3
streamingtv	3
streamingmovies	3
contract	3
paperlessbilling	2
paymentmethod	4
dtype: int64	

**Рис. 3.11.** Количество различных значений для каждой категориальной переменной. Мы видим, что все переменные имеют очень мало уникальных значений

Теперь мы переходим к другой важной части исследовательского анализа данных: пониманию того, какие признаки могут оказаться важными для нашей модели.

### 3.1.4. Важность признака

Знание того, как другие переменные влияют на целевую переменную (отток), — ключ к пониманию данных и построению хорошей модели. Этот процесс называется *анализом важности признаков*, и он часто выполняется как часть предварительного анализа данных, позволяя выяснить, какие переменные полезны для модели. Он также дает нам дополнительную информацию о наборе данных и помогает ответить на такие вопросы, как «Что вызывает отток клиентов?» и «Каковы характеристики людей, которые уходят?»

Существуют два разных вида признаков: категориальные и числовые. Измерять важность признаков можно с помощью разных способов, поэтому мы рассмотрим их по отдельности.

## Коэффициент оттока

Начнем с рассмотрения категориальных переменных. Первое, что мы можем сделать, — это взглянуть на коэффициент оттока для каждой переменной. Мы знаем, что категориальная переменная имеет набор значений, которые она может принимать, и каждое значение определяет группу внутри набора данных.

Мы можем посмотреть на все различные значения переменной. Далее, с каждой переменной сопоставлена группа клиентов: все клиенты, имеющие это значение. Для каждой такой группы мы можем вычислить коэффициент оттока, который является коэффициентом оттока группы. Когда он у нас будет, мы сможем сравнить его с глобальным коэффициентом оттока, рассчитанным для всех наблюдений сразу.

Если разница между ними невелика, то значение не очень важно при прогнозировании оттока, поскольку эта группа клиентов в действительности не отличается от остальных клиентов. С другой стороны, если разница ощутима, то что-то внутри этой группы отличает ее от остальных. Алгоритм машинного обучения должен быть способен улавливать это и использовать при составлении прогнозов.

Сначала проверим переменную `gender`. Она может принимать два значения: `female` и `male`. У нас две группы клиентов: те, у которых `gender == 'female'`, и те, у которых `gender == 'male'` (рис. 3.12).

The diagram illustrates the splitting of a data frame into two groups: one for females and one for males. On the left, there is a large data frame with columns `customerid`, `gender`, and `churn`. The rows are numbered from 0 to 9. An arrow points from this main frame to two smaller frames on the right. The top right frame is titled "gender == 'female'" and contains rows 0, 4, 5, 7, and 8. The bottom right frame is titled "gender == 'male'" and contains rows 1, 2, 3, 6, and 9.

				gender == "female"			
				customerid	gender	churn	
0	7590-vhveg	female	0	0	7590-vhveg	female	0
1	5575-gnvde	male	0	4	9237-hqitu	female	1
2	3668-qpybk	male	1	5	9305-cdskc	female	1
3	7795-cfocw	male	0	7	6713-okomc	female	0
4	9237-hqitu	female	1	8	7892-pookp	female	1
5	9305-cdskc	female	1				
6	1452-kiovk	male	0				
7	6713-okomc	female	0				
8	7892-pookp	female	1				
9	6388-tabgu	male	0				

				customerid	gender	churn	
1	5575-gnvde	male	0	1	5575-gnvde	male	0
2	3668-qpybk	male	1	2	3668-qpybk	male	1
3	7795-cfocw	male	0	3	7795-cfocw	male	0
6	1452-kiovk	male	0	6	1452-kiovk	male	0
9	6388-tabgu	male	0	9	6388-tabgu	male	0

gender == "male"

**Рис. 3.12.** Датафрейм разделен значениями переменной `gender` на две группы: группа с `gender == "female"` и группа с `gender == "male"`

Чтобы вычислить коэффициент оттока для всех клиентов женского пола, мы сначала выбираем только те строки, которые соответствуют `gender == "female"`, а затем вычисляем для них коэффициент оттока:

```
female_mean = df_train_full[df_train_full.gender == 'female'].churn.mean()
```

Затем мы проделываем то же самое для всех клиентов мужского пола:

```
male_mean = df_train_full[df_train_full.gender == 'male'].churn.mean()
```

Выполнив этот код и проверив результаты, мы увидим, что уровень оттока клиентов-женщин составляет 27,7 %, а клиентов-мужчин — 26,3 %, тогда как глобальный уровень оттока составляет 27 % (рис. 3.13). Разница между групповыми коэффициентами женщин и мужчин довольно мала, и это указывает на то, что знание пола клиента не поможет нам определить, расторгнет ли он договор.

```
global_mean = df_train_full.churn.mean()
round(global_mean, 3)
```

0.27

```
female_mean = df_train_full[df_train_full.gender == 'female'].churn.mean()
print('gender == female:', round(female_mean, 3))

male_mean = df_train_full[df_train_full.gender == 'male'].churn.mean()
print('gender == male: ', round(male_mean, 3))
```

```
gender == female: 0.277
gender == male: 0.263
```

**Рис. 3.13.** Глобальный уровень оттока по сравнению с оттоком среди мужчин и женщин. Цифры довольно похожи, и это означает, что `gender` не является полезной переменной при прогнозировании оттока

Теперь взглянем на другую переменную: `partner`. Он принимает значения `yes` и `no`, поэтому есть две группы клиентов: те, для которых `partner == 'yes'`, и те, для которых `partner == 'no'`.

Мы можем проверить их групповые коэффициенты оттока, используя тот же код. Нам нужно изменить лишь условия фильтрации:

```
partner_yes = df_train_full[df_train_full.partner == 'yes'].churn.mean()
partner_no = df_train_full[df_train_full.partner == 'no'].churn.mean()
```

Как мы видим, коэффициенты для тех, у кого имеется партнер, сильно отличаются от показателей тех, у кого его нет: 20 и 33 % соответственно. Это означает, что клиенты, у которых нет партнера, имеют больше шансов расторгнуть договор, чем те, у кого он есть (рис. 3.14).

```

partner_yes = df_train_full[df_train_full.partner == 'yes'].churn.mean()
print('partner == yes:', round(partner_yes, 3))

partner_no = df_train_full[df_train_full.partner == 'no'].churn.mean()
print('partner == no :', round(partner_no, 3))

partner == yes: 0.205
partner == no : 0.33

```

**Рис. 3.14.** Коэффициент оттока среди людей с партнером значительно ниже, чем среди людей без партнера, — 20,5 против 33 %, что указывает на то, что переменная partner полезна для прогнозирования оттока

## Коэффициент риска

В дополнение к рассмотрению разницы между групповым и глобальным коэффициентом, интересно взглянуть и на соотношение между ними. В статистике соотношение между вероятностями в разных группах называется *коэффициентом риска*, где *риск* относится к риску возникновения эффекта. В нашем случае эффект — это отток, так что риск оттока выглядит так:

$$\text{риск} = \text{групповой коэффициент} / \text{глобальный коэффициент}$$

Например, для `gender == female` риск оттока составляет 1,02:

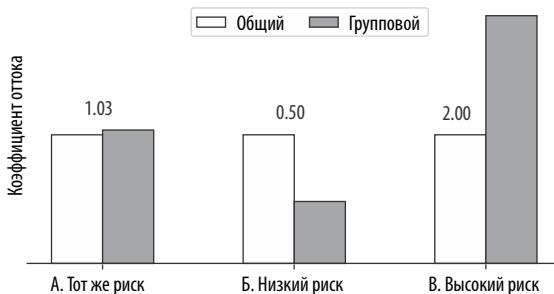
$$\text{риск} = 27,7 \% / 27 \% = 1,02$$

Риск — это число от нуля до бесконечности. У него есть полезная интерпретация, которая показывает вероятность того, насколько элементы группы испытывают эффект (отток) по сравнению со всей совокупностью элементов.

Если разница между групповым и общим показателем невелика, риск близок к 1: эта группа имеет тот же уровень риска, что и остальная часть. Клиенты в данной группе настолько же склонны к оттоку, насколько и все остальные. Другими словами, группа с риском, близким к 1, вообще не является группой риска (рис. 3.15, группа А).

Если риск ниже 1, то группа имеет более низкие риски: уровень оттока в этой группе меньше, чем глобальный отток. Например, значение 0,5 означает, что вероятность оттока клиентов в этой группе в два раза ниже, чем у клиентов в целом (рис. 3.15, группа Б).

С другой стороны, если значение выше 1, это группа риска: в ней ожидается больший отток, чем среди общего числа. Таким образом, риск, равный 2, означает, что вероятность оттока клиентов из группы в два раза выше (рис. 3.15, группа В).



**Рис. 3.15.** Коэффициент оттока различных групп в сравнении с глобальным коэффициентом оттока. В группе А показатели примерно одинаковы, поэтому риск оттока составляет около 1. В группе Б коэффициент оттока группы меньше общего показателя, поэтому риск составляет около 0,5. Наконец в группе В коэффициент оттока группы выше, чем общий показатель, поэтому риск близок к 2

Термин *risk* первоначально появился в результате контролируемых исследований, в которых одной группе пациентов назначается лечение (лекарство), а другой группе — нет (только плацебо). Затем мы сравниваем, насколько эффективно лекарство, рассчитав частоту отрицательных исходов в каждой группе, а затем соотношение между показателями:

$$\text{риск} = \frac{\text{процент отрицательных результатов в группе 1}}{\text{процент отрицательных результатов в группе 2}}$$

Если лекарство оказывается эффективным, то говорят, что оно снижает риск негативного исхода, и значение риска составляет менее 1.

Подсчитаем риски для переменных *gender* и *partner*. Что касается *gender*, то риск как для мужчин, так и для женщин составляет около 1, поскольку показатели в обеих группах существенно не отличаются от общего показателя. Неудивительно, что для переменной *partner* все оказывается по-другому; отсутствие партнера сопряжено с большим риском (табл. 3.1).

**Таблица 3.1.** Показатели оттока и риски для переменных *gender* и *partner*. Коэффициенты оттока среди женщин и мужчин существенно не отличаются от общих, поэтому риски оттока для них невелики: обе имеют значения рисков в районе 1. С другой стороны, уровень оттока людей, у которых нет партнера, значительно выше среднего, что делает их группой риска со значением риска 1,22. Люди с партнерами, как правило, менее подвержены оттоку, поэтому для них риск составляет всего 0,75

Переменная	Значение	Коэффициент оттока	Риск
<i>gender</i>	Female	27,7 %	1,02
	Male	26,3 %	0,97
<i>partner</i>	Да	20,5 %	0,75
	Нет	33 %	1,22

Пока мы использовали только две переменные. Проделаем это для всех категориальных переменных. Для этого нам понадобится фрагмент кода, который проверяет все значения, которые принимает переменная, и вычисляет коэффициент оттока для каждого из них.

Если бы мы использовали SQL, то все было бы довольно просто. Что касается пола, нам пришлось бы написать что-то наподобие этого:

```
SELECT
    gender, AVG(churn),
    AVG(churn) - global_churn,
    AVG(churn) / global_churn
FROM
    data
GROUP BY
    gender
```

Вот приблизительный перевод на Pandas:

```
global_mean = df_train_full.churn.mean()           ← ❶ Вычисляет AVG(churn)
df_group = df_train_full.groupby(by='gender').churn.agg(['mean'])
df_group['diff'] = df_group['mean'] - global_mean ← ❷ Вычисляет разницу между
df_group['risk'] = df_group['mean'] / global_mean   ← ❸ Вычисляет риск оттока
                                                       ❸ и глобальным коэффициентом
df_group
```

В ❶ мы вычисляем часть `AVG(churn)`. Для этого мы используем функцию `agg`, тем самым указывая, что нам нужно объединить данные в одно значение для каждой группы: среднее значение. В ❷ мы создаем еще один столбец, `diff`, где сохраним разницу между групповым средним и общим средним. Аналогично в ❸ мы создаем столбец `risk`, где вычисляем отношение между групповым и глобальным средним.

Результаты показаны на рис. 3.16.

	mean	diff	risk
gender			
female	0.276824	0.006856	1.025396
male	0.263214	-0.006755	0.974980

**Рис. 3.16.** Коэффициент оттока для переменной `gender`. Мы видим, что для обоих значений разница между коэффициентом группового оттока и глобальным коэффициентом оттока не слишком велика

Проделаем это для всех категориальных переменных. Мы можем перебрать их и применить для каждой один и тот же код:

```

from IPython.display import display
for col in categorical: ← Циклы по всем
    df_group = df_train_full.groupby(by=col).churn.agg(['mean']) ← Выполняет
    df_group['diff'] = df_group['mean'] - global_mean groupby
    df_group['rate'] = df_group['mean'] / global_mean для каждой
    display(df_group) ← Отображает итоговый категорииальной
                           датафрейм переменной

```

Этот код имеет ряд различий. Так, вместо того чтобы вручную указывать имя столбца, мы перебираем все категориальные переменные.

Второе различие более тонкое: нам нужно вызвать функцию `display`, чтобы отобразить датафрейм внутри цикла. Способ, которым мы обычно отображаем датафрейм, заключается в том, чтобы оставить его в качестве последней строки в ячейке Jupyter Notebook, а затем выполнить ячейку. Если делать это таким образом, то датафрейм будет отображаться как вывод ячейки. Именно так нам удалось посмотреть на содержимое датафрейма в начале главы (рис. 3.1, см. выше). Однако мы не можем делать так внутри цикла. Чтобы по-прежнему иметь возможность видеть содержимое датафрейма, мы вызываем функцию `display` явно.

Из результатов (рис. 3.17) мы узнаем следующую информацию:

- что касается пола, то между женщинами и мужчинами нет большой разницы. Оба средних примерно одинаковы, и для обеих групп риски близки к 1;
- пожилые люди склонны к оттоку больше, чем те, кто не является пенсионером: риск оттока составляет 1,53 для пожилых людей и 0,89 для остальных;
- у людей, имеющих партнера, отток меньше, чем у людей без партнера. Риски составляют 0,75 и 1,22 соответственно;
- люди, пользующиеся телефонной связью, не подвержены риску оттока: риск близок к 1, и почти нет разницы в сравнении с общим уровнем оттока. Люди, которые не пользуются телефонной связью, еще менее склонны к оттоку: риск ниже 1, а разница с общим показателем оттока отрицательна.

	mean	diff	risk		mean	diff	risk	
<b>gender</b>								
female	0.276824	0.006856	1.025396		0	0.242270	-0.027698	0.897403
male	0.263214	-0.006755	0.974980		1	0.413377	0.143409	1.531208
<b>А. Коэффициент оттока и риск: gender</b>								
	mean	diff	risk		mean	diff	risk	
<b>partner</b>								
no	0.329809	0.059841	1.221659		no	0.241316	-0.028652	0.893870
yes	0.205033	-0.064935	0.759472		yes	0.273049	0.003081	1.011412
<b>Б. Коэффициент оттока и риск: seniorcitizen</b>								
	mean	diff	risk		mean	diff	risk	
<b>phoneservice</b>								
no	0.241316	-0.028652	0.893870					
yes	0.273049	0.003081	1.011412					
<b>Г. Коэффициент оттока и риск: phoneservice</b>								

**Рис. 3.17.** Разница в уровне оттока и риск для четырех категориальных переменных: `gender`, `seniorcitizen`, `partner` и `phoneservice`

Некоторые переменные показывают довольно существенные различия (рис. 3.18):

- клиенты, не имеющие технической поддержки, как правило, уходят чаще, чем те, у кого она есть;
- люди, с которыми заключен ежемесячный контракт, расторгают его намного чаще, чем остальные, а люди с двухлетними контрактами уходят очень редко.

	mean	diff	risk		mean	diff	risk
techsupport				contract			
no	0.418914	0.148946	1.551717	month-to-month	0.431701	0.161733	1.599082
no_internet_service	0.077805	-0.192163	0.288201	one_year	0.120573	-0.149395	0.446621
yes	0.159926	-0.110042	0.592390	two_year	0.028274	-0.241694	0.104730

А. Коэффициент оттока и риск: techsupport

Б. Коэффициент оттока и риск: contract

**Рис. 3.18.** Разница между уровнем оттока в группе и общим уровнем оттока для techsupport и contract. Люди, не имеющие технической поддержки, с которыми заключен ежемесячный контракт, как правило, уходят гораздо чаще, чем клиенты из других групп, в то время как люди, имеющие техническую поддержку и двухлетние контракты, остаются клиентами с очень низким уровнем риска

Таким образом, просто взглянув на разницы и риски, мы можем определить наиболее отличительные признаки — те, которые будут полезны для выявления оттока. Таким образом, можно ожидать, что эти признаки окажутся полезными для наших будущих моделей.

## Взаимная информация

Различия, которые мы только что рассмотрели, полезны в нашем анализе и важны для понимания данных, но их трудно использовать для определения того, какой признак наиболее важен и является ли переменная технической поддержки более полезной, чем тип контракта.

К счастью, нам на помощь приходят показатели важности: мы можем измерить степень зависимости между категориальной и целевой переменной. Если две переменные являются зависимыми, то знание значения одной переменной дает нам некоторую информацию о другой. С другой стороны, если переменная полностью независима от целевой переменной, то является бесполезной и может быть безопасно удалена из набора данных.

В нашем случае знание о том, что у клиента заключен ежемесячный контракт, может указывать на то, что данный клиент скорее уйдет, чем останется.

## ВАЖНО

Клиенты с ежемесячными контрактами, как правило, гораздо более склонны к расторжению контракта, чем клиенты с другими видами контрактов. Это именно та взаимосвязь,

которую мы хотим найти в наших данных. Без таких взаимосвязей в данных модели машинного обучения работать не будут — они не смогут делать прогнозы. Чем выше степень зависимости, тем полезнее признак.

Для категориальных переменных одним из таких показателей служит взаимная информация, которая показывает, сколько информации мы узнаем об одной переменной, если знаем значение другой. Это концепция из теории информации, и в машинном обучении мы часто используем ее для измерения взаимозависимости двух переменных.

Более высокие значения взаимной информации означают более высокую степень зависимости: если взаимная информация между категориальной и целевой переменными высока, то с помощью категориальной можно будет спрогнозировать целевую. С другой стороны, если взаимная информация невелика, то категориальная переменная и цель независимы, и, следовательно, переменная не будет полезна для прогнозирования цели.

Взаимная информация уже реализована в Scikit-learn в функции `mutual_info_score` из пакета `metrics`, поэтому мы можем просто ее использовать:

```
from sklearn.metrics import mutual_info_score
❶ Создает автономную функцию
для вычисления взаимной информации
def calculate_mi(series):
    return mutual_info_score(series, df_train_full.churn)
❷ Использует
функцию mutual_
info_score из Scikit-
learn
df_mi = df_train_full[categorical].apply(calculate_mi)
❸ Применяет функцию из ❶ к каждому
категориальному столбцу набора данных
df_mi = df_mi.sort_values(ascending=False).to_frame(name='MI')
❹ Сортирует значения результата
```

В ❸ мы используем метод `apply` для применения функции `calculate_mi` (которую определили в ❶) к каждому столбцу датафрейма `df_train_full`. Поскольку мы включаем дополнительный шаг выбора лишь категориальных переменных, он применяется только к ним. Функция, которую мы определяем в ❶, принимает лишь один параметр: `series`. Это столбец из датафрейма, для которого мы вызвали метод `apply()`. В ❸ мы вычисляем коэффициент взаимной информации между серией и целевой переменной `churn`. Выходные данные представляют собой одно число, поэтому выходные данные метода `apply()` представляют собой серию Pandas. Наконец, мы сортируем элементы серии по коэффициенту взаимной информации и преобразуем серию в датафрейм. Таким образом, результат хорошо отображается в Jupyter.

Как мы видим, `contract`, `onlinesecurity` и `techsupport` относятся к числу наиболее важных признаков (рис. 3.19). И действительно, мы уже отмечали, что `contract` и `techsupport` довольно информативны. Неудивительно и то, что `gender` является одним из наименее важных признаков, поэтому не стоит ожидать, что он окажется полезным для модели.

MI	
<b>contract</b>	0.098320
<b>onlinesecurity</b>	0.063085
<b>techsupport</b>	0.061032
<b>internetservice</b>	0.055868
<b>onlinebackup</b>	0.046923
<b>partner</b>	0.009968
<b>seniorcitizen</b>	0.009410
<b>multiplelines</b>	0.000857
<b>phoneservice</b>	0.000229
<b>gender</b>	0.000117

А. Наиболее полезные признаки в соответствии с коэффициентом взаимной информации

Б. Наименее полезные признаки в соответствии с коэффициентом взаимной информации

**Рис. 3.19.** Взаимная информация между категориальными переменными и целевой переменной. Чем выше значения, тем лучше. Таким образом, *contract* является наиболее полезной переменной, тогда как *gender* — наименее полезной

## Коэффициент корреляции

Взаимная информация — способ количественной оценки степени зависимости между двумя категориальными переменными, но он не сработает, когда один из признаков числовой, поэтому мы не сможем применить его к трем числовым переменным, которые у нас есть.

Однако мы можем измерить зависимость между двоичной целевой переменной и числовой переменной. Мы можем притвориться, что двоичная переменная является числовой (содержащей только числа 0 и 1), а затем использовать классические методы статистики, чтобы проверить наличие какой-либо зависимости между этими переменными.

Одним из таких методов служит коэффициент *корреляции* (иногда называемый *коэффициентом корреляции Пирсона*). Это значение от  $-1$  до  $1$ :

- положительная корреляция означает, что когда одна переменная увеличивается, другая также имеет тенденцию к увеличению. В случае двоичной цели, когда значения переменной высокие, мы наблюдаем единицы чаще, чем нули. Но когда значения переменной невелики, нули встречаются чаще, чем единицы;
- нулевая корреляция означает отсутствие связи между двумя переменными: они полностью независимы;
- отрицательная корреляция возникает, когда одна переменная увеличивается, а другая уменьшается. В случае двоичной переменной при высоких значениях мы наблюдаем больше нулей, чем единиц, в целевой переменной. Когда значения низкие, мы видим больше единиц.

Рассчитать коэффициент корреляции в Pandas очень легко:

```
df_train_full[numerical].corrwith(df_train_full.churn)
```

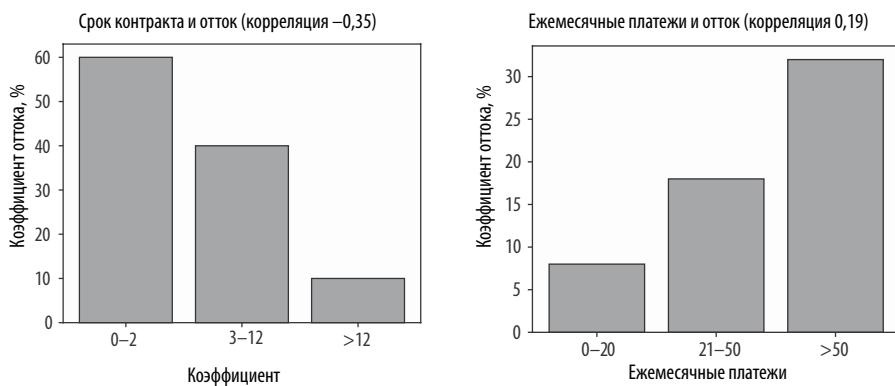
Результаты показаны на рис. 3.20.

- Корреляция между `tenure` и оттоком составляет  $-0,35$ : она имеет отрицательное значение, поэтому чем дольше клиенты остаются клиентами, тем менее они склонны к оттоку. Для клиентов, остающихся в компании в течение двух месяцев или менее, коэффициент оттока составляет  $60\%$ ; для клиентов со сроком пребывания от 3 до 12 месяцев —  $40\%$ ; а для клиентов, остающихся дольше года, —  $17\%$ . Таким образом, чем дольше действует контракт, тем меньше коэффициент оттока (рис. 3.21, А).

correlation	
<code>tenure</code>	$-0.351885$
<code>monthlycharges</code>	$0.196805$
<code>totalcharges</code>	$-0.196353$

**Рис. 3.20.** Корреляция между числовыми переменными и оттоком. Переменная `tenure` имеет высокую отрицательную корреляцию: по мере роста срока контракта уровень оттока снижается. `monthlycharges` имеет положительную корреляцию: чем больше клиенты платят, тем выше вероятность их оттока

- Переменная `monthlycharges` имеет положительный коэффициент  $0,19$ ; это значит, клиенты, которые платят больше, как правило, уходят чаще. Только  $8\%$  из тех, кто платит менее  $20$  долларов в месяц, уходят. Клиенты, платящие от  $21$  до  $50$  долларов, уходят чаще, их коэффициент оттока составляет  $18\%$ . А вот среди людей, платящих более  $50$  долларов, отток составляет  $32\%$  (рис. 3.21, Б).



А. Коэффициент оттока при различных значениях `tenure`. Коэффициент корреляции отрицательный, поэтому тенденция нисходящая: при более высоких значениях `tenure` коэффициент оттока меньше

Б. Коэффициент оттока для различных значений `monthlycharges`. Коэффициент корреляции положительный, поэтому тенденция восходящая: при более высоких значениях `monthlycharges` коэффициент оттока выше

**Рис. 3.21.** Коэффициент оттока для переменной `tenure` (отрицательная корреляция  $-0,35$ ) и `monthlycharges` (положительная корреляция  $0,19$ )

- Переменная `totalcharges` имеет отрицательную корреляцию, что вполне объяснимо: чем дольше люди остаются в компании, тем больше они заплатили в общей сложности, поэтому вероятность того, что они уйдут, меньше. В этом случае мы ожидаем картину, аналогичную случаю с переменной `tenure`. При малых значениях коэффициент оттока высок; при больших значениях он ниже.

Проведя первоначальный исследовательский анализ данных, определив важные функции и получив некоторое представление о задаче, мы готовы перейти к следующему шагу: конструированию признаков и обучению модели.

## 3.2. КОНСТРУИРОВАНИЕ ПРИЗНАКОВ

Сначала мы изучили данные и определили, что может оказаться полезным для модели. После этого у нас сложилось четкое понимание того, как те или иные переменные влияют на отток — нашу цель.

Однако прежде чем приступить к обучению, нам нужно выполнить этап конструирования признаков: преобразовать все категориальные переменные в числовые признаки. Мы сделаем это в следующем подразделе и после этого будем готовы обучать модель логистической регрессии.

### 3.2.1. Прямое кодирование для категориальных переменных

Как мы уже выяснили в главе 1, мы не можем просто взять категориальную переменную и поместить ее в модель машинного обучения. Модели могут работать только с числами в матрицах. Следовательно, нам требуется преобразовать наши категориальные данные в матричную форму или закодировать.

Одним из таких методов кодирования является *прямое кодирование*. Мы уже сталкивались с этим методом в предыдущей главе при создании признаков для марки автомобиля и других категориальных переменных. Там мы упомянули об этом лишь вкратце и использовали очень простым способом. В текущей главе мы уделим ему больше времени.

Если переменная `contract` имеет возможные значения (`monthly`, `yearly` и `two-year`), то мы можем представить клиента с годовым контрактом как  $(0, 1, 0)$ . В этом случае значение `yearly` является активным, или *горячим*, поэтому оно равно 1, тогда как остальные значения являются неактивными, или *холодными*, поэтому они равны 0.

Чтобы лучше понять, рассмотрим случай с двумя категориальными переменными и посмотрим, как мы создаем из них матрицу. Это переменные:

- `gender` со значениями `female` и `male`;
- `contract` со значениями `monthly`, `yearly` и `two-year`.

Поскольку переменная `gender` имеет только два возможных значения, мы создаем два столбца в итоговой матрице. Переменная `contract` имеет три столбца, и в общей сложности наша новая матрица будет содержать пять столбцов:

- `gender=female`;
- `gender=males`;
- `contract=monthly`;
- `contract=yearly`;
- `contract=two-year`.

Рассмотрим двух клиентов (рис. 3.22):

- клиентку с годовым контрактом;
- клиента с ежемесячным контрактом.

gender	contract
male	monthly
female	yearly

gender		contract		
female	male	monthly	yearly	two-year
0	1	1	0	0
1	0	0	1	0

**Рис. 3.22.** Исходный набор данных с категориальными переменными находится слева, а представление после прямого кодирования — справа. Для первого клиента столбцы `gender=males` и `contract=monthly` являются «горячими», поэтому получают 1. Для второго клиента «горячими» столбцами будут уже `gender=female` и `contract=yearly`

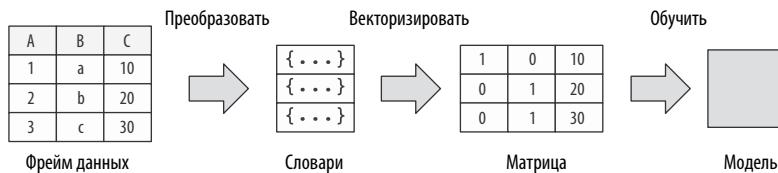
Для первого клиента переменная `gender` кодируется путем ввода 1 в столбец `gender=female` и 0 в столбец `gender=males`. Аналогично `contract=yearly` получает значение 1, тогда как остальные столбцы (`contract=monthly` и `contract=two-year`) получают значение 0.

Что касается второго клиента, то `gender=males` и `contract=monthly` получают единицы, а остальные столбцы — нули (см. рис. 3.22). Способ, которым мы реализовали это ранее, был простым, но довольно ограниченным. Сначала мы просмотрели пять первых значений переменной, после чего прошлись по каждому значению и вручную создали столбец в датафрейме. Однако когда количество признаков растет, этот процесс становится излишне трудоемким.

К счастью, обычно нам не приходится реализовывать это вручную: мы можем использовать Scikit-learn. Выполнить однократное кодирование в Scikit-learn можно несколькими способами, но мы будем использовать `DictVectorizer`.

Как следует из названия, `DictVectorizer` берет словарь и *векторизирует* его, то есть создает из него векторы. Затем векторы складываются в виде строк одной

матрицы. Эта матрица и используется в качестве входных данных для алгоритма машинного обучения (рис. 3.23).



**Рис. 3.23.** Процесс создания модели. Сначала мы преобразуем датафрейм в список словарей, затем векторизуем список в матрицу и, наконец, используем ее для обучения модели

Чтобы использовать этот метод, нам потребуется преобразовать наш датафрейм в список словарей, что просто сделать в Pandas, используя `to_dict` с параметром `orient='records'`:

```
train_dict = df_train[categorical + numerical].to_dict(orient='records')
```

Если мы взглянем на первый элемент этого нового списка, то увидим следующее:

```
{'gender': 'male',
'seniorcitizen': 0,
'partner': 'yes',
'dependents': 'yes',
'phoneservice': 'yes',
'multiplelines': 'no',
'internetservice': 'no',
'onlinesecurity': 'no_internet_service',
'onlinebackup': 'no_internet_service',
'deviceprotection': 'no_internet_service',
'techsupport': 'no_internet_service',
'streamingtv': 'no_internet_service',
'streamingmovies': 'no_internet_service',
'contract': 'two_year',
'paperlessbilling': 'no',
'paymentmethod': 'mailed_check',
'tenure': 12,
'monthlycharges': 19.7,
'totalcharges': 258.35}
```

Каждый столбец из датафрейма является ключом в этом словаре, а значения берутся из фактических значений строк датафрейма.

Теперь мы можем использовать `DictVectorizer`. Мы создаем его, а затем используем со списком словарей, который создали ранее:

```
from sklearn.feature_extraction import DictVectorizer
```

```
dv = DictVectorizer(sparse=False)
dv.fit(train_dict)
```

В этом коде мы создаем экземпляр `DictVectorizer`, который называем `dv`, и «обучаем» его, вызывая метод `fit`. Данный метод просматривает содержимое этих словарей и вычисляет возможные значения для каждой переменной и то, как сопоставить их со столбцами в выходной матрице. Если объект является категориальным, то применяется схема прямого кодирования, но если объект числовой, то он остается нетронутым.

Класс `DictVectorizer` может принимать набор параметров. Мы указываем один из них: `sparse=False`. Этот параметр означает, что созданная матрица не будет разреженной и вместо такой матрицы будет создан простой массив NumPy. Если вы не знаете о разреженных матрицах, то не волнуйтесь: в этой главе они нам не нужны.

После того как мы настроим векторизацию, мы можем преобразовать словари в матрицу с помощью метода `transform`:

```
X_train = dv.transform(train_dict)
```

Эта операция создает матрицу с 45 столбцами. Взглянем на первую строку, которая соответствует клиенту, которого мы рассматривали ранее:

```
X_train[0]
```

Поместив этот код в ячейку Jupyter Notebook и выполнив его, мы получаем следующий вывод:

```
array([ 0. ,  0. ,  1. ,  1. ,  0. ,  0. ,  0. ,  0. ,  1. ,
       0. ,  1. ,  1. ,  0. ,  0. ,  86.1,  1. ,  0. ,
       0. ,  0. ,  0. ,  1. ,  0. ,  0. ,  1. ,  0. ,
      1. ,  0. ,  1. ,  1. ,  0. ,  0. ,  0. ,  0. ,
      1. ,  0. ,  0. ,  0. ,  1. ,  0. ,  0. ,  1. ,
      0. ,  0. ,  1. ,  71. ,  6045.9])
```

Как видим, большинство элементов представляют собой единицы и нули — это категориальные переменные после прямого кодирования. Однако не все из них представлены единицами и нулями. Легко заметить, что три из них — другие числа. Это наши числовые переменные: `monthlycharges`, `tenure` и `totalcharges`.

Мы можем узнать имена всех этих столбцов, используя метод `get_feature_names`:

```
dv.get_feature_names()
```

Он выводит:

```
['contract=month-to-month',
 'contract=one_year',
 'contract=two_year',
```

```
'dependents=no',
'dependents=yes',
# некоторые строки пропущены
'tenure',
'totalcharges']
```

Как видим, для каждого категориального признака создается несколько столбцов для каждого из его отдельных значений. Для `contract` у нас `contract=month-to-month`, `contract=one_year` и `contract=two_year`, а для `dependents` — `dependents=no` и `dependents=yes`. Такие признаки, как `tenure` и `totalcharges`, сохраняют исходные названия, поскольку являются числовыми, и `DictVectorizer` их не меняет.

Теперь наши функции закодированы в виде матрицы, поэтому мы можем перейти к следующему шагу: использованию модели для прогнозирования оттока.

### Упражнение 3.2

Как бы `DictVectorizer` закодировал следующий список словарей?

```
records = [
    {'total_charges': 10, 'paperless_billing': 'yes'},
    {'total_charges': 30, 'paperless_billing': 'no'},
    {'total_charges': 20, 'paperless_billing': 'no'}
]
```

- A. Столбцы: `['total_charges', 'paperless_billing=yes', 'paperless_billing=no']`.

Значения: `[10, 1, 0], [30, 0, 1], [20, 0, 1]`.

- B. Столбцы: `['total_charges=10', 'total_charges=20', 'total_charges=30', 'paperless_billing=yes', 'paperless_billing=no']`.

Значения: `[1, 0, 0, 1, 0], [0, 0, 1, 0, 1], [0, 1, 0, 0, 1]`.

## 3.3. МАШИННОЕ ОБУЧЕНИЕ ДЛЯ КЛАССИФИКАЦИИ

Мы научились использовать Scikit-learn для выполнения прямого кодирования категориальных переменных и теперь можем преобразовать их в набор числовых признаков и собрать все это в матрицу.

Когда у нас будет матрица, мы уже сможем выполнить часть обучения модели. В этом разделе мы узнаем, как обучить модель логистической регрессии и интерпретировать результаты ее работы.

### 3.3.1. Логистическая регрессия

В этой главе мы используем логистическую регрессию в качестве модели классификации, и на этот раз мы обучим ее различать ушедших и не ушедших пользователей.

Логистическая регрессия имеет много общего с линейной, которую мы изучали в предыдущей главе. Напомним: модель линейной регрессии — это регрессионная модель, которая может прогнозировать число. Она имеет форму:

$$g(x_i) = w_0 + x_i^T w,$$

где

- $x_i$  — вектор признаков, соответствующий  $i$ -му наблюдению;
- $w_0$  — компонент смещения;
- $w$  — вектор с весами модели.

Мы применяем эту модель и получаем  $g(x_i)$  — прогноз того, каким, по нашему мнению, должно быть значение для  $x_i$ . Линейная регрессия обучается прогнозировать целевую переменную  $y_i$  — фактическое значение наблюдения  $i$ . В предыдущей главе это была цена автомобиля.

Линейная регрессия — линейная модель. Она называется *линейной*, поскольку объединяет веса модели с вектором признаков *линейно*, используя скалярное произведение. Линейные модели просты как в реализации, так и в обучении и использовании. Из-за своей простоты они еще и быстро работают.

Логистическая регрессия также является линейной моделью, но, в отличие от линейной регрессии, это классификационная модель, а не регрессионная, хотя название и указывает на обратное. Это модель двоичной классификации, поэтому целевая переменная  $y_i$  является двоичной; единственные значения, которые она может принять, — это ноль и единица. Наблюдения с  $y_i = 1$  обычно называются *положительными примерами*: примерами, в которых присутствует эффект, который мы хотим спрогнозировать. Аналогично примеры с  $y_i = 0$  называются *отрицательными примерами*: прогнозируемый эффект в них отсутствует. Для нашего проекта  $y_i = 1$  означает, что клиент ушел, а  $y_i = 0$  означает обратное: клиент остался в компании.

Результатом логистической регрессии является вероятность того, что наблюдение  $x_i$  является положительным, или, другими словами, вероятность того, что  $y_i = 1$ . В нашем случае это вероятность того, что клиент  $i$  расторгнет договор.

Чтобы рассматривать выходные данные как вероятность, нам нужно гарантировать, что прогнозы модели всегда остаются в диапазоне между нулем и единицей.

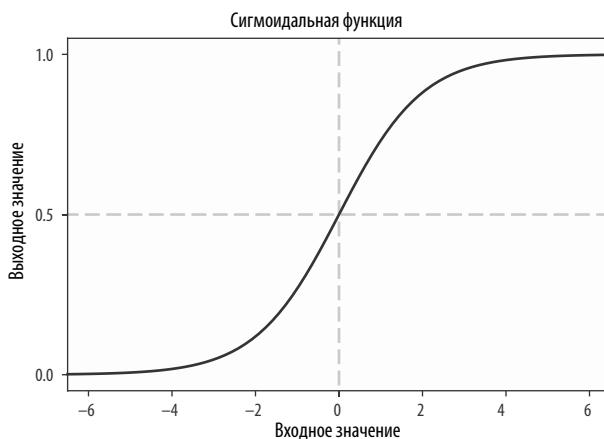
Для этой цели мы используем специальную математическую функцию, называемую *сигмоидальной*, и полная формула для модели логистической регрессии выглядит следующим образом:

$$g(x_i) = \text{sigmoid}(w_0 + x_i^T w).$$

Если мы сравним это с формулой линейной регрессии, то единственным различием будет эта сигмоидальная функция: в случае линейной регрессии мы имеем только  $w_0 + x_i^T w$ . Вот почему обе эти модели линейны — они основаны на операции скалярного произведения.

Сигмоидальная функция преобразует любое значение в число от нуля до единицы (рис. 3.24). Это определяется следующим образом:

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$



**Рис. 3.24.** Сигмоидальная функция выдает значения, которые всегда находятся в диапазоне от 0 до 1. Когда входные данные равны 0, результат сигмоидальной функции равен 0,5; при отрицательных значениях результаты ниже 0,5 и начинают приближаться к 0 при входных значениях меньше -6. Когда входные данные положительны, результат превышает 0,5 и приближается к 1 для входных значений, начиная с 6

Из главы 2 мы помним, что если вектор признаков  $x_i$  является  $n$ -мерным, то скалярное произведение  $x_i^T w$  может быть развернуто в виде суммы, и мы можем записать  $g(x_i)$  как

$$g(x_i) = \text{sigmoid}(w_0 + x_{i1}w_1 + x_{i2}w_2 + \dots + x_{in}w_n).$$

Или, используя обозначение суммы, как

$$g(x_i) = \text{sigmoid}\left(w_0 + \sum_{j=1}^n x_{ij} w_j\right).$$

Ранее мы переводили формулы на Python для иллюстрации. Сделаем то же самое и здесь.

Модель линейной регрессии имеет следующую формулу:

$$g(x_i) = w_0 + \sum_{j=1}^n x_{ij} w_j.$$

Если вы помните из предыдущей главы, эта формула преобразуется в следующий код Python:

```
def linear_regression(xi):
    result = bias
    for j in range(n):
        result = result + xi[j] * w[j]
    return result
```

Перевод формулы логистической регрессии на Python почти идентичен случаю линейной регрессии, за исключением того, что в конце мы применяем сигмоидальную функцию:

```
def logistic_regression(xi):
    score = bias
    for j in range(n):
        score = score + xi[j] * w[j]
    prob = sigmoid(score)
    return prob
```

Конечно, нам также нужно определить эту сигмоидальную функцию:

```
import math

def sigmoid(score):
    return 1 / (1 + math.exp(-score))
```

Мы используем *оценку* (score) для обозначения промежуточного результата перед применением сигмоидальной функции. Оценка может принимать любое реальное значение. *Вероятность* представляет собой результат применения сигмоидальной функции к оценке; это конечный результат, и он может принимать только значения от нуля до единицы.

Параметры модели логистической регрессии такие же, как и в случае линейной:

- $w_0$  — компонент смещения;
- $w = (w_1, w_2, \dots, w_n)$  — вектор весов.

Чтобы узнать веса, нам нужно обучить модель, что мы сейчас и сделаем с помощью Scikit-learn.

### Упражнение 3.3

Зачем нам нужна сигмоидальная функция для логистической регрессии?

- А. Сигмоидальная функция преобразует выходные данные в значения от  $-6$  до  $6$ , с которыми легче иметь дело.
- Б. Она гарантирует, что выходное значение остается между нулем и единицей, что может быть интерпретировано как вероятность.

### 3.3.2. Обучение логистической регрессии

Прежде всего импортируем модель:

```
from sklearn.linear_model import LogisticRegression
```

Затем мы обучим ее, вызвав метод `fit`:

```
model = LogisticRegression(solver='liblinear', random_state=1)
model.fit(X_train, y_train)
```

Класс `LogisticRegression` из Scikit-learn инкапсулирует логику обучения, лежащую в основе этой модели. Он настраивается, и мы можем изменять довольно много параметров. Фактически мы уже указали два из них: `solver` и `random_state`. И тот и другой необходимы для воспроизводимости:

- параметр `random_state` — начальное значение для генератора случайных чисел. Данные при обучении модели перетасовываются, и чтобы убедиться, что перетасовка каждый раз одинакова, мы фиксируем начальное значение;
- параметр `solver` — базовая библиотека оптимизации. В текущей версии (на момент написания это v0.20.3) значение по умолчанию для этого параметра равно `liblinear`, но согласно документации ([https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.LogisticRegression.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)) изменится на другое в версии v0.22. Чтобы убедиться, что наши результаты воспроизводимы в более поздних версиях, мы также устанавливаем этот параметр.

Другие полезные параметры для модели включают параметр `C`, который управляет уровнем регуляризации. Мы поговорим об этом в следующей главе, когда будем рассматривать настройку параметров. Указание `C` необязательно, по умолчанию оно принимает значение `1.0`.

Обучение занимает несколько секунд, и после его завершения модель готова делать прогнозы. Посмотрим, насколько хорошо она работает. Мы можем

применить ее к нашим проверочным данным, чтобы получить вероятность оттока для каждого клиента в проверочном наборе.

Для этого нам придется применить схему прямого кодирования ко всем категориальным переменным. Сначала мы преобразуем датафрейм в список словарей, а затем передадим его в `DictVectorizer`, который мы обучили ранее:

```
val_dict = df_val[categorical + numerical].to_dict(orient='records') ←
X_val = dv.transform(val_dict) ←
    Вместо подгонки, а затем преобразования мы
    используем преобразование, которое подгоняли ранее
    Мы выполняем прямое кодирование точно
    так же, как и во время обучения
```

В результате мы получаем `X_val`, матрицу с признаками из проверочного набора данных. Теперь мы готовы применить ее к модели. Чтобы получить вероятности, мы используем метод модели `predict_proba`:

```
y_pred = model.predict_proba(X_val)
```

Результатом `predict_proba` служит двумерный массив NumPy или матрица из двух столбцов. Первый столбец массива содержит вероятность того, что цель отрицательна (отток отсутствует), а второй — вероятность того, что она положительна (отток) (рис. 3.25).

model.predict_proba(X_val)	
Вероятность того, что наблюдение относится к отрицательному классу, то есть клиент не уйдет	array([[0.76508957, 0.23491043], [0.73113584, 0.26886416], [0.68054864, 0.31945136], ..., [0.94274779, 0.05725221], [0.38476995, 0.61523005], [0.9387273 , 0.0612727 ]])
	Вероятность того, что наблюдение относится к положительному классу, то есть клиент уйдет

**Рис. 3.25.** Прогнозы модели: матрица из двух столбцов. Первый столбец содержит вероятность того, что цель равна нулю (клиент не уйдет). Второй столбец содержит противоположную вероятность (цель равна единице, и клиент уйдет)

Эти столбцы содержат одну и ту же информацию. Мы знаем вероятность оттока — это  $p$ , а вероятность отсутствия оттока всегда равна  $1 - p$ , поэтому нам не нужны оба столбца.

Таким образом, нам достаточно иметь только второй столбец прогноза. Чтобы выбрать только один столбец из двумерного массива в NumPy, мы можем использовать операцию среза `[:, 1]`:

```
y_pred = model.predict_proba(X_val)[:, 1]
```

Синтаксис выглядит запутанным, поэтому разберем его. Внутри квадратных скобок две позиции: первая для строк, а вторая для столбцов.

Когда мы используем `[ :, 1]`, NumPy интерпретирует это так:

- `:` означает выбрать все строки.
- `1` означает, что выбрать нужно только столбец с индексом `1`, и поскольку индексация начинается с `0`, это второй столбец.

В результате мы получаем одномерный массив NumPy, который содержит значения только из второго столбца.

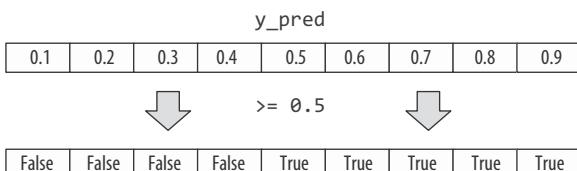
Такие выходные данные (вероятности) часто называют *мягкими* прогнозами. Они выражают вероятность ухода в виде числа от нуля до единицы. Нам остается решить, как интерпретировать это число и как его использовать.

Вспомните, как мы намеревались использовать эту модель: мы хотели удержать клиентов, выявив тех, кто собирается расторгнуть свой контракт, и отправив им после этого рекламные сообщения, предложения скидок и другое. Все это будет делаться в надежде, что после получения предложений клиенты останутся в компании. С другой стороны, мы не хотим проводить рекламные акции для всех клиентов подряд, поскольку это чревато финансовым ущербом: мы получим меньше прибыли, если таковая вообще будет.

Чтобы принять верное решение о том, следует ли отправлять рекламное письмо тому или иному клиенту, недостаточно использовать только вероятность. Нам нужны *твёрдые* прогнозы — двоичные значения `True` (отток, поэтому отправляем письмо) или `False` (не отток, поэтому не отправляем письмо). Чтобы получить двоичные прогнозы, мы берем вероятности и делаем разрез выше определенного порога. Если вероятность для клиента выше этого порога, то мы прогнозируем отток, в противном случае — его отсутствие. Если мы выберем `0,5` в качестве такого порога, то делать двоичные прогнозы будет легко. Мы просто используем оператор `>=`:

```
y_pred >= 0.5
```

Операторы сравнения в NumPy применяются поэлементно, и результатом будет новый массив, содержащий только логические значения: `True` и `False`. Скрытым образом выполняется поэлементное сравнение для каждого элемента массива `y_pred`. Если элемент больше или равен `0,5`, то соответствующий элемент в выходном массиве получает значение `True`, в противном случае — `False` (рис. 3.26).



**Рис. 3.26.** В NumPy оператор `>=` применяется поэлементно. Для каждого элемента он выполняет сравнение, и результатом становится еще один массив со значениями `True` или `False`, в зависимости от результата сравнения

Запишем результаты в массив `churn`:

```
churn = y_pred >= 0.5
```

Получив эти новые точные прогнозы, необходимо понять, насколько они хороши. Для этого мы готовы перейти к следующему шагу: оценке качества прогнозов. В следующей главе мы уделим гораздо больше времени изучению различных методов оценки двоичной классификации, сейчас же проведем простую проверку, чтобы убедиться, что наша модель выяснила что-то полезное.

Самая простая проверка заключается в сравнении каждого прогноза с фактическим значением. Если мы прогнозируем отток и фактическое значение равно оттоку, или мы прогнозируем отсутствие оттока и фактическое значение не равно оттоку — наша модель дала правильный прогноз. Если прогнозы не совпадают, значит, они не очень хороши. Подсчитав, сколько раз наши прогнозы совпадали с фактическим значением, мы сможем использовать этот показатель для измерения качества нашей модели.

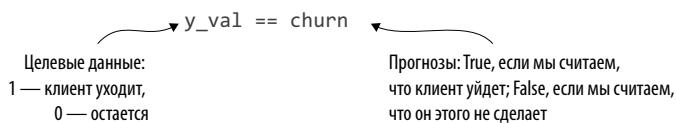
Такой показатель качества называется *достоверностью*. Очень легко рассчитать достоверность с помощью NumPy:

```
(y_val == churn).mean()
```

Несмотря на простоту вычисления, поначалу может быть трудно понять, что делает это выражение. Попробуем разбить его на отдельные шаги.

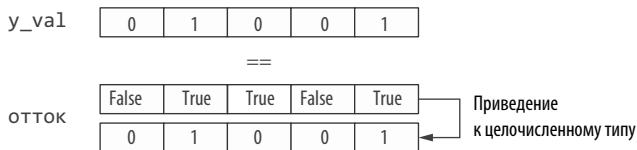
Сначала мы применяем оператор `==` для сравнения двух массивов NumPy: `y_val` и `churn`. Вспомним, что первый массив, `y_val`, содержит только числа: нули и единицы. Это наша целевая переменная: единица, если клиент расторгнул договор, и ноль в противном случае. Второй массив содержит логические прогнозы: значения `True` и `False`.

В данном случае значение `True` означает, что мы прогнозируем отток клиента, а значение `False` означает, что клиент останется (рис. 3.27).



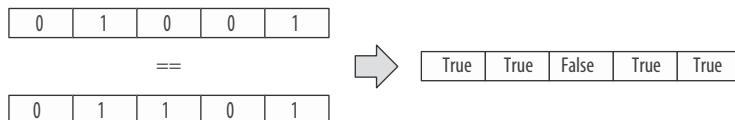
**Рис. 3.27.** Применение оператора `==` для сравнения целевых данных с нашими прогнозами

Несмотря на то что внутри этих двух массивов имеются различные типы (`integer` и `boolean`), их все равно можно сравнивать. Логический массив преобразуется в целочисленный таким образом, что `True` преобразуются в 1, а `False` — в 0. После этого NumPy уже может выполнить фактическое сравнение (рис. 3.28).



**Рис. 3.28.** Для сравнения прогноза с целевыми данными массив с прогнозами преобразуется в целочисленный

Как и оператор `>=`, оператор `==` применяется поэлементно. В этом случае у нас для сравнения имеются два массива, так что мы сравниваем каждый элемент одного массива с соответствующим элементом другого. Результатом опять станет логический массив со значениями `True` или `False`, в зависимости от результата сравнения (рис. 3.29).

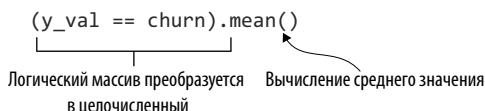


**Рис. 3.29.** Оператор `==` из NumPy применяется поэлементно для двух массивов NumPy

В нашем случае, если истинное значение в `y_pred` соответствует нашему прогнозу в `churn`, метка получает значение `True`, в ином случае — `False`. Другими словами, мы имеем значение `True`, если наш прогноз верный, и `False`, если это не так.

Наконец, мы берем результаты сравнения — массив Boolean — и вычисляем его среднее значение с помощью метода `mean()`.

Однако этот метод применяется к числам, а не к логическим значениям, поэтому перед вычислением среднего значения значения преобразуются в целые числа: `True` преобразуются в 1, а `False` — в 0 (рис. 3.30).



**Рис. 3.30.** При вычислении среднего значения логического массива NumPy сначала преобразует его в целочисленный, а затем вычисляет среднее значение

Как мы уже знаем, при вычислении среднего значения массива, содержащего только единицы и нули, результатом станет доля единиц в этом массиве, которую мы уже использовали для расчета коэффициента оттока. Поскольку 1 (`True`)

в данном случае отражает правильный прогноз, а `0 (False)` — неправильный, итоговое число сообщает нам процент правильных прогнозов.

После выполнения этой строки кода мы видим на выходе `0.8`. Это означает, что прогнозы модели соответствовали фактическому значению в 80 % случаев, или модель делает правильные прогнозы в 80 % случаев. Это то, что мы называем достоверностью модели.

Теперь мы знаем, как обучить модель и оценить ее достоверность, но все равно будет полезно понять, как именно она делает прогнозы. В следующем подразделе мы попытаемся заглянуть внутрь моделей и разобраться, как можно интерпретировать полученные коэффициенты.

### 3.3.3. Интерпретация модели

Мы знаем, что модель логистической регрессии имеет два параметра, которые она извлекает из данных:

- $w_0$  — это компонент смещения.
- $w = (w_1, w_2 \dots w_n)$  — вектор весов.

Мы можем получить компонент смещения из `model.intercept_[0]`. Когда мы обучаем нашу модель по всем признакам, коэффициент смещения равен `-0.12`.

Остальные веса хранятся в `model.coef_[0]`. Если мы заглянем внутрь, то увидим, что это просто массив чисел, значение которого трудно понять.

Чтобы увидеть, какой признак связан с каждым весом, воспользуемся методом `get_feature_names` из `DictVectorizer`. Предварительно можно объединить названия признаков с коэффициентами:

```
dict(zip(dv.get_feature_names(), model.coef_[0].round(3)))
```

Вывод будет таким:

```
{'contract=month-to-month': 0.563,
 'contract=one_year': -0.086,
 'contract=two_year': -0.599,
 'dependents=no': -0.03,
 'dependents=yes': -0.092,
 ... # остальные веса опущены
 'tenure': -0.069,
 'totalcharges': 0.0}
```

Чтобы понять, как работает модель, узнаем, что происходит при ее применении. Чтобы облегчить восприятие, обучим более простую и компактную модель, которая использует только три переменные: `contract`, `tenure` и `totalcharges`.

Переменные `tenure` и `totalcharges` являются числовыми, поэтому нам не потребуется никакая дополнительная предварительная обработка; мы можем использовать их как есть. С другой стороны, `contract` — это категориальная переменная, поэтому перед ее использованием необходимо применить прямое кодирование.

Повторим те же шаги, которые мы уже делали для обучения, но на этот раз используем меньший набор признаков:

```
small_subset = ['contract', 'tenure', 'totalcharges']
train_dict_small = df_train[small_subset].to_dict(orient='records')
dv_small = DictVectorizer(sparse=False)
dv_small.fit(train_dict_small)

X_small_train = dv_small.transform(train_dict_small)
```

Чтобы не путать код с предыдущей моделью, мы добавим `small` ко всем именам. Таким образом, становится ясно, что мы используем модель меньшего размера, и это избавляет нас от случайной перезаписи уже полученных результатов. Кроме того, мы будем использовать эти данные для сравнения качества маленькой и полной моделей.

Посмотрим, какие признаки будет использовать маленькая модель. Для этого, как и ранее, мы используем метод `get_feature_names` из `DictVectorizer`:

```
dv_small.get_feature_names()
```

Команда выводит имена признаков:

```
['contract=month-to-month',
 'contract=one_year',
 'contract=two_year',
 'tenure',
 'totalcharges']
```

У нас пять признаков. Как и ожидалось, в их числе `tenure` и `totalcharges`, и поскольку они числовые, их названия не изменяются.

Что касается переменной `contract`, то она категориальная, поэтому для преобразования ее в числовой формат `DictVectorizer` применяет схему прямого кодирования. Эта переменная имеет три различных значения: `month-to-month`, `one year` и `two years`. Следовательно, прямое кодирование даст три новых признака: `contract=month-to-month`, `contract=one_year` и `contract=two_year`.

Обучим на этом наборе признаков нашу небольшую модель:

```
model_small = LogisticRegression(solver='liblinear', random_state=1)
model_small.fit(X_small_train, y_train)
```

Модель готова через несколько секунд, и мы можем взглянуть на веса, которые она выучила. Сначала проверим компонент смещения:

```
model_small.intercept_[0]
```

Результат:  $-0,638$ . Далее, используя тот же код, что и ранее, мы можем проверить и другие веса:

```
dict(zip(dv_small.get_feature_names(), model_small.coef_[0].round(3)))
```

Эта строка кода выводит вес для каждого признака:

```
{'contract=month-to-month': 0.91,
'contract=one_year': -0.144,
'contract=two_year': -1.404,
'tenure': -0.097,
'totalcharges': 0.000}
```

Сведем полученные веса в одну таблицу и назовем  $w_1, w_2, w_3, w_4$  и  $w_5$  (табл. 3.2).

**Таблица 3.2.** Веса модели логистической регрессии

Смещение	Переменная contract			Переменная tenure	Переменная charges
	month	year	2-year		
$w_0$ -0,639	$w_1$ 0,91	$w_2$ -0,144	$w_3$ -1,404	$w_4$ -0,097	$w_5$ 0,0

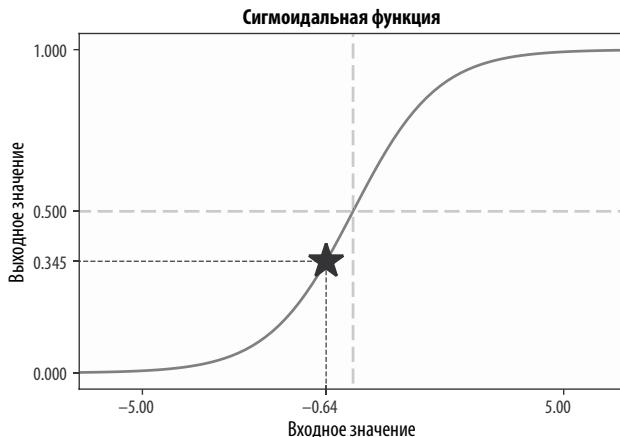
Теперь поближе рассмотрим эти веса и попытаемся понять, что они означают и как их можно интерпретировать.

Для начала остановимся на компоненте смещения и на том, что он означает. Мы помним, что в случае линейной регрессии это базовый прогноз, который мы бы сделали, не зная ничего больше о наблюдении. В проекте прогнозирования цен на автомобили это средняя цена автомобиля. Это не окончательный прогноз; позже этот базовый уровень корректируется с помощью других весовых коэффициентов.

В случае логистической регрессии все аналогично: это базовый прогноз — или оценка, которую мы получили бы в среднем. Аналогично позже мы корректируем данный результат с помощью других весов. Однако для логистической регрессии интерпретация несколько усложняется, поскольку нам также нужно применить сигмоидальную функцию, прежде чем мы получим окончательный результат. Рассмотрим пример, который поможет нам это понять.

В нашем случае компонент смещения имеет значение  $-0,639$ . Это значение отрицательное. Если мы посмотрим на сигмоидальную функцию, то увидим, что при отрицательных значениях выходное значение ниже 0,5 (рис. 3.31).

Для  $-0,639$  итоговая вероятность оттока составляет  $34\%$ . Это значит, что в среднем клиент с большей вероятностью останется с нами, нежели уйдет.

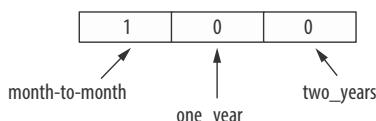


**Рис. 3.31.** Компонент смещения  $-0,639$  на сигмоидальной кривой.  
Итоговая вероятность составляет менее  $0,5$ , поэтому средний клиент,  
скорее всего, не расторгнет договор

Причина, по которой знак перед смещением отрицательный, заключается в дисбалансе классов. В обучающих данных намного меньше ушедших пользователей, чем оставшихся; это значит, вероятность оттока в среднем невелика, поэтому такое значение для компонента смещения вполне объяснимо. Следующие три веса представляют собой веса для переменной контракта. Поскольку мы используем прямое кодирование, у нас имеется три признака `contract` и три веса, по одному для каждого признака:

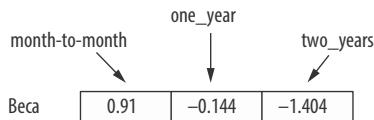
```
'contract=month-to-month': 0.91,
'contract=one_year': -0.144,
'contract=two_year': -1.404.
```

Чтобы улучшить понимание того, как можно интерпретировать веса после прямого кодирования, представим клиента с ежемесячным контрактом. Переменная `contract` имеет следующее представление после прямого кодирования: первая позиция соответствует значению `month-to-month` и является «горячей» частью, поэтому ей присвоено значение  $1$ . Остальные позиции соответствуют `one_year` и `two_years`, поэтому они «холодные» и установлены в  $0$  (рис. 3.32).



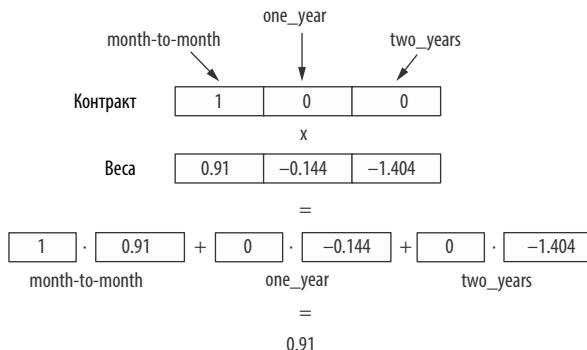
**Рис. 3.32.** Представление после прямого кодирования  
для клиента с ежемесячным контрактом

Мы также знаем веса  $w_1$ ,  $w_2$  и  $w_3$ , которые соответствуют `contract=month-to-month`, `contract=one_year` и `contract=two_years` (рис. 3.33).



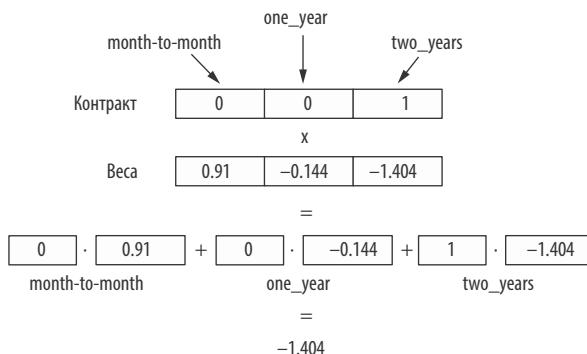
**Рис. 3.33.** Веса признаков `contract=month-to-month`, `contract=one_year`, и `contract=two_years`

Чтобы получить прогноз, мы выполняем скалярное произведение векторов признаков и весов, представляющее собой умножение значений в каждой позиции, а затем суммирование. Результат произведения составляет 0,91, то есть столько же, сколько и вес признака `contract=month-to-month` (рис. 3.34).



**Рис. 3.34.** Скалярное произведение представления переменной контракта после прямого кодирования и соответствующих весов. Результат составляет 0,91, что соответствует весу «горячего» признака

Рассмотрим другой пример: клиент с двухлетним контрактом. В этом случае «горячим» будет признак `contract=two_year` (он получает значение 1), а остальные



**Рис. 3.35.** Для клиента с двухлетним контрактом результат скалярного произведения составит -1,404

будут «холодными». Умножая вектор с представлением переменной после прямого кодирования на вектор весов, мы получаем  $-1,404$  (рис. 3.35).

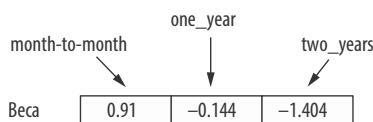
Как мы видим, во время прогнозирования учитывается только вес «горячего» признака, а остальные веса не учитываются. Это объяснимо: «холодные» признаки имеют нулевые значения, и, умножая на ноль, мы получаем ноль (рис. 3.36).

<table border="1"><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>x</td><td></td><td></td></tr><tr><td>0.91</td><td>-0.144</td><td>-1.404</td></tr></table>	1	0	0	x			0.91	-0.144	-1.404	<table border="1"><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>x</td><td></td><td></td></tr><tr><td>0.91</td><td>-0.144</td><td>-1.404</td></tr></table>	0	1	0	x			0.91	-0.144	-1.404	<table border="1"><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>x</td><td></td><td></td></tr><tr><td>0.91</td><td>-0.144</td><td>-1.404</td></tr></table>	0	0	1	x			0.91	-0.144	-1.404
1	0	0																											
x																													
0.91	-0.144	-1.404																											
0	1	0																											
x																													
0.91	-0.144	-1.404																											
0	0	1																											
x																													
0.91	-0.144	-1.404																											
=	=	=																											
0.91	-0.144	-1.404																											

**Рис. 3.36.** Когда мы умножаем представление переменной после прямого кодирования на вектор весов из модели, результатом становится вес, соответствующий «горячему» признаку

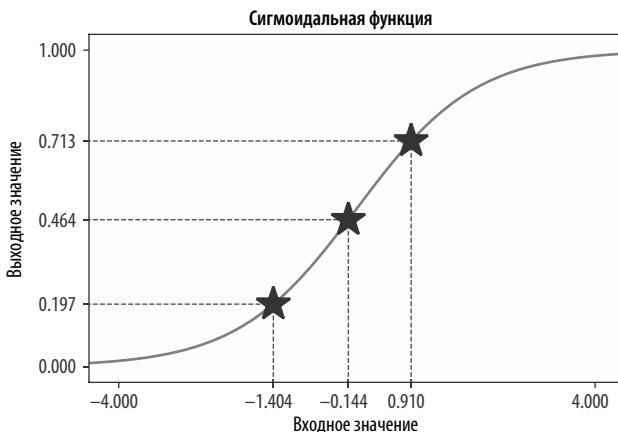
Интерпретация знаков весов для признаков с прямым кодированием следует той же логике, что и компонент смещения. Если вес положительный, то соответствующий признак является показателем оттока, и наоборот. Если он отрицательный, то, скорее всего, принадлежит клиенту, который не попадает в группу риска.

Еще раз посмотрим на веса переменной `contract`. Первый вес для `contract=month-to-month` положительный, поэтому клиенты с таким типом контракта скорее уйдут, чем останутся. Два других признака, `contract=one_year` и `contract=two_years`, имеют отрицательные знаки, поэтому такие клиенты с большей вероятностью останутся в компании (рис. 3.37).



**Рис. 3.37.** Знак веса имеет значение. Если он положительный, то это хороший показатель оттока; если отрицательный, то это указывает на лояльного клиента

Величина весов также имеет значение. Для `two_year` вес равен  $-1,404$ , что больше по величине, чем  $-0,144$  (вес `one_year`). Таким образом, двухлетний контракт является более сильным стимулом оставаться, чем однолетний. Это подтверждает анализ важности признаков, который мы проводили ранее. Коэффициенты риска (риск оттока) для этого набора признаков составляют 1,55 для ежемесячного контракта, 0,44 для годичного и 0,10 для двухлетнего (рис. 3.38).



**Рис. 3.38.** Веса для признаков контракта и их перевод в вероятности.

В случае `contract=two_year` вес равен  $-1,404$ , что означает очень низкую вероятность оттока. В случае `contract=one_year` вес равен  $-0,144$ , так что вероятность умеренная. А вот для `contract=month-to-month` вес составляет  $0,910$ , и вероятность в данном случае довольно высока

Теперь обратимся к числовым признакам. У нас их два: `tenure` и `totalcharges`. Вес признака `tenure` составляет  $-0,097$ , и он имеет отрицательный знак. Это означает все то же: признак является показателем отсутствия оттока.

Из анализа важности признаков мы уже знаем, что чем дольше клиенты остаются с нами, тем меньше вероятность их оттока. Корреляция между `tenure` и оттоком составляет  $-0,35$ , что также является отрицательным числом. Вес этого признака ее лишь подтверждает: за каждый месяц, который клиент проводит с нами, общая оценка снижается на  $0,097$ .

Другой числовой признак, `totalcharges`, имеет нулевой вес. Поскольку он нулевой, то, независимо от значения этого признака, модель не будет его учитывать, поэтому данный признак в действительности не важен для получения прогноза.

Чтобы лучше это понять, рассмотрим пару примеров. В случае первого представим, что у нас есть пользователь с ежемесячным контрактом, который провел с нами год и заплатил 1000 долларов (рис. 3.39).

$$-0.639 + 0.91 - 12 \cdot 0.097 + 0 \cdot 1000 = -0.893$$

Смещение Ежемесячный 12 месяцев Общая сумма расходов Отрицательный  
контракт (срок контракта) не имеет значения показатель, поэтому  
вероятность оттока низкая

**Рис. 3.39.** Оценка, рассчитанная моделью для клиента с помесячным контрактом и 12 месяцами общего срока контракта

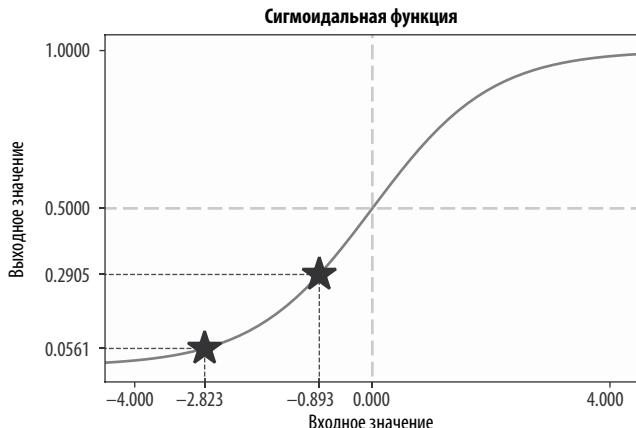
Ниже представлен прогноз, который мы делаем для этого клиента.

- Мы начинаем с базового балла. Это компонент смещения со значением  $-0,639$ .
- Поскольку это помесячный контракт, мы добавляем  $0,91$  к этому значению и получаем  $0,271$ . Оценка становится положительной, так что это может означать, что клиент собирается уйти. Мы знаем, что ежемесячный контракт является сильным показателем оттока.
- Далее мы рассмотрим переменную `tenure`. За каждый месяц, в течение которого клиент оставался с нами, мы вычитаем  $0,097$  из полученного на данный момент балла. Таким образом, мы получаем  $0,271 - 12 \times 0,097 = -0,893$ . Оценка снова отрицательная, так что вероятность оттока уменьшается.
- Теперь мы добавляем сумму, которую заплатил нам клиент (`totalcharges`), умноженную на вес этого признака, но поскольку он нулевой, мы по сути ничего не делаем. Результат остается прежним:  $-0,893$ .
- Итоговый балл — отрицательное число, поэтому мы считаем, что вероятность оттока клиента в ближайшее время невелика.
- Чтобы увидеть фактическую вероятность оттока, мы вычисляем сигмоидальную функцию оценки, и она составляет примерно  $0,29$ . Ее мы можем рассматривать как вероятность того, что этот клиент уйдет.

$$-0.639 + 0.144 - 24 \cdot 0.097 + 0 \cdot 2000 = -2.823$$

Смещение	Годичный контракт	24 месяца (срок контракта)	Общая сумма расходов	Отрицательное значение, очень низкая вероятность оттока
			не имеет значения	

**Рис. 3.40.** Оценка, рассчитанная моделью для клиента с годовым контрактом и общим сроком контракта 24 месяца



**Рис. 3.41.** Оценки  $-2,823$  и  $-0,893$  переведены в вероятность:  $0,05$  и  $0,29$  соответственно

Если у нас имеется еще один клиент с годовым контрактом, который проработал с нами 24 месяца и потратил 2000 долларов, то оценка составит  $-2,823$  (рис. 3.40).

После применения сигмоидальной функции оценка  $-2,823$  превращается в 0,056, поэтому вероятность оттока для этого клиента еще ниже (рис. 3.41).

### **3.3.4. Использование модели**

Теперь мы намного лучше знаем, как работает логистическая регрессия, можем интерпретировать то, что узнала наша модель, и понимать, как именно она делает свои прогнозы.

Кроме того, мы применили модель к проверочному набору, вычислили вероятности оттока для каждого клиента и пришли к выводу, что модель достоверна на 80 %. В следующей главе мы узнаем, является ли это число удовлетворительным, а пока попробуем использовать свою обученную модель. Теперь мы можем с помощью данной модели оценивать клиентов. Это довольно просто.

Сначала мы берем клиента, которого хотим оценить, и помещаем все значения переменных в словарь:

```
customer = {
    'customerid': '8879-zkjof',
    'gender': 'female',
    'seniorcitizen': 0,
    'partner': 'no',
    'dependents': 'no',
    'tenure': 41,
    'phoneservice': 'yes',
    'multiplelines': 'no',
    'internetservice': 'dsl',
    'onlinesecurity': 'yes',
    'onlinebackup': 'no',
    'deviceprotection': 'yes',
    'techsupport': 'yes',
    'streamingtv': 'yes',
    'streamingmovies': 'yes',
    'contract': 'one_year',
    'paperlessbilling': 'yes',
    'paymentmethod': 'bank_transfer_(automatic)',
    'monthlycharges': 79.85,
    'totalcharges': 3320.75,
}
```

#### **ПРИМЕЧАНИЕ**

Когда мы готовим элементы для прогнозирования, они должны пройти те же этапы предварительной обработки, что и при обучении. Если мы этого не сделаем, то модель не получит ожидаемых данных, и в этом случае прогнозы могут действительно быть

недостоверными. Вот почему в предыдущем примере в словаре `customer` имена полей и строковые значения указаны в нижнем регистре, а пробелы заменены символами подчёркивания.

Теперь мы можем использовать нашу модель и узнать, подвержен ли данный клиент оттоку. Выясним это.

Сначала преобразуем словарь в матрицу с помощью `DictVectorizer`:

```
X_test = dv.transform([customer])
```

Входные данные для векторизатора представляют собой список с одним элементом: мы хотим оценить лишь одного клиента. Результатом будет матрица с признаками, и она содержит только одну строку — признаки для этого единственного клиента:

```
[[ 0. , 1. , 0. , 1. , 0. , 0. , 0. ,
  1. , 1. , 0. , 1. , 0. , 0. , 79.85,
  1. , 0. , 0. , 1. , 0. , 0. , 0. ,
  0. , 1. , 0. , 1. , 1. , 0. , 1. ,
  0. , 0. , 0. , 0. , 1. , 0. , 0. ,
  0. , 1. , 0. , 0. , 1. , 0. , 0. ,
  1. , 41. , 3320.75]]
```

Мы видим множество признаков после прямого кодирования (единицы и нули), а также несколько числовых (`monthlycharges`, `tenure` и `totalcharges`).

Далее мы берем эту матрицу и отправляем в обученную модель:

```
model.predict_proba(X_test)
```

На выходе получаем матрицу с прогнозами. Для каждого клиента она выводит два числа, которые представляют собой вероятность того, что он останется в компании, и вероятность оттока. Поскольку клиент у нас только один, мы получаем крошечный массив NumPy с одной строкой и двумя столбцами:

```
[[0.93, 0.07]]
```

Все, что нам нужно из этой матрицы, — число в первой строке и втором столбце: вероятность оттока для этого клиента. Чтобы выбрать данное число из массива, мы используем оператор скобок:

```
model.predict_proba(X_test)[0, 1]
```

Мы используем данный оператор, чтобы выбрать второй столбец из массива. Однако на сей раз у нас лишь одна строка, поэтому мы можем явно попросить NumPy вернуть значение из этой строки. Поскольку индексы начинаются с 0 в NumPy, `[0, 1]` означает первую строку, второй столбец.

Выполнив эту команду, мы видим, что результат равен 0,073, так что вероятность оттока клиента составляет всего 7 %. Это менее 50 %, поэтому мы не будем отправлять такому клиенту рекламное письмо.

Мы можем попытаться оценить другого клиента:

```
customer = {  
    'gender': 'female',  
    'seniorcitizen': 1,  
    'partner': 'no',  
    'dependents': 'no',  
    'phoneservice': 'yes',  
    'multiplelines': 'yes',  
    'internetservice': 'fiber_optic',  
    'onlinesecurity': 'no',  
    'onlinebackup': 'no',  
    'deviceprotection': 'no',  
    'techsupport': 'no',  
    'streamingtv': 'yes',  
    'streamingmovies': 'no',  
    'contract': 'month-to-month',  
    'paperlessbilling': 'yes',  
    'paymentmethod': 'electronic_check',  
    'tenure': 1,  
    'monthlycharges': 85.7,  
    'totalcharges': 85.7  
}
```

Давайте сделаем прогноз:

```
X_test = dv.transform([customer])  
model.predict_proba(X_test)[0, 1]
```

Результатом работы модели становится 83 %-ная вероятность оттока, поэтому нам следует направить этому клиенту рекламное письмо в надежде удержать его.

До сих пор мы разбирались с тем, как работает логистическая регрессия, как обучать ее с помощью Scikit-learn и как применять ее к новым данным. Но мы пока не рассматривали оценку результатов, и это как раз то, что мы сделаем в следующей главе.

## 3.4. СЛЕДУЮЩИЕ ШАГИ

### 3.4.1. Упражнения

Чтобы углубиться в тему, вы можете попробовать выполнить несколько действий.

- В предыдущей главе мы многое реализовали самостоятельно, включая линейную регрессию и разделение набора данных. В данной главе мы узнали,

как использовать для этого Scikit-learn. Попробуйте переделать проект из предыдущей главы с помощью Scikit-learn. Чтобы использовать линейную регрессию, вам понадобится `LinearRegression` из пакета `sklearn.linear_model`. Чтобы использовать регуляризованную регрессию, вам необходимо импортировать `Ridge` из того же пакета `sklearn.linear_model`.

- Мы рассмотрели, как на основе показателей важности признаков получить некое представление о наборе данных, однако на самом деле не воспользовались этой информацией для каких-то других целей. Одним из способов ее использования может быть удаление ненужных признаков из набора данных, что упростит, ускорит и потенциально улучшит модель. Попробуйте исключить два наименее полезных признака (`gender` и `phoneservices`) из матрицы обучающих данных и посмотрите, что произойдет с достоверностью проверки. А что если мы удалим самый полезный признак (`contract`)?

### 3.4.2. Другие проекты

Классификация может стать способом решения множества реальных проблем, и теперь, после изучения материалов этой главы, у вас должно быть достаточно знаний, чтобы применять логистическую регрессию для решения аналогичных задач. В частности, предлагаемые нами варианты использования описаны ниже.

- Классификационные модели часто используются в маркетинговых целях, и одна из задач, которую они решают, — это *оценка лидов*. *Лид* — потенциальный клиент, который может конвертироваться в реального (или нет). В таком случае конверсия — это цель, которую мы хотим спрогнозировать. Вы можете взять набор данных из <https://www.kaggle.com/ashydv/leads-dataset> и построить для него модель. Легко заметить, что проблема подсчета лидеров похожа на прогнозирование оттока, но в одном случае мы хотим, чтобы новый клиент подписал контракт, а в другом случае — чтобы клиент не расторгал его.
- Другим популярным применением классификации является прогнозирование дефолта, которое заключается в оценке риска невозврата клиентом кредита. В таком случае переменная, которую мы хотим предсказать, — это дефолт, и у нее также есть два значения: удалось ли клиенту вернуть кредит вовремя (хороший клиент) или нет (дефолт). Для обучения модели вы можете найти в Интернете множество наборов данных, например: <https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients> (или такой же, доступный через Kaggle: <https://www.kaggle.com/pratjain/credit-card-default>).

## РЕЗЮМЕ

- *Risk* категориального признака говорит нам, будет ли группа, обладающая этим признаком, иметь моделируемое условие. В случае оттока значения ниже 1,0 указывают на низкий риск, тогда как значения выше 1,0 — на высокий. Это позволяет нам определить, какие признаки важны для прогнозирования целевой переменной, и помогает лучше понять решаемую задачу.
- Взаимная информация позволяет измерить степень (не)зависимости категориальной переменной и цели. Это хороший способ нахождения важных признаков: чем выше взаимная информация, тем важнее признак.
- Корреляция измеряет зависимость между двумя числовыми переменными, и ее можно использовать для определения того, полезен ли числовой признак для прогнозирования целевой переменной.
- Прямое кодирование дает нам способ представления категориальных переменных в виде чисел. Без него было бы невозможно так легко использовать эти переменные в модели. Модели машинного обучения обычно ожидают, что все входные переменные будут числовыми, поэтому наличие схемы кодирования имеет решающее значение, если мы хотим использовать категориальные признаки.
- Мы можем реализовать прямое кодирование с помощью `DictVectorizer` из Scikit-learn. Он автоматически определяет категориальные переменные и применяет к ним схему прямого кодирования, оставляя числовые переменные нетронутыми. Это очень удобно и не требует от нас написания большого количества кода.
- Логистическая регрессия — линейная модель, такая же, как и линейная регрессия. Разница в том, что логистическая регрессия имеет дополнительный шаг в конце: применяет сигмоидальную функцию для преобразования оценок в вероятности (число от нуля до единицы). Это позволяет нам использовать ее для классификации. Результатом является вероятность принадлежности к положительному классу (в нашем случае к оттоку).
- Когда данные подготовлены, обучить логистическую регрессию очень просто: мы используем класс `LogisticRegression` из Scikit-learn и вызываем функцию `fit`.
- Модель выдает вероятности, а не твердые прогнозы. Чтобы превратить выходные данные в двоичные, мы делаем срез прогноза до определенного порога. Если вероятность больше или равна 0,5, то мы прогнозируем `True` (отток) и `False` (без оттока) в противном случае. Это позволяет нам использовать модель для решения нашей задачи — прогнозирования оттока клиентов.

- Веса модели логистической регрессии легко интерпретировать и объяснить, особенно когда речь заходит о категориальных переменных, преобразованных с помощью прямого кодирования. Это помогает лучше понять поведение модели и объяснить другим, что именно она делает и как работает.

В следующей главе продолжим этот проект по прогнозированию оттока. Мы рассмотрим способы оценки бинарных классификаторов, а затем используем эту информацию для настройки производительности модели.

## ОТВЕТЫ К УПРАЖНЕНИЯМ

- Упражнение 3.1. Ответ Б. Процент элементов `True`.
- Упражнение 3.2. Ответ А. Он сохранит числовую переменную как есть и закодирует только категориальную переменную.
- Упражнение 3.3. Ответ Б. Сигмоидальная функция преобразует выходные данные в значение от нуля до единицы.

# *Оценочные показатели для классификации*

## **В этой главе**

- ✓ Достоверность как способ оценки моделей бинарной классификации и ее ограничения.
- ✓ Определение мест, в которых наша модель допускает ошибки, с помощью матрицы ошибок.
- ✓ Получение других показателей, таких как точность и отклик, из матрицы ошибок.
- ✓ Использование ROC (receiver operating characteristics, рабочие характеристики приемника) и AUC (area under the ROC curve, площадь под кривой ROC) для дальнейшего понимания производительности модели бинарной классификации.
- ✓ Перекрестная проверка модели, помогающая убедиться, что она ведет себя оптимально.
- ✓ Настройка параметров модели для достижения наилучших результатов прогнозирования.

В этой главе мы продолжаем проект, начатый в предыдущей: прогнозирование оттока. Мы уже загрузили набор данных, выполнили первоначальную

предварительную обработку и исследовательский анализ данных, после чего обучили модель, которая предсказывает, ожидается ли отток клиентов. Мы также оценили эту модель на основе проверочного набора данных и пришли к выводу, что ее достоверность 80 %.

Вопрос, который мы откладывали до сих пор, заключался в том, хороша ли сама по себе достоверность в 80 % и что она на самом деле означает с точки зрения качества нашей модели. В этой главе мы получим ответ на данный вопрос, а также обсудим другие способы оценки модели бинарной классификации: матрицу ошибок, точность и отклик, кривую ROC и AUC.

Далее содержится много сложной информации, но показатели оценки, которые мы рассматриваем, необходимы для практического машинного обучения. Не волнуйтесь, если вы не сразу поймете все тонкости различных показателей оценки — это требует времени и практики. Спокойно возвращайтесь к этой главе, чтобы повторно изучить какие-то нюансы.

## 4.1. ПОКАЗАТЕЛИ ОЦЕНКИ

Мы уже построили модель бинарной классификации для прогнозирования оттока клиентов. Теперь нам нужно научиться определять, насколько она хороша.

Для этого мы используем *метрику* — функцию, которая просматривает прогнозы, сделанные моделью, и сравнивает их с фактическими значениями. Затем на основе данного сравнения она вычисляет, насколько хороша модель. Это крайне полезно: мы можем с ее помощью сравнивать различные модели и выбирать ту, которая имеет наилучшее значение этого показателя.

Существуют различные виды показателей. В главе 2 мы с помощью RMSE (среднеквадратичной ошибки) оценивали регрессионные модели. Однако эта метрика может использоваться только для регрессионных моделей и не работает для классификации.

Для оценки классификационных моделей у нас есть другие, более подходящие показатели. В этом разделе мы рассмотрим наиболее распространенные оценочные показатели для двоичной классификации, начиная с достоверности, которую мы уже получали в главе 3.

### 4.1.1. Достоверность классификации

Как вы, вероятно, помните, достоверность модели бинарной классификации — это процент правильных прогнозов, которые она дает (рис. 4.1).



**Рис. 4.1.** Достоверность модели — это доля прогнозов, которые оказались правильными

Достоверность — самый простой способ оценить классификатор: подсчитав количество случаев, в которых наша модель оказалась права, мы можем многое узнать о ее поведении и качестве.

Вычислить достоверность по набору проверочных данных проверки легко — мы просто вычисляем долю верных прогнозов:

```
y_pred = model.predict_proba(X_val)[:, 1]    ❶ Получает прогнозы из модели
churn = y_pred >= 0.5    ❷ Делает «твёрдые» прогнозы
(churn == y_val).mean()    ❸ Вычисляет достоверность
```

Сначала мы применяем модель к проверочному набору, чтобы получить прогнозы в ❶. Они являются вероятностями, поэтому мы делаем срез до 0,5 в ❷. Наконец мы вычисляем долю прогнозов, которые соответствовали реальности в ❸.

Результат равен 0,8016, что означает, что наша модель достоверна на 80 %.

Первое, что мы должны выяснить, — почему мы выбрали именно 0,5 в качестве порогового значения. Это был произвольный выбор, однако на самом деле не трудно проверить и другие пороговые значения: мы можем просто перебрать все возможные пороговые значения и вычислить достоверность для каждого. Затем можем выбрать то, которое демонстрирует наилучший показатель достоверности.

Несмотря на то что достоверность легко реализовать самостоятельно, мы можем использовать и существующие реализации. Библиотека Scikit-learn предлагает множество показателей, включая достоверность и многие другие, которые будем применять позже. Вы можете найти эти показатели в пакете метрик.

Мы продолжим работу с тем же блокнотом, который начали в главе 3. Откроем его и добавим инструкцию `import` для импорта достоверности из пакета показателей Scikit-learn:

```
from sklearn.metrics import accuracy_score
```

Теперь мы можем перебрать различные пороговые значения и выяснить, какое из них дает наилучшую достоверность:

```
thresholds = np.linspace(0, 1, 11) ← Создает массив с различными пороговыми
for t in thresholds:               значениями: 0,0, 0,1, 0,2 и т. д.
    churn = y_pred >= t           ← Использует функцию accuracy_score из Scikit-learn
    acc = accuracy_score(y_val, churn) ← для вычисления достоверности
    print('%0.2f %0.3f' % (t, acc)) ← Выводит пороговые значения и значения
                                    достоверности в стандартный вывод
Циклы по каждому
пороговому значению
```

В этом коде мы сначала создаем массив с пороговыми значениями. Для этого используем функцию `linspace` из NumPy: она принимает два числа (в нашем случае 0 и 1) и количество элементов, которые должен содержать массив (11). В результате мы получаем массив с числами 0,0, 0,1, 0,2... 1,0. Больше информации о `linspace` и других функциях NumPy вы можете узнать в приложении B.

Мы используем эти числа в качестве пороговых значений, перебирая их и вычисляя достоверность для каждого. Наконец, выводим пороговые значения и оценки достоверности, чтобы решить, какой порог является наилучшим.

Выполнив код, мы увидим следующее:

```
0.00  0.261
0.10  0.595
0.20  0.690
0.30  0.755
0.40  0.782
0.50  0.802
0.60  0.790
0.70  0.774
0.80  0.742
0.90  0.739
1.00  0.739
```

Как мы видим, использование порога 0,5 дает наилучшую достоверность. Как правило, 0,5 – хорошее начальное пороговое значение, но мы всегда должны пробовать и другие, чтобы убедиться, что 0,5 – действительно лучший выбор.

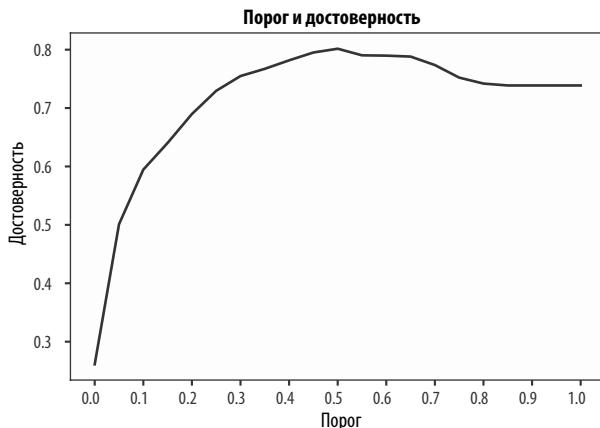
Для наглядности можем использовать Matplotlib для создания графика, который показывает, как изменяется достоверность в зависимости от порогового значения. Мы повторяем тот же процесс, что и ранее, но вместо того, чтобы просто печатать показатели достоверности, сначала помещаем значения в список:

```
thresholds = np.linspace(0, 1, 21) ← Создает различные пороговые значения
(на этот раз 21 вместо 11)
accuracies = []                  ← Создает пустой список для хранения
for t in thresholds:             значений достоверности
    acc = accuracy_score(y_val, y_pred >= t) ← Вычисляет достоверность
    accuracies.append(acc)          ← для заданного порога
                                    ← Записывает достоверность для этого порога
```

А затем мы выводим эти значения с помощью Matplotlib:

```
plt.plot(thresholds, accuracies)
```

Когда эта строка будет выполнена, мы должны увидеть график, показывающий взаимосвязь между порогом и достоверностью (рис. 4.2). Как мы уже знаем, порог 0,5 оказался наилучшим с точки зрения достоверности.



**Рис. 4.2.** Достоверность нашей модели, оцененная при различных пороговых значениях. Наилучшая достоверность достигается при срезе прогнозов на пороге 0,5: если прогноз выше 0,5, мы прогнозируем «отток», в противном случае — «отсутствие оттока»

Итак, наилучший порог составляет 0,5, а наилучшая достижимая достоверность для этой модели составляет 80 %.

В предыдущей главе мы обучали и более простую модель: мы назвали ее `model_small`. Она была основана только на трех переменных: `contract`, `tenure` и `totalcharges`.

Проверим и ее достоверность. Для этого мы сначала сделаем прогнозы на основе проверочного набора данных, после чего вычислим показатель достоверности:

```
val_dict_small = df_val[small_subset].to_dict(orient='records') | Применяет прямое
X_small_val = dv_small.transform(val_dict_small) | кодирование к проверочным данным
y_pred_small = model_small.predict_proba(X_small_val)[:, 1] | Прогнозирует отток
churn_small = y_pred_small >= 0.5 | с помощью малой модели
accuracy_score(y_val, churn_small) ← | Вычисляет достоверность прогнозов
```

Запустив этот код, мы увидим, что достоверность малой модели составляет 76 %. Таким образом, большая модель на 4 % достовернее, чем малая.

Однако это все еще ничего не говорит о том, является ли 80 % (или 76 %) хорошим показателем достоверности.

### 4.1.2. Фиктивная базовая линия

Данный показатель кажется хорошим, однако чтобы определить, действительно ли 80 % — это хорошо, нам следует связать его с чем-то — например, с простой базовой линией, которую легко понять. Одной из таких базовых линий может стать фиктивная модель, которая всегда предсказывает одно и то же значение.

В нашем примере набор данных не сбалансирован, и у нас не так много ушедших пользователей. Таким образом, фиктивная модель всегда может прогнозировать класс большинства — «оттока нет». Другими словами, эта модель всегда будет выдавать `False`, независимо от признаков. Это не очень полезная модель, но ее можно использовать в качестве базовой и сравнить с двумя другими.

Создадим этот базовый прогноз:

```
size_val = len(y_val) ←———— Получает количество клиентов в проверочном наборе
baseline = np.repeat(False, size_val) ←———— Создает массив, содержащий только элементы False
```

Чтобы создать массив с базовыми прогнозами, нам сначала нужно выяснить, сколько элементов в проверочном наборе.

Далее мы создаем массив фиктивных прогнозов — все элементы этого массива принимают значение `False`. Мы делаем это с помощью функции `repeat` из NumPy: она принимает элемент и повторяет его столько раз, сколько мы просим. Более подробную информацию о функции `repeat` и других функциях NumPy можно найти в приложении В.

Теперь мы можем проверить достоверность этого базового прогноза с помощью кода, который использовали ранее:

```
accuracy_score(baseline, y_val)
```

После запуска мы видим 0,738. Это означает, что достоверность базовой модели составляет около 74 % (рис. 4.3).

```
size_val = len(y_val)
baseline = np.repeat(False, size_val)
baseline
array([False, False, False, ..., False, False, False])

accuracy_score(baseline, y_val)
0.7387096774193549
```

**Рис. 4.3.** Базовая линия — «модель», которая всегда прогнозирует одинаковое значение для всех клиентов. Достоверность этой базовой линии составляет 74 %

Как видим, малая модель лучше базовой всего на 2 %, а большая — на 6 %. Если мы вспомним все те трудности, которые нам пришлось преодолеть, чтобы обучить эту большую модель, то 6 % могут показаться не таким уж значительным приростом по сравнению с фиктивным базовым уровнем.

Прогнозирование оттока — сложная задача, и, возможно, это отличное улучшение. Однако это совсем не очевидно, если использовать только показатель достоверности. Согласно ему наша модель лишь немного лучше фиктивной, которая рассматривает всех клиентов как надежных и не пытается удержать никого из них.

Таким образом, нам нужны дополнительные показатели — другие способы измерения качества нашей модели. Эти показатели основаны на матрице ошибок, концепции, которую мы рассмотрим в следующем разделе.

## 4.2. МАТРИЦА ОШИБОК

Несмотря на то что достоверность легко понять, это не всегда лучший показатель. На самом деле иногда она может даже ввести в заблуждение. Мы уже видели, как это происходит: достоверность нашей модели составляет 80 %, и хотя это значение кажется хорошей цифрой, оно всего на 6 % лучше, чем точность фиктивной модели, которая всегда выдает один и тот же прогноз «без оттока».

Такая ситуация обычно возникает, когда у нас наблюдается дисбаланс классов (больше экземпляров одного класса, чем другого). Мы знаем, что это определенно относится и к нашей задаче: 74 % клиентов не подверглись оттоку и только 26 % — подверглись.

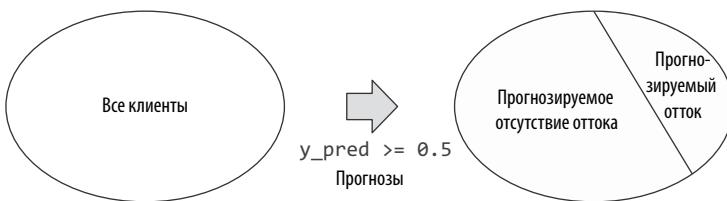
Такие случаи требуют другого способа измерения качества наших моделей. Для этого имеется несколько вариантов, и большинство из них основаны на матрице ошибок: таблице, которая в сжатой форме представляет все возможные результаты для прогнозов нашей модели.

### 4.2.1. Введение в матрицу ошибок

Мы знаем, что для модели бинарной классификации у нас может быть только два возможных предсказания: `True` и `False`. В нашем случае мы можем предсказать, что клиент либо собирается уйти (`True`), либо нет (`False`).

Применяя модель ко всему проверочному набору данных, мы разделяем его на две части (рис. 4.4):

- клиенты, для которых модель предсказывает «отток»;
- клиенты, для которых модель дает прогноз «без оттока».

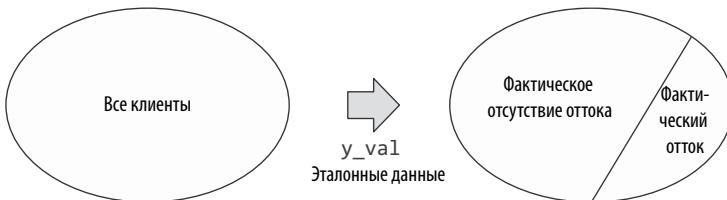


**Рис. 4.4.** Наша модель разделяет всех клиентов в проверочном наборе данных на две группы: тех, которые, по нашему мнению, уйдут, и тех, которые этого не сделают

Возможны только два корректных результата: опять же, `True` или `False`. Клиент либо действительно ушел (`True`), либо нет (`False`).

Это означает, что, используя эталонные данные — информацию о целевой переменной, — мы можем снова разделить набор данных на две части (рис. 4.5):

- клиенты, которые ушли;
- клиенты, которые не ушли.



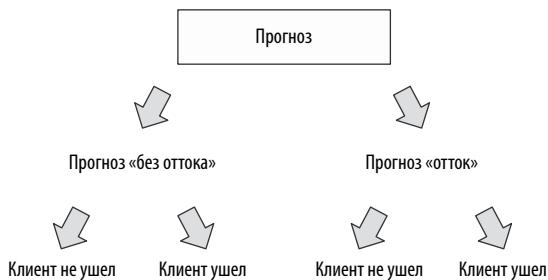
**Рис. 4.5.** Используя эталонные данные, мы можем разделить проверочный набор данных на две группы: клиенты, которые действительно ушли, и клиенты, которые этого не сделали

Когда мы делаем прогноз, он оказывается либо верным, либо нет:

- если мы прогнозируем «отток», то клиент может действительно уйти, а может и остаться;
- если мы прогнозируем «без оттока», то вполне вероятно, что клиент действительно не уйдет, но возможно и то, что в действительности он расторгнет договор.

Таким образом, мы получаем четыре возможных результата (рис. 4.6):

- мы предсказываем `False`, и ответ оказывается `False`;
- мы предсказываем `False`, и ответ оказывается `True`;
- мы прогнозируем `True`, и ответ оказывается `False`;
- мы прогнозируем `True`, и ответ оказывается `True`.



**Рис. 4.6.** Возможны четыре исхода: мы прогнозируем «отток», и клиенты либо уходят, либо нет, и мы прогнозируем «без оттока», и клиенты снова либо уходят, либо нет

Две из этих ситуаций — первая и последняя — хорошие: прогноз соответствовал фактическому значению. Две оставшиеся плохие: мы получили неправильный прогноз.

Каждый из этих четырех результатов имеет собственное название (рис. 4.7):

- истинно отрицательный (TN): мы предсказываем `False` («без оттока»), и фактическая метка тоже `False` («без оттока»);
- истинно положительный (TP): мы прогнозируем `True` («отток»), и фактическая метка тоже `True` («отток»);
- ложноотрицательный (FN): мы прогнозируем `False` («без оттока»), но на самом деле это `True` (клиент ушел);
- ложноположительный (FP): мы прогнозируем `True` («отток»), но на самом деле это `False` (клиент остался с нами).



**Рис. 4.7.** Каждый из четырех возможных исходов имеет собственное название: истинно отрицательный, ложноотрицательный, ложноположительный и истинно положительный

Чтобы представлять эти результаты более наглядно, полезно расположить их в таблице. Мы можем поместить предсказанные классы (`False` и `True`) в столбцы, а фактические классы (`False` и `True`) — в строки (рис. 4.8).

		Прогнозы	
		False (``без оттока``)	True (``отток``)
Фактическое	False (``без оттока``)	TN	FP
	True (``отток``)	FN	TP

**Рис. 4.8.** Мы можем организовать результаты в виде таблицы: прогнозируемые значения — в виде столбцов, а фактические — в виде строк. Таким образом, мы разбиваем все сценарии прогнозирования на четыре отдельные группы: TN (истинно отрицательный), TP (истинно положительный), FN (ложноотрицательный) и FP (ложноположительный)

Подставив количество раз, когда происходит тот или иной результат, мы получаем матрицу ошибок для нашей модели (рис. 4.9).

		Прогнозы	
		False (``без оттока``)	True (``отток``)
Фактическое	False (``без оттока``)	1202	172
	True (``отток``)	197	289

**Рис. 4.9.** В матрице ошибок каждая ячейка содержит количество раз, когда происходит каждый результат

Вычислить значения в ячейках матрицы ошибок довольно просто с помощью NumPy. Далее мы узнаем, как это сделать.

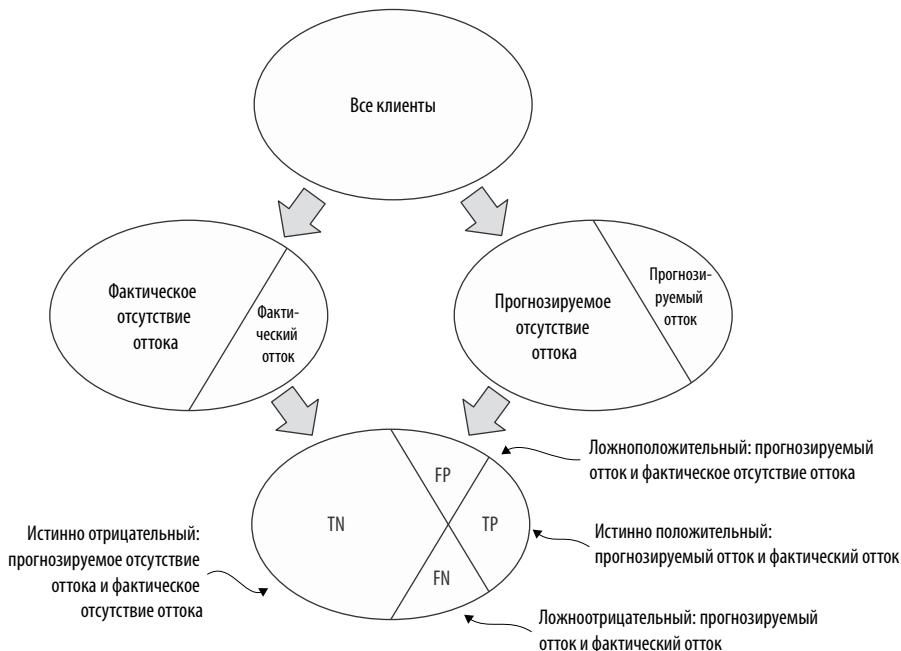
## 4.2.2. Вычисление матрицы ошибок с помощью NumPy

Чтобы лучше понять нашу матрицу ошибок, мы можем визуально изобразить, что она делает с проверочным набором данных (рис. 4.10).

Чтобы рассчитать матрицу ошибок, нам нужно проделать следующие шаги:

- сначала прогнозы разделяют набор данных на две части: часть, для которой мы прогнозируем True (``отток``), и часть, для которой мы прогнозируем False (``без оттока``);

- в то же время целевая переменная разбивает этот набор данных на две различные части: клиентов, которые действительно ушли (1 в  $y_{val}$ ), и клиентов, которые этого не сделали (0 в  $y_{val}$ );
- объединив эти части, мы получаем четыре группы клиентов, которые в точности соответствуют четырем различным результатам из матрицы ошибок.



**Рис. 4.10.** Применяя модель к набору данных проверки, мы получаем четыре различных результата (TN, FP, TP и FN)

Перевести эти шаги в NumPy очень просто:

```
t = 0.5
predict_churn = (y_pred >= t)
predict_no_churn = (y_pred < t)

actual_churn = (y_val == 1)
actual_no_churn = (y_val == 0)

true_positive = (predict_churn & actual_churn).sum()
false_positive = (predict_churn & actual_no_churn).sum()

false_negative = (predict_no_churn & actual_churn).sum()
true_negative = (predict_no_churn & actual_no_churn).sum()
```

1 Делает прогнозы при пороговом значении 0,5

2 Получает фактические целевые значения

3 Вычисляет истинно положительные результаты (случаи, в которых мы правильно спрогнозировали отток)

4 Вычисляет ложноположительные результаты (случаи, когда мы прогнозировали отток, но его не произошло)

5 Вычисляет ложноотрицательные результаты (случаи, когда мы прогнозировали «без оттока», но клиенты ушли)

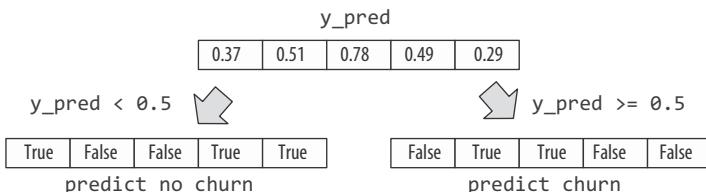
6 Вычисляет истинно отрицательные значения (случаи, когда мы правильно спрогнозировали «без оттока»)

Мы начинаем с того, что получаем прогнозы при пороге 0,5 ❶.

Результатом служат два массива NumPy:

- в первом (`predict_churn`) элемент имеет значение `True`, если модель считает, что соответствующий клиент собирается расторгнуть контракт, и `False` в противном случае;
- аналогично во втором массиве (`predict_no_churn`) значение `True` означает, что модель считает, что клиент не собирается уходить.

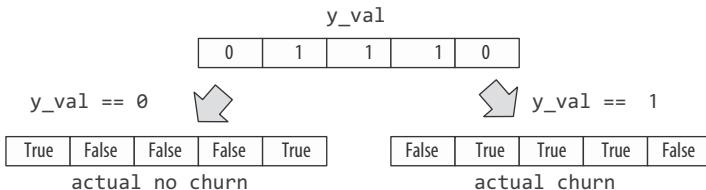
Второй массив, `predict_no_churn`, полностью противоположен первому, `predict_churn`: если элемент имеет значение `True` в `predict_churn`, то он имеет значение `False` в `predict_no_churn`, и наоборот (рис. 4.11). Это первое разделение набора данных проверки на две части на основе прогнозов.



**Рис. 4.11.** Разделение прогнозов на два логических массива NumPy: `predict_churn`, если вероятность выше 0,5, и `predict_no_churn`, если ниже

Далее мы записываем фактические значения целевой переменной в ❷. В результате также получаются два массива NumPy (рис. 4.12):

- если клиент ушел в результате оттока (значение 1), то соответствующий элемент `actual_churn` получает значение `True`, в противном случае — `False`;
- для `actual_no_churn` все с точностью до наоборот: `True`, когда клиент не ушел.



**Рис. 4.12.** Разделение массива с фактическими значениями на два логических массива NumPy: `actual_no_churn`, если клиент не ушел (`y_val == 0`), и `actual_churn`, если ушел (`y_val == 1`)

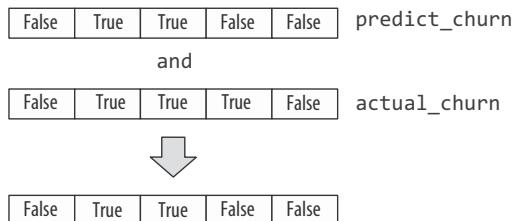
Это второе разделение набора данных, основанное на целевой переменной.

Теперь мы объединяем эти два разделения — или, если быть точным, четыре массива NumPy.

Чтобы вычислить количество истинно положительных результатов в ❸, мы используем логический оператор «И» в NumPy (&) и метод sum:

```
true_positive = (predict_churn & actual_churn).sum()
```

Логический оператор «И» принимает значение True только в том случае, если оба значения истинны. Если хотя бы одно из них (или оба) False, то результатом становится False. В случае true\_positive значение True будет присвоено только в том случае, если мы прогнозируем «отток» и клиент действительно ушел (рис. 4.13).



**Рис. 4.13.** Применение поэлементного оператора «и» (&) к двум массивам NumPy, predict\_churn и actual\_churn. В итоге создается новый массив с True в позициях, где оба массива содержали True, и с False во всех остальных

Затем мы используем метод sum из NumPy, который просто подсчитывает, сколько значений True содержится в массиве. Сначала массив Boolean преобразуется в целочисленный, а затем суммируется (рис. 4.14). Мы уже встречали подобное поведение в предыдущей главе, когда использовали метод mean.



**Рис. 4.14.** Вызов метода sum для массива Boolean: мы получаем количество элементов в массиве, которые содержат True

В результате мы получаем количество истинно положительных случаев. Остальные значения вычисляются аналогично в строках ❹, ❺, ❻.

Теперь нам просто требуется объединить все эти значения в массив NumPy:

```
confusion_table = np.array([
    [true_negative, false_positive],
    [false_negative, true_positive]])
```

Выведя его на экран, мы наблюдаем следующие цифры:

```
[[1202, 172],  
 [ 197, 289]]
```

Абсолютные числа бывает трудно понять, поэтому мы можем преобразовать их в дроби, разделив каждое значение на общее количество элементов:

```
confusion_table / confusion_table.sum()
```

При этом выводится следующее:

```
[[0.646, 0.092],  
 [0.105, 0.155]]
```

Мы можем обобщить результаты в таблице (табл. 4.1). Мы видим, что модель довольно хорошо прогнозирует отрицательные значения: 65 % прогнозов — истинно отрицательные. Однако она допускает довольно много ошибок обоих типов: количество ложноположительных и ложноотрицательных результатов примерно одинаково (9 и 11 % соответственно).

**Таблица 4.1.** Матрица ошибок для классификатора оттока при пороговом значении 0,5. Мы видим, что модель с легкостью правильно прогнозирует пользователей «без оттока», но «отток» ей идентифицировать уже сложнее

		Полная модель со всеми признаками	
		Прогнозируемое	
		False	True
Фактическое	False	1202 (65 %)	172 (9 %)
	True	197 (11 %)	289 (15 %)

Таблица дает нам лучшее представление о производительности модели — теперь легко разбить производительность на различные компоненты и понять, где модель допускает ошибки. На самом деле мы видим, что производительность модели не слишком велика: она допускает довольно много ошибок при попытке идентифицировать пользователей, которые планируют уйти («отток»). Это то, чего мы не смогли бы узнать, используя только показатель достоверности.

Мы можем повторить тот же процесс и для малой модели, используя точно такой же код (табл. 4.2).

**Таблица 4.2.** Матрица ошибок для малой модели

		Малая модель с тремя признаками	
		Прогнозируемое	
		False	True
Фактическое	False	1189 (63 %)	185 (10 %)
	True	248 (12 %)	238 (13 %)

Сравнивая малую модель с полной, мы видим, что она на 2 % хуже распознает пользователей «без оттока» (63 против 65 % для истинно отрицательных результатов), и на 2 % хуже распознает пользователей, испытывающих «отток» (13 против 15 % для истинно положительных результатов), что в совокупности составляет 4 % разницы в достоверности этих двух моделей (76 против 80 %).

Значения из матрицы ошибок служат основой для многих других показателей оценки. Например, мы можем рассчитать достоверность, взяв все верные прогнозы (TN и TP) и разделив полученное число на общее количество наблюдений во всех четырех ячейках таблицы:

$$\text{достоверность} = (\text{TN} + \text{TP}) / (\text{TN} + \text{TP} + \text{FN} + \text{FP})$$

Помимо достоверности, мы можем рассчитать другие показатели на основе значений из матрицы ошибок. Наиболее полезными из них будут точность и отклик, о которых мы расскажем далее.

### **Упражнение 4.1**

Что такое ложноположительный результат?

- A. Клиент, для которого мы прогнозировали «без оттока», но который прекратил пользоваться нашими услугами.
- B. Клиент, для которого мы прогнозировали «отток», но он не произошел.
- C. Клиент, для которого мы прогнозировали «отток», и он произошел.

### **4.2.3. Точность и отклик**

Как мы уже говорили, достоверность может ввести в заблуждение при работе с несбалансированными наборами данных, такими как наши. В подобных случаях полезно использовать другие показатели: точность и отклик.

Как точность, так и отклик рассчитываются на основе значений матрицы ошибок. Оба показателя помогают нам оценить качество модели в случаях дисбаланса классов.

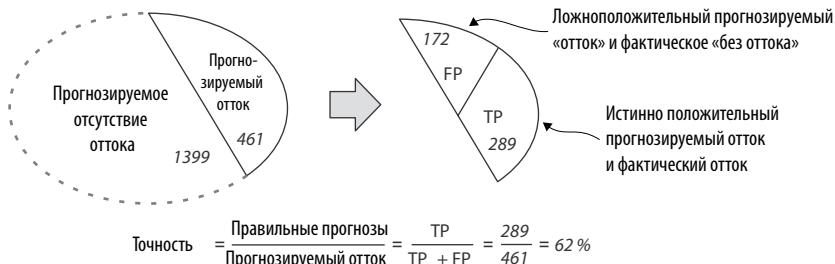
Начнем с точности модели. Данный показатель сообщает нам, сколько положительных прогнозов оказались верными. Это доля правильно спрогнозированных положительных примеров.

В нашем случае это количество клиентов, которые фактически ушли (TP) по сравнению с количеством, которое должно было уйти (согласно нашей модели), то есть (TP + FP) (рис. 4.15):

$$P = \text{TP} / (\text{TP} + \text{FP})$$

Для нашей модели точность составляет 62 %:

$$P = 289 / (289 + 172) = 172 / 461 = 0,62$$



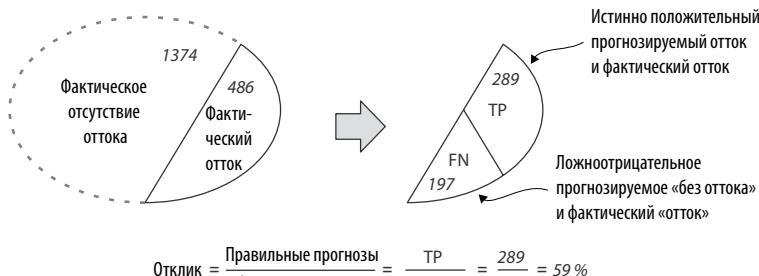
**Рис. 4.15.** Точность модели представляет собой долю правильных прогнозов (TP) среди всех положительных прогнозов (TP + FP)

Отклик — это доля правильно классифицированных положительных примеров среди всех положительных примеров. В нашем случае, чтобы рассчитать отклик, мы сначала выбираем всех клиентов, которые ушли, и смотрим, скольких из них нам удалось правильно определить.

Формула для расчета отклика такова:

$$R = TP / (TP + FN)$$

Как и в формуле точности, числитель представляет собой количество истинно положительных результатов, но знаменатель другой: это количество всех положительных примеров (`y_val == 1`) в нашем проверочном наборе данных (рис. 4.16).



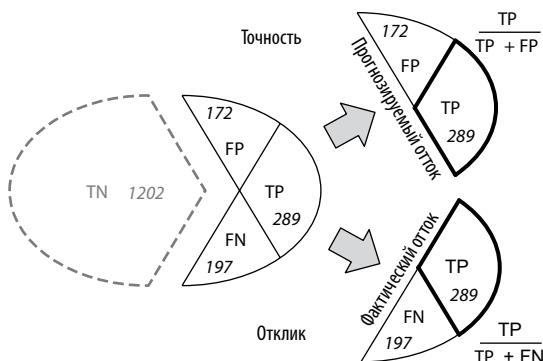
**Рис. 4.16.** Отклик модели — это доля правильно спрогнозированных клиентов с оттоком (TP) среди всех клиентов, которые ушли (TP + FN)

Для нашей модели отклик составляет 59 %:

$$R = 286 / (289 + 197) = 289 / 486 = 0,59$$

Разница между точностью и откликом на первый взгляд может показаться неуловимой. В обоих случаях мы смотрим на количество правильных прогнозов, но разница кроется в знаменателях (рис. 4.17):

- точность — каков процент правильных прогнозов (TP) среди клиентов с прогнозом «отток» ( $TP + FP$ )?
- отклик — каков процент правильных прогнозов «оттока» (TP) среди всех клиентов с «оттоком» ( $TP + FN$ )?



**Рис. 4.17.** Как точность, так и отклик указывают на верные прогнозы (TP), но знаменатели разные. Для точности это количество клиентов, прогнозируемое как отток, в то время как для отклика это количество клиентов, которые ушли на самом деле

Данный рисунок также позволяет нам увидеть, что ни точность, ни отклик не учитывают истинно отрицательные результаты. Именно поэтому они и являются хорошими оценочными показателями для несбалансированных наборов данных. В ситуациях с дисбалансом классов истинно отрицательных результатов обычно больше, чем остальных, но в то же время они также часто не представляют для нас особого интереса. Посмотрим, каковы причины этому.

Цель нашего проекта — выявить клиентов, которые, скорее всего, собираются уходить. Как только мы это сделаем, мы сможем отправить им рекламные сообщения в надежде, что они изменят свой настрой.

Делая это, мы можем допустить два типа ошибок:

- во-первых, мы можем случайно отправить сообщения людям, которые не собирались уходить, — эти люди относятся к ложноположительным результатам модели;
- во-вторых, нам иногда не удается идентифицировать людей, которые действительно собираются уйти. Мы не отправляем им сообщения — эти люди относятся к нашим ложноотрицательным результатам.

Точность и отклик помогают нам количественно оценить эти ошибки.

Точность помогает понять, сколько людей получили рекламное сообщение по ошибке. Чем выше точность, тем меньше у нас ложных срабатываний.

Точность 62 % означает, что 62 % охваченных клиентов действительно собирались отказаться (наши истинно положительные результаты), в то время как остальные 38 % — нет (ложноположительные результаты).

Отклик помогает понять, скольких из уходящих клиентов нам не удалось обнаружить. Чем лучше отклик, тем меньше у нас ложноотрицательных результатов. Отклик 59 % означает, что мы охватываем только 59 % всех пользователей, которые планируют уйти (истинно положительные результаты), и не в силах обнаружить остальные 41 % (ложноотрицательные результаты).

Очевидно, что в обоих случаях нам действительно не нужно количество истинно отрицательных результатов: даже если удастся правильно определить, что люди не уходят, мы не планируем ничего предпринимать по этому поводу.

Хотя достоверность 80 % может свидетельствовать о том, что модель великолепна, анализ ее точности и отклика говорит нам о том, что в действительности она допускает довольно много ошибок. Обычно это не является препятствием: при машинном обучении модели неизбежно допускают ошибки, но теперь у нас хотя бы есть лучшее и реалистичное понимание производительности нашей модели прогнозирования оттока.

Точность и отклик — полезные показатели, но они описывают производительность классификатора только при определенном пороге. Часто бывает полезно иметь метрику, которая суммирует производительность классификатора для всех возможных пороговых значений. Мы рассмотрим такие показатели в следующем разделе.

### **Упражнение 4.2**

Что такое точность?

- А. Процент правильно идентифицированных ушедших клиентов («отток») в проверочном наборе данных.
- Б. Процент клиентов, которые действительно ушли, среди клиентов, которые, как мы спрогнозировали, собирались уходить.

### **Упражнение 4.3**

Что такое отклик?

- А. Процент правильно идентифицированных ушедших клиентов среди всех ушедших клиентов.
- Б. Процент правильно классифицированных клиентов среди клиентов, которые, как мы спрогнозировали, собирались уходить.

## 4.3. КРИВАЯ ROC И ОЦЕНКА AUC

Показатели, которые мы рассматривали до сих пор, работают лишь с двоичными прогнозами — когда в выходных данных у нас только значения `True` и `False`. Тем не менее есть способы оценить производительность модели при всех возможных пороговых значениях. Кривые ROC — один из таких способов.

ROC расшифровывается как «рабочие характеристики приемника» (*receiver operating characteristic*), и изначально этот показатель был разработан для оценки надежности радаров во время Второй мировой войны. Он использовался для оценки того, насколько хорошо детектор может разделить два сигнала: определен самолет или нет. В настоящее время ROC служит для аналогичной цели: показывает, насколько хорошо модель может разделять два класса, положительный и отрицательный. В нашем случае это классы «отток» и «без оттока».

Нам нужны две метрики для кривых ROC: TPR (true positive rate, доля истинно положительных результатов) и FPR (false positive rate, доля ложноположительных результатов). Познакомимся с ними поближе.

### 4.3.1. Доля истинно положительных и ложноположительных результатов

Кривая ROC основана на двух величинах, FPR и TPR:

- доля ложноположительных результатов (FPR) — доля ложноположительных результатов среди всех отрицательных;
- доля истинно положительных результатов (TPR) — доля истинно положительных результатов среди всех положительных примеров.

Как и в случае точности с отзывом, эти значения основаны на матрице ошибок. Мы можем рассчитать их, используя следующие формулы:

$$\text{FPR} = \text{FP} / (\text{FP} + \text{TN})$$

$$\text{TPR} = \text{TP} / (\text{TP} + \text{FN})$$

FPR и TPR включают в себя две отдельные части таблицы ошибок (рис. 4.18):

- для FPR мы смотрим на первую строку таблицы: доля ложноположительных результатов среди всех отрицательных;
- для TPR мы смотрим на вторую строку: доля истинно положительных результатов среди всех положительных.

		Прогнозы	
		False «без оттока»)	True «отток»)
Фактическое	False «без оттока»)	TN	FP
	True «отток»)	FN	TP

$$FPR = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{TP + FN}$$

**Рис. 4.18.** Для вычисления FPR мы смотрим на первую строку матрицы ошибок, а для вычисления TPR — на вторую

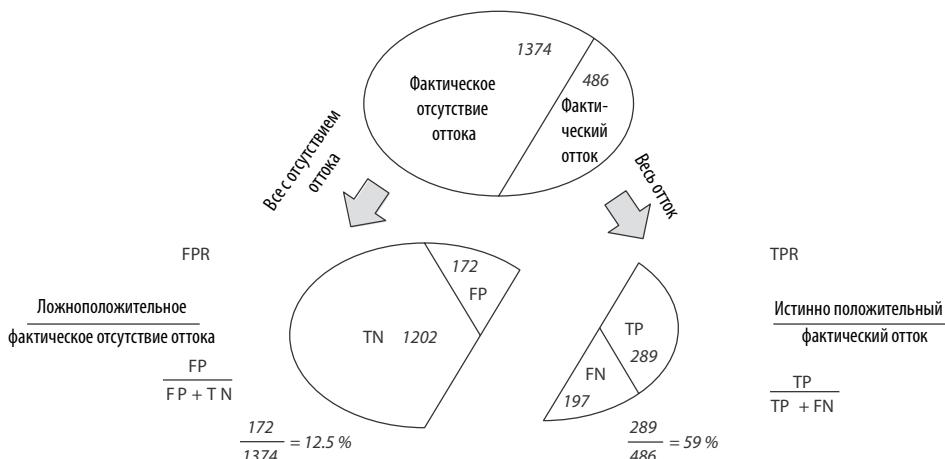
Рассчитаем указанные значения для нашей модели (рис. 4.19):

$$FPR = 172 / 1374 = 12,5 \%$$

FPR — это доля пользователей, для которой мы спрогнозировали отток, среди всех, кто не подвергся оттоку. Небольшое значение FPR говорит о том, что модель хороша — у нее оказалось мало ложных срабатываний:

$$TPR = 289 / 486 = 59 \%$$

TPR — это доля пользователей, для которых мы прогнозировали отток, среди всех, кто действительно расторг договор. Обратите внимание, что TPR совпадает с откликом, поэтому чем выше TPR, тем лучше.



**Рис. 4.19.** FPR — доля ложноположительных результатов среди всех клиентов «без оттока»: чем меньше FPR, тем лучше. TPR — доля истинно положительных результатов среди всех ушедших клиентов («отток»): чем больше TPR, тем лучше

Однако мы по-прежнему рассматриваем показатели FPR и TPR только при одном пороговом значении, и в нашем случае это 0,5. Чтобы иметь возможность использовать их для кривых ROC, нам нужно рассчитать эти показатели для множества различных пороговых значений.

### 4.3.2. Оценка модели при нескольких пороговых значениях

Модели бинарной классификации, такие как логистическая регрессия, обычно выдают вероятность — оценку от нуля до единицы. Чтобы сделать реальные прогнозы, мы бинаризуем выходные данные, устанавливая некоторый порог, чтобы получать только значения `True` и `False`.

Вместо того чтобы оценивать модель по одному конкретному порогу, мы можем сделать это для целого ряда — точно так же, как мы делали это для достоверности ранее в текущей главе.

Для этого мы сначала перебираем различные пороговые значения и вычисляем значения матрицы ошибок для каждого из них (листинг 4.1).

**Листинг 4.1.** Вычисление матрицы ошибок для различных пороговых значений

```
scores = [] ← Создает список, в котором мы будем
                  сохранять результаты
thresholds = np.linspace(0, 1, 101) ← Создает массив с различными пороговыми
                                         значениями и перебирает их
for t in thresholds:
    tp = ((y_pred >= t) & (y_val == 1)).sum() ← Вычисляет матрицу ошибок
    fp = ((y_pred >= t) & (y_val == 0)).sum() ← для прогнозов при каждом
    fn = ((y_pred < t) & (y_val == 1)).sum() ← пороге
    tn = ((y_pred < t) & (y_val == 0)).sum()
    scores.append((t, tp, fp, fn, tn)) ← Добавляет результаты
                                         в список оценок
```

Идея аналогична тому, что мы ранее проделали с достоверностью, но вместо записи только одного значения мы записываем в матрицу ошибок все четыре результата.

Работать со списком кортежей не слишком просто, поэтому преобразуем его в датафрейм Pandas:

```
df_scores = pd.DataFrame(scores) ← Превращает список
                                         в датафрейм Pandas
df_scores.columns = ['threshold', 'tp', 'fp', 'fn', 'tn'] ← Присваивает
                                         имена столбцам
                                         датафрейма
```

Так мы получаем датафрейм с пятью столбцами (рис. 4.20).

threshold	tp	fp	fn	tn	
0	0.0	486	1374	0	0
10	0.1	458	726	28	648
20	0.2	421	512	65	862
30	0.3	380	350	106	1024
40	0.4	337	257	149	1117
50	0.5	289	172	197	1202
60	0.6	200	105	286	1269
70	0.7	99	34	387	1340
80	0.8	7	1	479	1373
90	0.9	0	0	486	1374
100	1.0	0	0	486	1374

threshold	tp	fp	fn	tn	
0	0.0	486	1374	0	0
10	0.1	458	726	28	648
20	0.2	421	512	65	862
30	0.3	380	350	106	1024
40	0.4	337	257	149	1117
50	0.5	289	172	197	1202
60	0.6	200	105	286	1269
70	0.7	99	34	387	1340
80	0.8	7	1	479	1373
90	0.9	0	0	486	1374
100	1.0	0	0	486	1374

**Рис. 4.20.** Датафрейм с элементами матрицы ошибок, рассчитанной при разных пороговых значениях. Выражение `[::10]` выбирает каждую десятую запись датафрейма

Теперь мы можем вычислить показатели TPR и FPR. Поскольку данные теперь находятся в датафрейме, мы можем сделать это для всех значений сразу:

```
df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)
```

Когда код будет выполнен, у нас появятся два новых столбца в датафрейме: `tpr` и `fpr` (рис. 4.21).

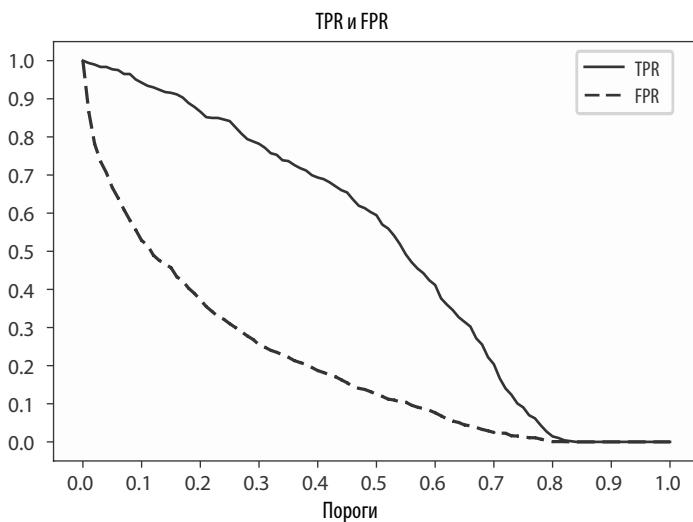
threshold	tp	fp	fn	tn	tpr	fpr	
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	458	726	28	648	0.942387	0.528384
20	0.2	421	512	65	862	0.866255	0.372635
30	0.3	380	350	106	1024	0.781893	0.254731
40	0.4	337	257	149	1117	0.693416	0.187045
50	0.5	289	172	197	1202	0.594650	0.125182
60	0.6	200	105	286	1269	0.411523	0.076419
70	0.7	99	34	387	1340	0.203704	0.024745
80	0.8	7	1	479	1373	0.014403	0.000728
90	0.9	0	0	486	1374	0.000000	0.000000
100	1.0	0	0	486	1374	0.000000	0.000000

threshold	tp	fp	fn	tn	tpr	fpr	
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	458	726	28	648	0.942387	0.528384
20	0.2	421	512	65	862	0.866255	0.372635
30	0.3	380	350	106	1024	0.781893	0.254731
40	0.4	337	257	149	1117	0.693416	0.187045
50	0.5	289	172	197	1202	0.594650	0.125182
60	0.6	200	105	286	1269	0.411523	0.076419
70	0.7	99	34	387	1340	0.203704	0.024745
80	0.8	7	1	479	1373	0.014403	0.000728
90	0.9	0	0	486	1374	0.000000	0.000000
100	1.0	0	0	486	1374	0.000000	0.000000

**Рис. 4.21.** Датафрейм со значениями матрицы ошибок, а также TPR и FPR, вычисленные при разных пороговых значениях

Построим их график (рис. 4.22):

```
plt.plot(df_scores.threshold, df_scores.tpr, label='TPR')
plt.plot(df_scores.threshold, df_scores.fpr, label='FPR')
plt.legend()
```



**Рис. 4.22.** TPR и FPR для нашей модели, оцененные при разных пороговых значениях

Как TPR, так и FPR начинаются со 100 % — при пороге 0,0 мы прогнозируем «отток» для всех:

- FPR равен 100 %, поскольку в нашем прогнозе только ложноположительные результаты. Истинно отрицательных нет: ни для кого нет прогноза «отток»;
- TPR равен 100 %, поскольку у нас есть только истинно положительные результаты и ни одного ложноотрицательного;

По мере увеличения порога оба показателя снижаются, но с разной скоростью.

В идеале FPR должен снижаться очень быстро. Небольшой FPR указывает на то, что модель допускает очень мало ошибок, прогнозируя отрицательные примеры (ложноположительные результаты).

С другой стороны, TPR должен снижаться медленно, в идеале оставаясь все время около 100 %: это будет означать, что модель хорошо прогнозирует истинно положительные результаты.

Чтобы лучше понять показатели TPR и FPR, сравним их с двумя базовыми моделями: случайной и идеальной. Начнем со случайной.

### 4.3.3. Случайная базовая модель

Случайная модель выдает случайный результат от 0 до 1, независимо от входных данных. Это легко реализовать — мы просто генерируем массив с однородными случайными числами:

```
np.random.seed(1) ← Фиксирует случайное начальное значение для воспроизводимости
y_rand = np.random.uniform(0, 1, size=len(y_val)) ← Генерирует массив со случайными числами между 0 и 1
```

Теперь мы можем просто сделать вид, что `y_rand` содержит предсказания нашей «модели».

Рассчитаем FPR и TPR для получившейся случайной модели. Чтобы упростить задачу, мы повторно используем код, который написали ранее, и поместим его в функцию (листинг 4.2).

**Листинг 4.2.** Функция вычисления TPR и FPR при различных пороговых значениях

```
def tpr_fpr_dataframe(y_val, y_pred):
    scores = [] | Определяет функцию, которая
                 принимает фактические
                 и прогнозируемые значения

    thresholds = np.linspace(0, 1, 101) | Вычисляет
                                         матрицу ошибок
                                         для различных
                                         пороговых
                                         значений

    for t in thresholds:
        tp = ((y_pred >= t) & (y_val == 1)).sum()
        fp = ((y_pred >= t) & (y_val == 0)).sum()
        fn = ((y_pred < t) & (y_val == 1)).sum()
        tn = ((y_pred < t) & (y_val == 0)).sum()
        scores.append((t, tp, fp, fn, tn)) | Преобразует числа
                                              из матрицы ошибок
                                              в датафрейм

    df_scores = pd.DataFrame(scores)
    df_scores.columns = ['threshold', 'tp', 'fp', 'fn', 'tn'] | Вычисляет
                                                               TPR и FPR,
                                                               используя
                                                               числа матрицы
                                                               ошибок

    df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
    df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn) | Возвращает итоговый датафрейм

    return df_scores ← Возвращает итоговый датафрейм
```

Теперь используем эту функцию для вычисления TPR и FPR случайной модели:

```
df_rand = tpr_fpr_dataframe(y_val, y_rand)
```

Код создает датафрейм со значениями TPR и FPR при разных пороговых значениях (рис. 4.23).

```
np.random.seed(1)
y_rand = np.random.uniform(0, 1, size=len(y_val))
df_rand = tpr_fpr_dataframe(y_val, y_rand)
df_rand[:10]
```

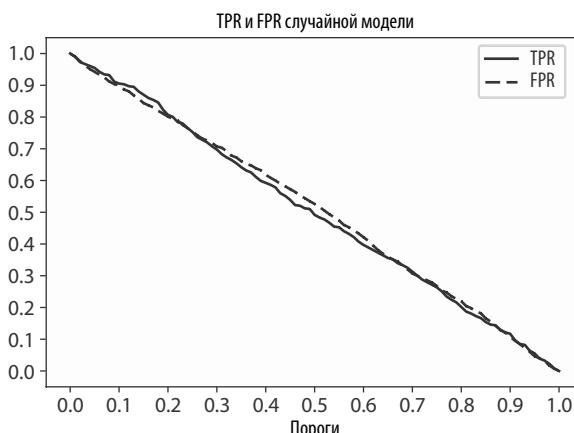
threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000
10	0.1	440	1236	46	138	0.905350
20	0.2	392	1101	94	273	0.806584
30	0.3	339	972	147	402	0.697531
40	0.4	288	849	198	525	0.592593
50	0.5	239	723	247	651	0.491770
60	0.6	193	579	293	795	0.397119
70	0.7	152	422	334	952	0.312757
80	0.8	98	302	388	1072	0.201646
90	0.9	57	147	429	1227	0.117284
100	1.0	0	0	486	1374	0.000000

**Рис. 4.23.** Значения TPR и FPR случайной модели

Теперь построим график:

```
plt.plot(df_rand.threshold, df_rand.tpr, label='TPR')
plt.plot(df_rand.threshold, df_rand.fpr, label='FPR')
plt.legend()
```

Мы видим, что кривые TPR и FPR идут от 100 до 0 %, почти следуя прямой линии (рис. 4.24).



**Рис. 4.24.** Как TPR, так и FPR случайного классификатора уменьшаются со 100 до 0 % в виде прямой линии

На пороге 0,0 мы подразумеваем «отток» во всех случаях. Как TPR, так и FPR равны 100 %:

- FPR равен 100 %, поскольку у нас есть только ложноположительные результаты: для всех клиентов «без оттока» прогнозируется «отток»;
- TPR составляет 100 %, поскольку у нас есть только истинно положительные результаты: мы можем правильно спрогнозировать «отток» для всех клиентов с «оттоком».

По мере увеличения порога как TPR, так и FPR уменьшаются.

При пороге 0,4 модель с вероятностью 40 % дает прогноз «без оттока», а с вероятностью 60 % прогнозирует «отток». Как TPR, так и FPR равны 60 %:

- FPR составляет 60 %, поскольку мы неправильно прогнозируем «отток» для 60 % фактически оставшихся клиентов;
- TPR составляет 60 %, поскольку мы правильно классифицируем 60 % ушедших клиентов как клиентов с «оттоком».

Наконец, при значении 1,0 как TPR, так и FPR равны 0 %. На этом пороге мы прогнозируем «без оттока» для всех:

- FPR равен 0 %, поскольку у нас нет ложноположительных результатов: мы можем правильно спрогнозировать «без оттока» для всех клиентов «без оттока»;
- TPR равен 0 %, поскольку у нас нет истинно положительных результатов: для всех клиентов с «оттоком» прогноз показывает «без оттока».

Теперь перейдем к следующей базовой линии и посмотрим, как TPR и FPR выглядят для идеальной модели.

#### **4.3.4. Идеальная модель**

Идеальная модель всегда принимает правильные решения. Мы сделаем еще один шаг вперед и рассмотрим идеальную модель ранжирования. Эта модель выводит оценки таким образом, что клиенты с «оттоком» всегда имеют более высокие баллы «без оттока». Другими словами, прогнозируемая вероятность для всех клиентов с «оттоком» должна быть выше, чем прогнозируемая вероятность для клиентов «без оттока».

Итак, если мы применим модель ко всем клиентам в нашем проверочном наборе, а затем отсортируем их по прогнозируемой вероятности, то в начале у нас расположатся все клиенты «без оттока», а затем клиенты, у которых «отток» есть (рис. 4.25).



**Рис. 4.25.** Идеальная модель упорядочивает клиентов таким образом, чтобы в начале располагались клиенты «без оттока», а затем с «оттоком»

Конечно, в реальности такую модель создать невозможно. Тем не менее она для нас полезна: мы можем использовать ее для сравнения наших TPR и FPR с TPR и FPR идеальной модели.

Сгенерируем идеальные прогнозы. Чтобы упростить задачу, мы генерируем массив с поддельными целевыми переменными, которые уже упорядочены: сначала он содержит только 0, а затем только 1 (см. рис. 4.25). Что касается «прогнозов», то мы просто можем создать массив с числами, которые растут от 0 в первой ячейке до 1 в последней, с помощью функции `np.linspace`.

Запустим ее:

```

num_neg = (y_val == 0).sum() | Вычисляет количество отрицательных
num_pos = (y_val == 1).sum() | и положительных примеров в наборе данных

y_ideal = np.repeat([0, 1], [num_neg, num_pos]) <-- Генерирует массив, который сначала
y_pred_ideal = np.linspace(0, 1, num_neg + num_pos) <-- вносит num_neg нулей, после чего
                                                               следуют единицы повторяющиеся
                                                               num_pos количество раз

df_ideal = tpr_fpr_dataframe(y_ideal, y_pred_ideal) <-- Вычисляет кривые TPR
| Генерирует прогнозы «модели»: числа, которые растут | и FPR для классификатора
| от 0 в первой ячейке до 1 в последней

```

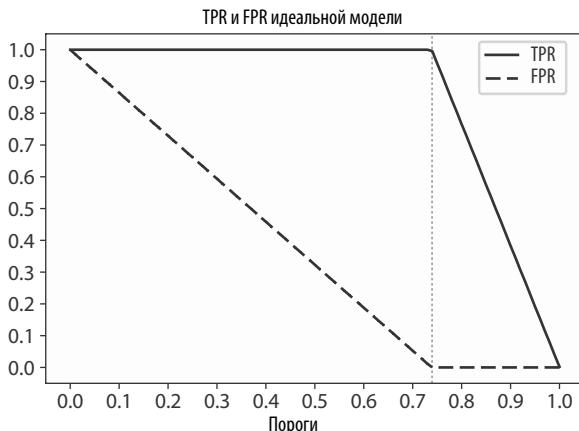
В результате мы получаем датафрейм со значениями TPR и FPR идеальной модели (рис. 4.26). Больше информации о функциях `np.linspace` и `np.repeat` можно узнать в приложении В.

threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000
10	0.1	486	1188	0	186	1.000000
20	0.2	486	1002	0	372	1.000000
30	0.3	486	816	0	558	1.000000
40	0.4	486	630	0	744	1.000000
50	0.5	486	444	0	930	1.000000
60	0.6	486	258	0	1116	1.000000
70	0.7	486	72	0	1302	1.000000
80	0.8	372	0	114	1374	0.765432
90	0.9	186	0	300	1374	0.382716
100	1.0	1	0	485	1374	0.002058

**Рис. 4.26.** Значения TPR и FPR для идеальной модели

Теперь можно построить график (рис. 4.27):

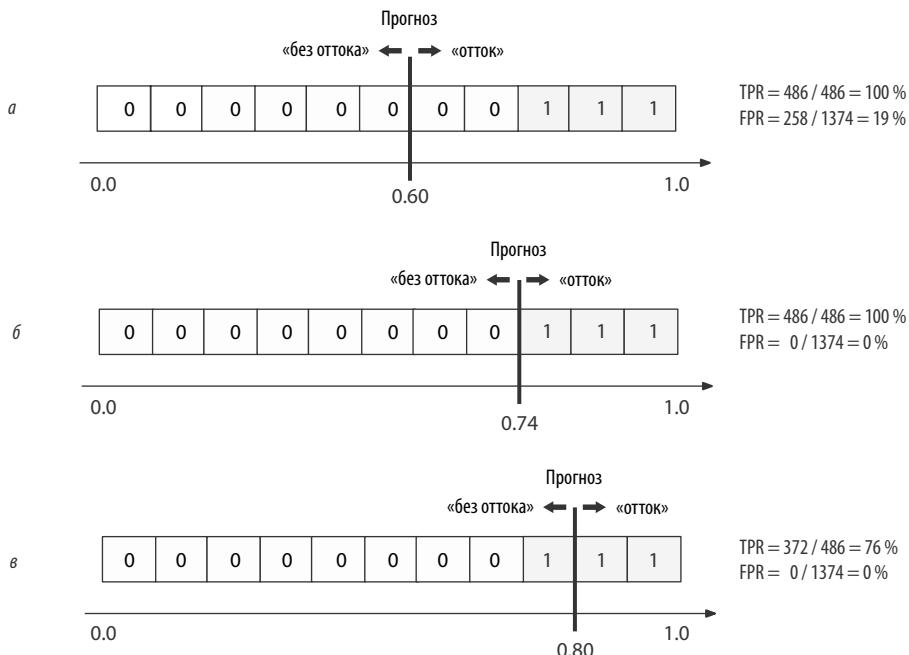
```
plt.plot(df_ideal.threshold, df_ideal.tpr, label='TPR')
plt.plot(df_ideal.threshold, df_ideal.fpr, label='FPR')
plt.legend()
```



**Рис. 4.27.** Кривые TPR и FPR для идеальной модели

Из графика мы видим следующее:

- как TPR, так и FPR начинаются со 100 % и заканчиваются на 0 %;
- для пороговых значений ниже 0,74 мы всегда правильно классифицируем всех клиентов с «оттоком»; вот почему TPR остается на уровне 100 %. С другой стороны, мы неправильно классифицируем некоторых клиентов без «оттока» как с «оттоком» — это наши ложноположительные результаты. По мере того как мы повышаем порог, все меньше и меньше клиентов «без оттока» классифицируются как с «оттоком», поэтому FPR снижается. При значении 0,6 мы ошибочно прогнозируем «отток» для 258 клиентов «без оттока» (рис. 4.28, а);
- пороговое значение 0,74 является идеальной ситуацией: все клиенты с «оттоком» классифицируются как с «оттоком», а все клиенты «без оттока» классифицируются как «без оттока»; вот почему TPR равен 100 %, а FPR равен 0 % (рис. 4.28, б);
- в диапазоне от 0,74 до 1,0 мы всегда правильно классифицируем всех клиентов «без оттока» поэтому FPR остается на уровне 0 %. Однако по мере увеличения порога мы начинаем неправильно классифицировать все больше и больше клиентов с «оттоком» как «без оттока», поэтому TPR снижается. При значении 0,8 114 из 446 клиентов с «оттоком» ошибочно классифицируются как «без оттока». Только 372 предсказания верны, поэтому TPR составляет 76 % (рис. 4.28, в).



**Рис. 4.28.** TPR и FPR идеальной модели ранжирования, вычисленные при различных пороговых значениях

Теперь мы готовы построить кривую ROC.

#### Упражнение 4.4

Что делает идеальная модель ранжирования?

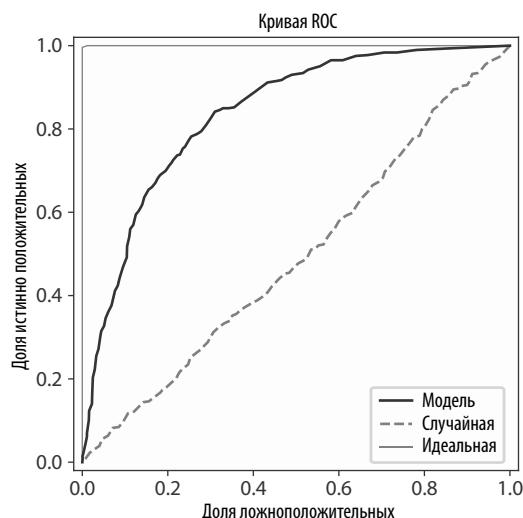
- A. При применении к проверочным данным она оценивает клиентов таким образом, что для клиентов «без оттока» оценка всегда ниже, чем для клиентов с «оттоком».
- B. Она оценивает клиентов «без оттока» выше, чем с «оттоком».

#### 4.3.5. Кривая ROC

Чтобы создать кривую ROC, вместо того, чтобы строить FPR и TPR по разным пороговым значениям, мы строим их друг напротив друга. Для сравнения мы также добавим на график идеальную и случайную модели:

```
plt.figure(figsize=(5, 5)) ←———— Делает график квадратным
plt.plot(df_scores.fpr, df_scores.tpr, label='Model') | Отображает кривую
plt.plot(df_rand.fpr, df_rand.tpr, label='Random') | ROC для модели
plt.plot(df_ideal.fpr, df_ideal.tpr, label='Ideal') | и базовых линий
plt.legend()
```

В результате мы получаем кривую ROC (рис. 4.29). При построении мы можем видеть, что кривая ROC случайного классификатора представляет собой практически прямую линию, начинающуюся снизу слева и идущую вверх направо. Однако в случае идеальной модели кривая сначала идет вверх, пока не достигнет 100 % TPR, а затем вправо, пока не достигнет 100 % FPR.



**Рис. 4.29.** Кривая ROC показывает взаимосвязь между FPR и TPR модели

Наши модели всегда должны располагаться где-то между этими двумя кривыми. Логично, что модель должна быть как можно ближе к идеальной кривой и как можно дальше от случайной.

Кривая ROC случайной модели служит хорошей визуальной базовой линией — когда мы добавляем ее на график, это помогает понять, насколько далека наша модель от данной базовой линии. Именно поэтому рекомендуется всегда включать эту прямую в график.

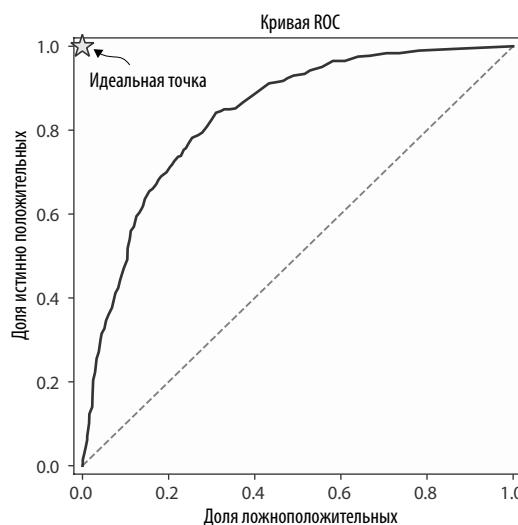
Однако в действительности нам не требуется генерировать случайную модель каждый раз, когда мы хотим получить кривую ROC: мы знаем, как она выглядит, поэтому можем просто включить в график прямую линию от (0, 0) до (1, 1).

Что касается идеальной модели, то мы знаем, что она всегда поднимается до  $(0, 1)$ , а затем переходит прямо к  $(1, 1)$ . Верхний левый угол называется «идеальной точкой»: это точка, в которой идеальная модель получает 100 % TPR и 0 % FPR. Мы хотим, чтобы наши модели были как можно ближе к идеальной точке.

Используя эту информацию, можно сократить код для построения кривой:

```
plt.figure(figsize=(5, 5))
plt.plot(df_scores.fpr, df_scores.tpr)
plt.plot([0, 1], [0, 1])
```

Это приводит к результату, показанному на рис. 4.30.



**Рис. 4.30.** Кривая ROC. Базовая линия позволяет легче увидеть, насколько далека кривая ROC нашей модели от кривой случайной модели. Верхний левый угол  $(0, 1)$  — это «идеальная точка»: чем ближе к ней наши модели, тем лучше

Хотя вычислять все значения FPR и TPR для множества пороговых значений — хорошее упражнение, нам не нужно проделывать это самостоятельно каждый раз, когда требуется построить кривую ROC. Мы можем просто использовать функцию `roc_curve` из пакета `metrics` Scikit-learn:

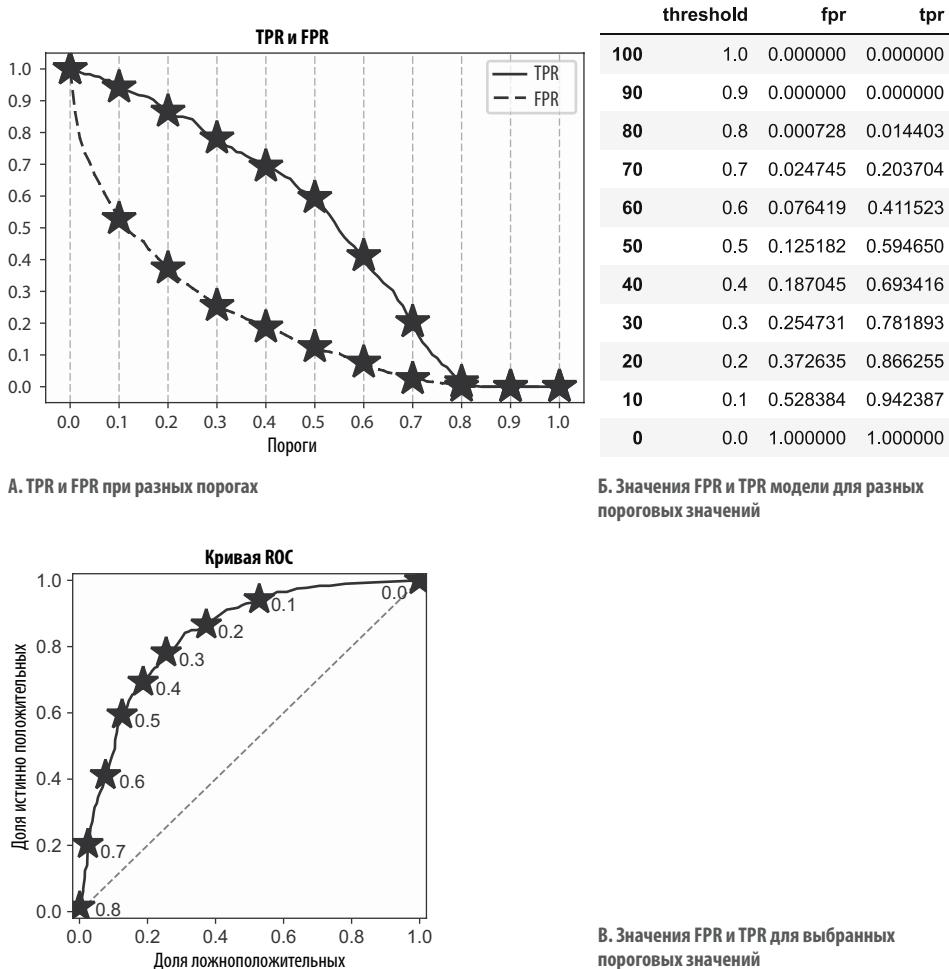
```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_val, y_pred)

plt.figure(figsize=(5, 5))
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1])
```

В результате мы получаем график, идентичный предыдущему (см. рис. 4.30).

Теперь попробуем лучше разобраться в кривой и понять, что она на самом деле может нам сказать. Для этого мы визуально сопоставим значения TPR и FPR с их пороговыми значениями на кривой ROC (рис. 4.31).



**Рис. 4.31.** Перевод графиков TPR и FPR для различных пороговых значений (A и Б) в кривую ROC (B). На графике ROC мы начинаем снизу слева с высокими пороговыми значениями, где для большинства клиентов прогнозируется «без оттока», и постепенно переходим к верхнему правому углу с низкими пороговыми значениями, где для большинства клиентов прогнозируется «отток»

На графике ROC мы начинаем с точки  $(0, 0)$  — это точка в левом нижнем углу. Это соответствует 0 % FPR и 0 % TPR, что происходит при высоких порогах, таких как 1,0, когда ни один клиент не превышает этот показатель.

В этих случаях мы просто в конечном итоге прогнозируем «без оттока» для всех. Вот почему наш TPR равен 0 %: мы никогда правильно не прогнозируем «отток» для клиентов. С другой стороны, FPR равен 0 %, поскольку эта фиктивная модель может дать правильный прогноз «без оттока» для всех клиентов «без оттока», поэтому ложных срабатываний нет. По мере продвижения вверх по кривой мы рассматриваем значения FPR и TPR, оцененные при меньших пороговых значениях. При 0,7 FPR изменяется незначительно, с 0 до 2 %, но TPR увеличивается от 0 до 20 % (см. рис. 4.31, *B* и *B*).

Следуя этой линии, мы продолжаем снижать пороговое значение и оценивать модель при меньших значениях, прогнозируя «отток» для все большего числа клиентов. В какой-то момент мы покрываем большинство положительных результатов (клиентов с «оттоком»). Например, при пороге 0,2 мы прогнозируем «отток» для большинства пользователей; то есть многие из этих прогнозов окажутся ложноположительными. Затем FPR начинает расти быстрее, чем TPR; при пороге 0,2 он уже составляет почти 40 %.

В конце концов, мы достигаем порога 0,0 и прогнозируем «отток» для всех, таким образом достигая верхнего правого угла графика ROC.

Когда мы начинаем с высоких пороговых значений, все модели равнозначны: любая модель с высокими пороговыми значениями деградирует до константной «модели», которая все время предсказывает `False`. По мере снижения порога мы начинаем прогнозировать «отток» для некоторых клиентов. Чем лучше модель, тем для большего количества клиентов правильно прогнозируется «отток», что приводит к лучшему TPR. Аналогично хорошие модели имеют меньший FPR, поскольку у них меньше ложноположительных результатов.

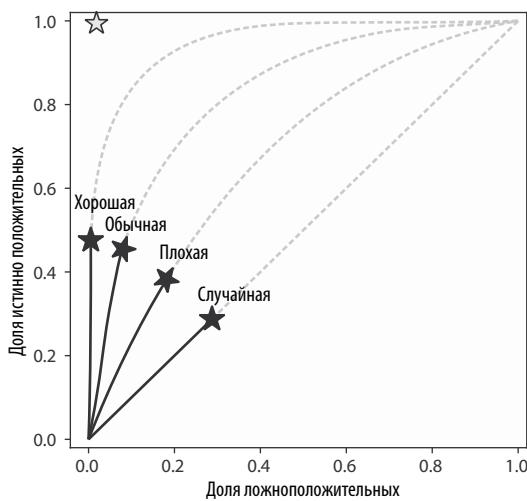
Таким образом, кривая ROC хорошей модели сначала поднимается настолько высоко, насколько возможно, и только затем начинает уклоняться вправо. Плохие модели, напротив, с самого начала имеют более высокие FPR и более низкие TPR, поэтому их кривые имеют тенденцию отклоняться вправо раньше (рис. 4.32).

Это позволяет нам сравнить несколько моделей: достаточно просто нанести их на один и тот же график и посмотреть, какая окажется ближе к идеальной точке (0, 1). Например, взглянем на кривые ROC большой и малой моделей и нанесем их на один и тот же график:

```
fpr_large, tpr_large, _ = roc_curve(y_val, y_pred)
fpr_small, tpr_small, _ = roc_curve(y_val, y_pred_small)

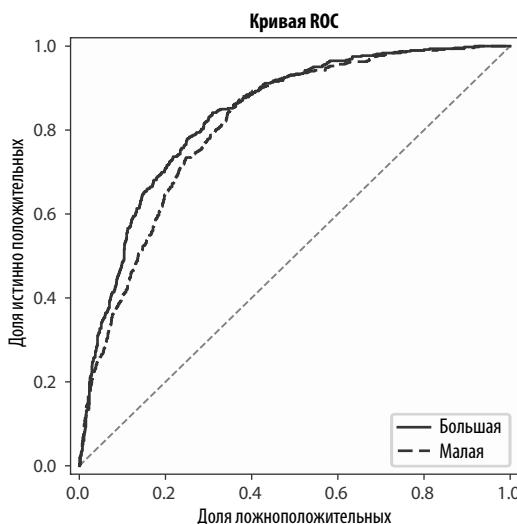
plt.figure(figsize=(5, 5))

plt.plot(fpr_large, tpr_large, color='black', label='Large')
plt.plot(fpr_small, tpr_small, color='black', label='Small')
plt.plot([0, 1], [0, 1])
plt.legend()
```



**Рис. 4.32.** Кривые ROC хороших моделей поднимаются насколько возможно перед уклоном направо. С другой стороны, плохие модели, как правило, имеют больше ложноположительных результатов изначально, поэтому они обычно уходят вправо раньше

Это позволяет получить две кривые ROC на одном и том же графике (рис. 4.33). Сразу заметно, что большая модель лучше, чем малая: она ближе к идеальной точке для всех пороговых значений.



**Рис. 4.33.** Построение нескольких кривых ROC на одном графике помогает визуально определить, какая модель работает лучше

Кривые ROC весьма полезны сами по себе, но у нас есть и другая метрика, основанная на них: площадь под кривой ROC (area under the ROC curve, AUC).

### 4.3.6. Площадь под кривой ROC (AUC)

Оценивая наши модели с помощью кривой ROC, мы хотим, чтобы они оказывались как можно ближе к идеальной точке и как можно дальше от случайной базовой линии.

Мы можем количественно оценить эту «близость», измерив площадь под кривой ROC. Можно использовать эту метрику — сокращенно AU ROC, или часто просто AUC, — в качестве метрики для оценки производительности модели бинарной классификации.

Идеальная модель образует квадрат  $1 \times 1$ , поэтому площадь под ее кривой ROC равна 1, или 100 %. Случайная модель занимает только половину от этого, поэтому ее AUC составляет 0,5, или 50 %. AUC двух наших моделей — большой и малой — будут находиться где-то между случайной базовой линией в 50 % и идеальной кривой в 100 %.

#### **ВАЖНО**

AUC в 0,9 указывает на достаточно хорошую производительность модели; 0,8 — на нормальную, 0,7 — на не очень эффективную, а 0,6 — на довольно низкую.

Чтобы рассчитать AUC для наших моделей, достаточно использовать функцию `auc` из пакета `metrics` в Scikit-learn:

```
from sklearn.metrics import auc
auc(df_scores.fpr, df_scores.tpr)
```

Для большой модели результат равен 0,84; для малой — 0,81 (рис. 4.34). Прогнозирование оттока — сложная проблема, поэтому AUC в 80 % видится довольно хорошим результатом.

```
from sklearn.metrics import auc
auc(df_scores.fpr, df_scores.tpr)
```

0.8359001084215382

```
auc(df_scores_small.fpr, df_scores_small.tpr)
```

0.8125475467380692

**Рис. 4.34.** AUC для наших моделей: 84 % для большой модели и 81 % для малой

Если все, что нам нужно, — это AUC, нам не требуется предварительно вычислять кривую ROC. Мы можем пойти кратчайшим путем и применить функцию `roc_auc_score` из Scikit-learn, которая сделает все сама и просто вернет AUC нашей модели:

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_val, y_pred)
```

Мы получаем примерно те же результаты, что и ранее (рис. 4.35).

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_val, y_pred)
```

0.8363366398907399

```
roc_auc_score(y_val, y_pred_small)
```

0.8129354083179088

**Рис. 4.35.** Расчет AUC с помощью функции `roc_auc_score` из Scikit-learn

### ВНИМАНИЕ

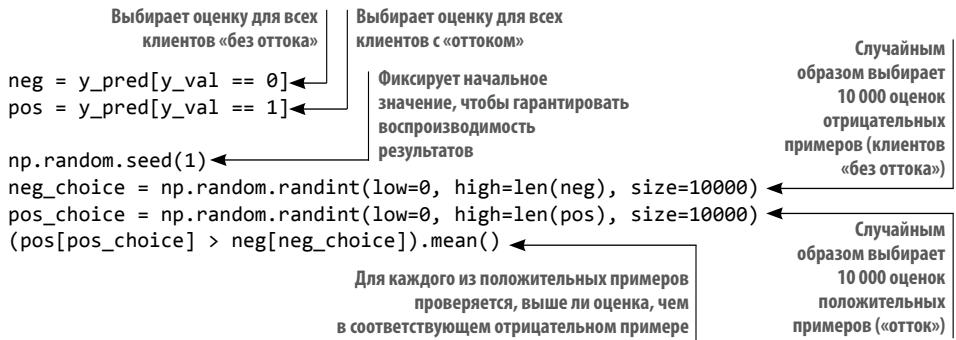
Значения после вызова `roc_auc_score` могут немного отличаться от AUC, вычисленных на основе датафреймов, где мы сами рассчитали TPR и FPR: Scikit-learn использует более точный метод для создания кривых ROC.

Кривые ROC и оценки AUC показывают, насколько хорошо модель разделяет положительные и отрицательные примеры.

Более того, AUC имеет хорошую вероятностную интерпретацию: она сообщает нам, какова вероятность того, что случайно выбранный положительный пример получит более высокую оценку, чем случайно выбранный отрицательный.

Предположим, мы случайным образом выбираем клиента, который, как мы знаем, ушел в результате «оттока», и клиента, который этого не сделал, а затем применяем к ним модель и оцениваем результат для каждого. Мы хотим, чтобы модель оценивала клиента с «оттоком» выше, чем «без оттока». AUC сообщает нам вероятность этого, то есть вероятность того, что оценка случайно выбранного клиента с «оттоком» выше, чем оценка случайно выбранного клиента «без оттока».

Мы можем это подтвердить. Если мы проведем этот эксперимент 10 000 раз, а затем подсчитаем, во сколько раз оценка положительного примера превышала оценку отрицательного, то процент случаев, когда это так, должен примерно соответствовать AUC:



Результат составляет 0,8356, что в действительности довольно близко к значению AUC нашего классификатора.

Такая интерпретация AUC дает нам дополнительное представление о качестве наших моделей. Идеальная модель упорядочивает всех клиентов таким образом, чтобы сначала располагались клиенты «без оттока», а затем клиенты с «оттоком». При таком порядке AUC всегда равен 1,0: оценка случайно выбранного клиента с «оттоком» всегда выше, чем оценка клиента «без оттока». С другой стороны, случайная модель просто перетасовывает клиентов, поэтому оценка клиента с «оттоком» имеет только 50%-ную вероятность оказаться выше, чем оценка клиента «без оттока».

Таким образом, AUC не только дает способ оценки моделей для всех возможных пороговых значений, но и описывает, насколько хорошо модель разделяет два класса: в нашем случае это «отток» и «без оттока».

Если разделение хорошее, то мы можем упорядочить клиентов так, чтобы большинство пользователей с «оттоком» оказались на первом месте. Такая модель будет иметь хороший показатель AUC.

### ПРИМЕЧАНИЕ

Вы должны иметь в виду эту интерпретацию: она позволяет простым языком объяснить значение AUC людям, не имеющим опыта машинного обучения (например, менеджерам и другим лицам, принимающим решения).

Это делает AUC классификационной метрикой по умолчанию в большинстве ситуаций, и часто именно ее мы используем при поиске наилучшего набора параметров для наших моделей.

Процесс поиска наилучших параметров называется настройкой параметров, и в следующем разделе мы увидим, как ее провести.

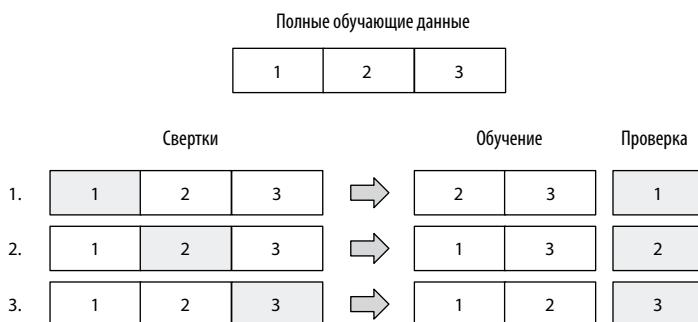
## 4.4. НАСТРОЙКА ПАРАМЕТРОВ

В предыдущей главе для тестирования наших моделей мы использовали простую схему проверки с удержанием. В этой схеме мы извлекаем часть данных и удерживаем их исключительно для целей проверки. Это хорошая практика, но она не всегда обеспечивает полную картину. Она говорит о том, насколько хорошо модель будет работать с этими конкретными точками данных. Однако это не обязательно означает, что модель будет так же хорошо работать с другими точками данных. Как же нам проверить, действительно ли модель работает последовательно и предсказуемо?

### 4.4.1. К-кратная перекрестная проверка

Можно использовать все доступные данные для оценки качества моделей и получения более надежных результатов проверки. Мы можем просто выполнить проверку несколько раз.

Сначала мы разделяем весь набор данных на определенное количество частей (скажем, на три). Затем обучаем модель на двух частях и проверяем на оставшейся. Мы повторяем процесс трижды и в итоге получаем три разные оценки. Именно эта идея лежит в основе К-кратной перекрестной проверки (рис. 4.36).



**Рис. 4.36.** К-кратная перекрестная проверка ( $K = 3$ ). Мы разделяем весь набор данных на три равные части, или свертки. Затем каждую свертку мы берем в качестве проверочного набора данных и используем оставшиеся  $K - 1$  сверток в качестве обучающих. После обучения модели мы оцениваем ее на проверочной свертке и в конце получаем  $K$  значений метрик

Прежде чем мы реализуем алгоритм, нам следует упростить процесс обучения, чтобы иметь возможность многократно запускать этот процесс. Для этого мы поместим весь код для обучения в функцию `train`, которая сначала преобразует данные с помощью прямого кодирования, а затем обучает модель (листинг 4.3).

**Листинг 4.3.** Обучение модели

```
def train(df, y):
    cat = df[categorical + numerical].to_dict(orient='records') | Применяет прямое
    dv = DictVectorizer(sparse=False) | кодирование
    dv.fit(cat)

    X = dv.transform(cat)

    model = LogisticRegression(solver='liblinear') | Обучает модель
    model.fit(X, y)

    return dv, model
```

Аналогичным образом мы помещаем логику прогнозирования в функцию `predict`. Эта функция принимает датафрейм с клиентами, векторизатор, который мы «обучили» ранее (для выполнения прямого кодирования) и модель. Затем мы применяем векторизатор к датафрейму, получаем матрицу и, наконец, применяем модель к матрице, чтобы получить прогнозы (листинг 4.4).

**Листинг 4.4.** Применение модели к новым данным

```
def predict(df, dv, model):
    cat = df[categorical + numerical].to_dict(orient='records') | Применяет то же
    X = dv.transform(cat) | самое прямое
    y_pred = model.predict_proba(X)[:, 1] | кодирование,
    | что и при обучении
    | Использует модель
    | для составления
    | прогнозов

    return y_pred
```

Теперь мы можем использовать эти функции для реализации К-кратной перекрестной проверки (листинг 4.5).

Нам не придется самостоятельно реализовывать перекрестную проверку в Scikit-learn для этого есть специальный класс. Он называется `KFold` и находится в пакете `model_selection`.

**Листинг 4.5.** К-кратная перекрестная проверка

```
from sklearn.model_selection import KFold | ❶ Импортирует класс KFold
kfold = KFold(n_splits=10, shuffle=True, random_state=1) | ❷ Использует его
| для разделения данных
| на десять частей
aucs = [] | ❸ Создает список
| для хранения результатов
for train_idx, val_idx in kfold.split(df_train_full): | ❹ Проходит
| по десяти различным
| разбиениям данных
    df_train = df_train_full.iloc[train_idx]
    df_val = df_train_full.iloc[val_idx]
    y_train = df_train.churn.values
    y_val = df_val.churn.values | ❺ Разбивает данные
| на обучающий
| и проверочный наборы
```

```

dv, model = train(df_train, y_train) | ⑥ Обучает модель
y_pred = predict(df_val, dv, model) | и делает прогнозы

auc = roc_auc_score(y_val, y_pred) ←
aucs.append(auc) ← ⑦ Оценивает качество модели
                     Сохраняет AUC в списке
                     ⑧ с результатами на проверочных данных

```

Обратите внимание, что при задании разделения в классе `KFold` в ② мы устанавливаем три параметра:

- `n_splits = 10` — это K, которое определяет количество разбиений;
- `shuffle = True` — мы просим перетасовать данные перед разбиением;
- `random_state = 1` — поскольку в процессе происходит перетасовка данных, мы хотим, чтобы результаты были воспроизводимыми, поэтому фиксируем начальное значение для генератора случайных чисел.

Здесь мы использовали К-кратную перекрестную проверку с K = 10. Таким образом, после запуска мы получаем десять разных чисел — десять баллов AUC, оцененных по десяти различным проверочным сверткам:

```
0.849, 0.841, 0.859, 0.833, 0.824, 0.841, 0.844, 0.822, 0.845, 0.861
```

Это больше не единое число, и мы можем рассматривать это как распределение баллов AUC для нашей модели. Из распределения мы можем извлечь кое-какую статистику, например среднее значение и стандартное отклонение:

```
print('auc = %.3f ± %.3f' % (np.mean(aucs), np.std(aucs)))
```

При этом выводится  $0,842 \pm 0,012$ .

Теперь мы не только знаем среднюю производительность, но и имеем представление о том, насколько она изменчива (или насколько далеко может отклоняться от средней).

Хорошая модель должна быть достаточно стабильной на различных свертках: таким образом, мы гарантируем, что не получим множество сюрпризов, когда модель начнет работать. Об этом нам говорит стандартное отклонение: чем оно меньше, тем стабильнее модель.

Теперь мы можем использовать К-кратную перекрестную проверку для настройки параметров (выбора наилучших параметров).

#### 4.4.2. Поиск наилучших параметров

Мы узнали, как можно использовать К-кратную перекрестную проверку для оценки производительности модели. Модель, которую мы обучали ранее, использовала значение по умолчанию для параметра C, который управляет степенью регуляризации.

Выберем процедуру перекрестной проверки для выбора наилучшего параметра С. Для этого мы сначала настроим функцию `train`, чтобы она принимала один дополнительный параметр (листинг 4.6).

**Листинг 4.6.** Функция для обучения модели с параметром С для управления регуляризацией

```
def train(df, y, C): ← Добавляет дополнительный
    cat = df[categorical + numerical].to_dict(orient='records') ← параметр к функции train

    dv = DictVectorizer(sparse=False)
    dv.fit(cat)

    X = dv.transform(cat) ← Использует указанный
    model = LogisticRegression(solver='liblinear', C=C) ← параметр во время
    model.fit(X, y) ← обучения

    return dv, model
```

Теперь найдем наилучший параметр С. Идея проста:

- цикл по различным значениям С;
- для каждого С выполните перекрестную проверку и запишите среднее значение AUC по всем сверткам, а также стандартное отклонение (листинг 4.7).

**Листинг 4.7.** Настройка модели: выбор наилучшего параметра С с помощью перекрестной проверки

```
nfolds = 5
kfold = KFold(n_splits=nfolds, shuffle=True, random_state=1)

for C in [0.001, 0.01, 0.1, 0.5, 1, 10]:
    aucss = []

    for train_idx, val_idx in kfold.split(df_train_full):
        df_train = df_train_full.iloc[train_idx]
        df_val = df_train_full.iloc[val_idx]

        y_train = df_train.churn.values
        y_val = df_val.churn.values

        dv, model = train(df_train, y_train, C=C)
        y_pred = predict(df_val, dv, model)

        auc = roc_auc_score(y_val, y_pred)
        aucss.append(auc)

    print('C=%s, auc = %0.3f ± %0.3f' % (C, np.mean(aucss), np.std(aucss)))
```

После запуска мы получаем

```
C=0.001, auc = 0.825 ± 0.013
C=0.01, auc = 0.839 ± 0.009
C=0.1, auc = 0.841 ± 0.008
C=0.5, auc = 0.841 ± 0.007
C=1, auc = 0.841 ± 0.007
C=10, auc = 0.841 ± 0.007
```

Мы наблюдаем, что после  $C = 0,1$  средняя AUC остается прежней и больше не растет.

Однако стандартное отклонение меньше для  $C = 0,5$ , чем для  $C = 0,1$ , поэтому мы должны использовать его. Причина, по которой мы выбираем  $C = 0,5$ , а не  $C = 1$  или  $C = 10$ , проста: когда параметр  $C$  мал, модель более упорядоченна. Веса этой модели более ограниченны, поэтому в целом они меньше. Небольшие веса в модели позволяют нам быть уверенными в том, что модель будет вести себя должным образом, когда мы применим ее к реальным данным. Итак, мы выбираем  $C = 0,5$ .

Теперь нужно сделать последний шаг: обучить модель на всех обучающих и проверочных наборах данных, после чего применить ее к тестовому набору, чтобы убедиться, что она действительно хорошо работает.

Воспользуемся для этого нашими функциями `train` и `predict`:

```
y_train = df_train_full.churn.values
y_test = df_test.churn.values
dv, model = train(df_train_full, y_train, C=0.5) ← Обучает модель по полному
y_pred = predict(df_test, dv, model) ← набору обучающих данных
                                         Применяет ее к тестовому
auc = roc_auc_score(y_test, y_pred) |   набору данных
print('auc = %.3f' % auc)             |   Оценивает прогнозы
                                         по тестовым данным
```

После выполнения мы видим, что производительность модели (AUC) надержанном тестовом наборе составляет 0,858.

Показатель немного выше по сравнению с проверочным набором, но это не проблема: так могло произойти случайно. Важно то, что оценка существенно не отличается от оценки при проверке.

Теперь мы можем использовать эту модель для оценки реальных клиентов и подумать о нашей маркетинговой кампании по предотвращению оттока. В следующей главе мы увидим, как развернуть данную модель в производственной среде.

## 4.5. СЛЕДУЮЩИЕ ШАГИ

### 4.5.1. Упражнения

Попробуйте выполнить следующие упражнения, чтобы углубиться в темы оценки и выбора модели.

- В текущей главе мы построили графики TPR и FPR для разных пороговых значений, и это помогло нам разобраться, что означают данные показатели, а также как меняется производительность нашей модели при выборе других пороговых значений. Будет весьма полезным выполнить аналогичное упражнение применительно к точности и отклику, поэтому попробуйте повторить данный эксперимент, на сей раз используя точность и отклик вместо TPR и FPR.
- При построении графика точности и отклика для разных пороговых значений мы наблюдаем конфликт: когда точность повышается, отклик понижается, и наоборот. Это называется «компромисс между точностью и откликом»: мы не сможем выбрать порог, который обеспечивает как хорошую точность, так и хороший отклик. Однако у нас имеются стратегии для выбора порога, даже несмотря на то, что точность и отклик противоречат друг другу. Одна из них — построение кривых точности и отклика, наблюдение за тем, где они пересекаются, и использование этого порога для бинаризации прогнозов. Попробуйте реализовать эту идею.
- Еще одна идея позволяет обойти компромисс между точностью и откликом — это оценка F1, которая объединяет точность и отклик в одно значение. Затем, выбирая наилучший порог, мы можем просто остановиться на том, который максимизирует оценку F1. Формула для вычисления оценки F1 такова:  $F1 = 2 \times P \times R / (P + R)$ , где P — точность, а R — отклик. Реализуйте эту идею и выберите наилучший порог на основе метрики F1.
- Мы узнали, что точность и отклик служат лучшими показателями для оценки моделей классификации, чем точность, поскольку не зависят от ложноположительных результатов, количество которых может быть высоким в несбалансированных наборах данных. Тем не менее позже мы узнали, что AUC в действительности использует ложноположительные результаты в FPR. Для очень сильно несбалансированных случаев (скажем, 1000 отрицательных результатов на 1 положительный) AUC также может стать проблемой. В таких случаях лучше работает другой показатель: площадь под кривой точности-отклика, или AU PR. Кривая точности-отклика аналогична ROC, но вместо построения FPR в сравнении с TPR мы строим отклик по оси x, а точность — по оси y. Как и для кривой ROC, мы можем вычислить площадь под кривой PR и использовать ее в качестве показателя для оценки различных

моделей. Попробуйте построить кривые PR для наших моделей, вычислить оценки PR AU и сравнить их с показателями случайной и идеальной модели.

- Мы рассмотрели К-кратную перекрестную проверку и использовали ее, чтобы понять, как может выглядеть распределение оценок AUC в тестовом наборе данных. Когда  $K = 10$ , мы получаем десять наблюдений, которых при некоторых обстоятельствах может оказаться недостаточно. Однако идея может быть расширена до повторяющихся шагов К-кратной перекрестной проверки. Процесс прост: мы повторяем процесс К-кратной перекрестной проверки несколько раз, постоянно по-разному перетасовывая набор данных, выбирая случайное начальное значение на каждой итерации. Реализуйте повторную перекрестную проверку и выполните десятикратную перекрестную проверку десять раз, чтобы увидеть, как выглядит распределение оценок.

### 4.5.2. Другие проекты

Вы также можете продолжить работу с другими проектами самообучения из предыдущей главы: проектом оценки лидеров и проектом прогнозирования по умолчанию. Попробуйте выполнить действия, указанные ниже.

- Рассчитайте все показатели, которые мы рассмотрели в этой главе: матрицу ошибок, точность и отклик, а также AUC. Кроме того, попробуйте подсчитать оценки из упражнений: оценку F1, а также AU PR (область под кривой точности-отклика).
- Используйте К-кратную перекрестную проверку, чтобы выбрать наилучший параметр  $C$  для модели.

## РЕЗЮМЕ

- Метрика — это число, которое позволяет оценивать производительность модели машинного обучения. Как только мы выберем метрику, мы можем с ее помощью сравнивать несколько моделей машинного обучения между собой и выбирать наилучшую из них.
- Достоверность — простейший показатель двоичной классификации: он сообщает нам процент правильно классифицированных наблюдений в проверочном наборе. Его легко понять и вычислить, но он может ввести в заблуждение, если набор данных несбалансирован.
- Когда модель бинарной классификации делает прогноз, у нас есть только четыре возможных результата: истинно положительный и истинно отрицательный (правильные ответы), ложноположительный и ложноотрицательный (неправильные ответы). Чтобы лучше понимать эти результаты, их можно визуально представить с помощью матрицы ошибок. Она служит основой для многих других показателей бинарной классификации.

- Точность — это доля правильных ответов среди наблюдений, для которых наш прогноз оказался верен. Если мы используем модель оттока для отправки рекламных сообщений, то точность сообщает нам процент клиентов, которые действительно собирались отказаться от контракта, среди всех, кто получил сообщение. Чем выше точность, тем меньше пользователей «без оттока», для которых мы ошибочно получили прогноз «отток».
- Отклик — это доля правильных ответов среди всех положительных наблюдений. Он говорит нам о процентах клиентов с «оттоком», которых мы спрогнозировали правильно. Чем выше отклик, тем меньше клиентов с «оттоком», которых мы не можем идентифицировать.
- Кривая ROC анализирует модели бинарной классификации сразу по всем пороговым значениям. Площадь под кривой ROC (AUC) показывает, насколько хорошо модель отделяет положительные наблюдения от отрицательных. Из-за своей легкости интерпретации и широкой применимости AUC стала метрикой по умолчанию для оценки моделей бинарной классификации.
- К-кратная перекрестная проверка предоставляет способ использовать все обучающие данные для проверки модели: мы разбиваем данные на K сверток и используем каждую по очереди как проверочный набор, а остальные K – 1 сверток — для обучения. В результате вместо одного числа мы получаем K значений, по одному для каждой свертки. С помощью этих цифр мы можем выяснить производительность модели в среднем, а также оценить, насколько она изменчива на разных свертках.
- К-кратная перекрестная проверка — лучший способ настройки параметров и выбора наилучшей модели: она дает надежную оценку показателя по множеству сверток.

В следующей главе мы рассмотрим развертывание нашей модели в производственной среде.

## ОТВЕТЫ К УПРАЖНЕНИЯМ

- Упражнение 4.1. Ответ Б. Клиент, для которого мы прогнозировали «отток», но фактически он не произошел.
- Упражнение 4.2. Ответ Б. Процент клиентов, которые действительно ушли («отток»), среди тех, для кого мы спрогнозировали «отток».
- Упражнение 4.3. Ответ А. Процент правильно идентифицированных клиентов с прогнозом «отток» среди всех ушедших («отток»).
- Упражнение 4.4. Ответ А. Идеальная модель ранжирования всегда оценивает клиентов с «оттоком» выше, чем клиентов «без оттока».

# 5

## Развертывание моделей машинного обучения

### В этой главе

- ✓ Сохранение моделей с помощью Pickle.
- ✓ Обеспечение доступности моделей с помощью Flask.
- ✓ Управление зависимостями с помощью Pipenv.
- ✓ Создание автономной службы с помощью Docker.
- ✓ Развертывание ее в облаке с помощью AWS Elastic Beanstalk.

Поскольку мы продолжаем работать с методами машинного обучения, мы продолжим использовать начатый ранее проект: прогнозирование оттока. В главе 3 мы использовали Scikit-learn для построения модели выявления оттока клиентов. После этого, в главе 4, оценили качество этой модели и выбрали наилучший параметр C с помощью перекрестной проверки.

У нас уже есть модель, которая работает в нашем блокноте Jupyter. Теперь нам нужно внедрить эту модель в производство, чтобы другие сервисы могли использовать ее для принятия решений на основе результатов ее работы.

В этой главе мы рассмотрим *развертывание модели*: процесс ввода моделей в эксплуатацию. В частности, мы узнаем, как упаковать модель в веб-сервис, чтобы другие сервисы могли ее использовать. Мы также рассмотрим, как развернуть веб-сервис в рабочей среде.

## 5.1. МОДЕЛЬ ПРОГНОЗИРОВАНИЯ ОТТОКА

Чтобы обучиться развертыванию, мы используем модель, которую обучали ранее. Сначала в этом разделе мы рассмотрим, как можно использовать модель для получения прогнозов, а затем узнаем, как сохранить ее с помощью Pickle.

### 5.1.1. Использование модели

Чтобы упростить задачу, мы можем продолжить работу с тем же блокнотом Jupyter, который использовался для глав 3 и 4.

Воспользуемся этой моделью для расчета вероятности оттока для следующего клиента:

```
customer = {
    'customerid': '8879-zkjof',
    'gender': 'female',
    'seniorcitizen': 0,
    'partner': 'no',
    'dependents': 'no',
    'tenure': 41,
    'phoneservice': 'yes',
    'multiplelines': 'no',
    'internetservice': 'dsl',
    'onlinesecurity': 'yes',
    'onlinebackup': 'no',
    'deviceprotection': 'yes',
    'techsupport': 'yes',
    'streamingtv': 'yes',
    'streamingmovies': 'yes',
    'contract': 'one_year',
    'paperlessbilling': 'yes',
    'paymentmethod': 'bank_transfer_(automatic)',
    'monthlycharges': 79.85,
    'totalcharges': 3320.75,
}
```

Чтобы спрогнозировать отток для этого клиента, мы можем использовать функцию `predict`, которую написали в предыдущей главе:

```
df = pd.DataFrame([customer])
y_pred = predict(df, dv, model)
y_pred[0]
```

Функции требуется датафрейм, поэтому сначала мы создадим такой фрейм с одной строкой — нашим клиентом. Затем мы поместим его в функцию `predict`. Результатом будет массив NumPy с одним элементом — прогнозируемой вероятностью оттока для этого клиента:

0,059605

Это означает, что для этого клиента спрогнозирована 6%-ная вероятность оттока.

Теперь взглянем на функцию `predict`, которую мы написали ранее в целях применения модели к клиентам в проверочном наборе:

```
def predict(df, dv, model):
    cat = df[categorical + numerical].to_dict(orient='rows')
    X = dv.transform(cat)
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred
```

Использование ее для одного клиента выглядит неэффективным и ненужным: мы создаем датафрейм из одного клиента только для того, чтобы позже преобразовать его обратно в словарь внутри `predict`.

Чтобы избежать этого ненужного преобразования, можно создать отдельную функцию для прогнозирования вероятности оттока для единственного клиента. Назовем ее `predict_single`:

```
def predict_single(customer, dv, model): ← Вместо передачи
    X = dv.transform([customer]) ← датафрейма передает
    y_pred = model.predict_proba(X)[:, 1] ← одного клиента | Векторизирует клиента:
    return y_pred[0] ← Применяет модель | создает матрицу X
                                         |
                                         | Поскольку у нас
                                         | только один клиент,
                                         | нам нужен только
                                         | первый элемент
                                         | результата
```

Использовать функцию становится проще — мы просто вызываем ее с нашим клиентом (словарем):

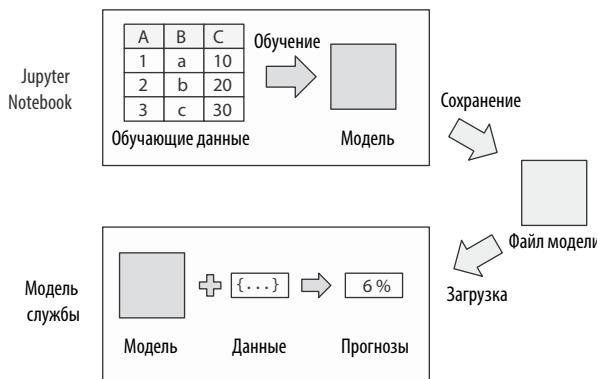
```
predict_single(customer, dv, model)
```

Результат тот же: вероятность оттока для этого клиента составляет 6 %.

Мы обучили нашу модель в блокноте Jupyter, который начали использовать в главе 3. Модель размещена именно там, и как только мы остановим Jupyter Notebook, обученная модель исчезнет. Это означает, что мы можем использовать ее лишь внутри блокнота и нигде больше. Далее мы узнаем, как преодолеть это ограничение.

### 5.1.2. Использование Pickle для сохранения и загрузки модели

Чтобы иметь возможность использовать модель за пределами нашего блокнота, нам требуется сохранить ее, тогда позже другой процесс сможет загрузить ее и использовать (рис. 5.1).



**Рис. 5.1.** Мы обучаем модель в блокноте Jupyter. Для дальнейшего использования нам сначала нужно ее сохранить, а затем загрузить в другом процессе

Pickle — это модуль сериализации/десериализации, который уже встроен в Python: используя его, мы можем сохранить произвольный объект Python (за некоторыми исключениями) в файл. Как только у нас появится файл, мы сможем загружать модель из него в другом процессе.

### ПРИМЕЧАНИЕ

Pickle также может использоваться как глагол: сериализация (pickling) объекта в Python означает сохранение его с помощью модуля Pickle.

### Сохранение модели

Чтобы сохранить модель, мы сначала импортируем модуль Pickle, а затем используем его функцию `dump`:

```
import pickle
with open('churn-model.bin', 'wb') as f_out: ← Указывает файл, в который
    pickle.dump(model, f_out) ← Сохраняет модель в файл
                                         с помощью Pickle
```

Чтобы сохранить модель, мы используем функцию `open`, которая принимает два аргумента:

- имя файла, который мы хотим открыть. В нашем случае это `churn-model.bin`;
- режим, в котором мы открываем файл. В нашем случае режим `wb` означает, что мы хотим записать в файл (`w`), и этот файл будет двоичным (`b`), а не текстовым — для записи в файлы Pickle использует двоичный формат.

Функция `open` возвращает `f_out` — дескриптор файла, который мы можем использовать для записи.

Далее мы используем функцию `dump` из Pickle. Она также принимает два аргумента:

- объект, который мы хотим сохранить. В нашем случае это `model`;
- дескриптор файла, указывающий на выходной файл, в нашем случае `f_out`.

Наконец, в этом коде мы используем конструкцию `with`. Когда мы открываем файл с помощью `open`, нам нужно закрыть его после окончания записи. В случае использования `with` это происходит автоматически. Без `with` наш код выглядел бы следующим образом:

```
f_out = open('churn-model.bin', 'wb')
pickle.dump(model, f_out)
f_out.close()
```

В нашем случае сохранения одной только модели недостаточно: помимо нее, у нас есть `DictVectorizer`, который мы «обучили» вместе с моделью. Нам требуется сохранить и то и другое.

Самый простой способ сделать это — поместить оба объекта в кортеж при сериализации:

```
with open('churn-model.bin', 'wb') as f_out:
    pickle.dump((dv, model), f_out) Объект, который мы сохраняем, представляет собой кортеж из двух элементов
```

## Загрузка модели

Чтобы загрузить модель, мы используем функцию `load` из Pickle. Протестировать ее можно в том же блокноте Jupyter:

```
with open('churn-model.bin', 'rb') as f_in: Открывает файл в режиме чтения
    dv, model = pickle.load(f_in) Загружает кортеж и распаковывает его
```

Мы снова используем функцию `open`, однако на этот раз в другом режиме: `rb`, что означает открытие для чтения (`r`) двоичного файла (`b`).

### ПРЕДУПРЕЖДЕНИЕ

Будьте осторожны при указании режима. Случайное указание неправильного режима может привести к потере данных: если вы откроете существующий файл в режиме `w` вместо `r`, то содержимое будет перезаписано.

Поскольку мы сохраняли кортеж, при загрузке он будет распакован, так что мы получим одновременно и векторизатор, и модель.

### ПРЕДУПРЕЖДЕНИЕ

Распаковка объектов, найденных в Интернете, может быть опасной: на вашем компьютере может выполниться произвольный код. Распаковывайте только то, чему вы доверяете, или то, что сохранили сами.

Напишем простой сценарий на Python, который загружает модель и применяет ее к клиенту.

Мы назовем файл `churn_serving.py`. (В репозитории книги на GitHub этот файл называется `churn_serving_simple.py`.) Он содержит:

- функцию `predict_single`, которую мы написали ранее;
- код для загрузки модели;
- код для применения модели к клиенту.

Узнать больше о создании сценариев на Python можно в приложении Б.

Начнем с импорта. Для этого сценария нам нужно импортировать Pickle и NumPy:

```
import pickle
import numpy as np
```

Далее поместим туда функцию `predict_single`:

```
def predict_single(customer, dv, model):
    X = dv.transform([customer])
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred[0]
```

Теперь можно загрузить нашу модель:

```
with open('churn-model.bin', 'rb') as f_in:
    dv, model = pickle.load(f_in)
```

И применить ее:

```
customer = {
    'customerid': '8879-zkjof',
    'gender': 'female',
    'seniorcitizen': 0,
    'partner': 'no',
    'dependents': 'no',
    'tenure': 41,
    'phoneservice': 'yes',
    'multiplelines': 'no',
    'internetservice': 'dsl',
    'onlinesecurity': 'yes',
    'onlinebackup': 'no',
    'deviceprotection': 'yes',
    'techsupport': 'yes',
    'streamingtv': 'yes',
    'streamingmovies': 'yes',
    'contract': 'one_year',
    'paperlessbilling': 'yes',
    'paymentmethod': 'bank_transfer_(automatic)',
```

```
'monthlycharges': 79.85,
'totalcharges': 3320.75,
}

prediction = predict_single(customer, dv, model)
```

Наконец отобразим результаты:

```
print('prediction: %.3f' % prediction)

if prediction >= 0.5:
    print('verdict: Churn')
else:
    print('verdict: Not churn')
```

После сохранения файла мы можем запускать этот сценарий с помощью Python:

```
python churn_serving.py
```

После запуска мы должны немедленно увидеть результаты:

```
prediction: 0.059
verdict: Not churn
```

Таким образом, мы можем загрузить модель и применить ее к клиенту, которого указали в сценарии.

Конечно, мы не планируем вводить информацию о клиентах в сценарий вручную. В следующем разделе рассмотрим более практичный подход: размещение модели в веб-сервисе.

## 5.2. ДОСТУПНОСТЬ МОДЕЛИ

Мы уже знаем, как загружать обученную модель в другом процессе. Теперь нам нужно обеспечить ее *доступность* — чтобы другие могли использовать ее.

На практике это обычно означает, что модель развертывается как веб-сервис, поэтому другие сервисы могут взаимодействовать с ней, запрашивать прогнозы, после чего использовать результаты для принятия собственных решений.

В данном разделе мы посмотрим, как это сделать на Python с помощью Flask — фреймворка Python для создания веб-сервисов. Сначала рассмотрим, почему нам стоит использовать для этого веб-сервис.

### 5.2.1. Веб-сервисы

Мы уже знаем, как использовать модель для прогнозирования, но до сих просто кодировали признаки клиента в виде словаря Python. Попробуем представить, как наша модель будет использоваться на практике.

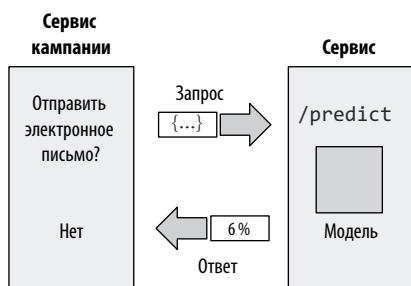
Предположим, у нас есть сервис для проведения маркетинговых кампаний. Для каждого клиента необходимо определить вероятность оттока, и если она достаточно высока, то отправить рекламное письмо со скидками. Естественно, этот сервис будет использовать нашу модель для принятия решения о том, следует ли ему отправлять такое письмо.

Один из возможных способов добиться этого — изменить код сервиса кампании: загрузить модель и оценить клиентов прямо в сервисе. Это неплохой подход, но сервис должен быть написан на Python, а мы должны иметь полный контроль над его кодом.

К сожалению, такая ситуация имеет место далеко не всегда: сервис может быть написан на каком-либо другом языке или за проект может отвечать другая команда (а это значит, что у нас не будет необходимого контроля).

Типичным решением задачи является размещение модели внутри веб-сервиса — небольшого сервиса (*микросервиса*), который работает только над оценкой клиентов.

Иными словами, нам требуется создать сервис оттока — сервис на Python, который будет обслуживать модель оттока. Он будет возвращать вероятность оттока для этого клиента с учетом его признаков. Для каждого клиента сервис кампаний будет запрашивать у сервиса оттока вероятность оттока и при достаточно высоком показателе отправлять рекламное электронное письмо (рис. 5.2).



**Рис. 5.2.** Сервис оттока заботится об обслуживании модели прогнозирования оттока, позволяя другим сервисам ее использовать

Это дает нам еще одно преимущество: разделение труда. Если модель создается специалистами по обработке данных, то они могут владеть сервисом и сопровождать его, в то время как другая команда займется рекламной кампанией.

Одним из самых популярных фреймворков для создания веб-сервисов на Python является Flask, который мы и рассмотрим далее.

## 5.2.2. Flask

Самый простой способ реализовать веб-сервис на Python – использовать Flask. Это довольно легковесная библиотека, начало работы с которой не требует большого количества кода; кроме того, она скрывает большую часть сложной работы с HTTP-запросами и ответами.

Прежде чем мы поместим нашу модель в веб-сервис, рассмотрим основы использования Flask. Для этого создадим простую функцию и сделаем ее доступной в виде веб-сервиса. Изучив основы, мы займемся моделью.

Предположим, у нас есть простая функция на Python с именем `ping`:

```
def ping():
    return 'PONG'
```

Она делает не так уж много: при вызове просто отвечает сообщением PONG. С помощью Flask превратим ее в веб-сервис.

Anaconda поставляется с предустановленной Flask, но если вы используете другой дистрибутив Python, то вам придется установить ее самостоятельно:

```
pip install flask
```

Мы поместим этот код в файл Python и назовем его `flask_test.py`.

Чтобы использовать Flask, сначала нужно его импортировать:

```
from flask import Flask
```

Теперь создадим приложение Flask – центральный объект для регистрации функций, которые необходимо предоставить в веб-сервисе. Мы назовем наше приложение `test`:

```
app = Flask('test')
```

Далее нужно указать, как именно можно получить доступ к функции, назначив ей адрес, или (в терминах Flask) *маршрут*. В нашем случае мы хотим использовать адрес `/ping`:

```
@app.route('/ping', methods=['GET']) ←
def ping():
    return 'PONG'
```

Регистрирует маршрут /ping  
и назначает его функции ping

В этом коде используются декораторы — расширенный функционал Python, который мы не рассматриваем в данной книге. Нам не нужно разбираться в тонкостях работы, а достаточно знать, что, помещая `@app.route` поверх определения функции, мы присваиваем функции `ping` адрес `/ping` веб-сервиса.

Для запуска потребуется лишь небольшой фрагмент кода:

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=9696)
```

Метод `run` объекта `app` запускает сервис. Мы указываем три параметра:

- `debug=True` — автоматически перезапускает наше приложение при внесении изменений в код;
- `host='0.0.0.0'` — делает веб-сервис общедоступным; в противном случае доступ к нему будет невозможен, если он размещен на удаленном компьютере (например, в AWS);
- `port=9696` — порт, который мы используем для доступа.

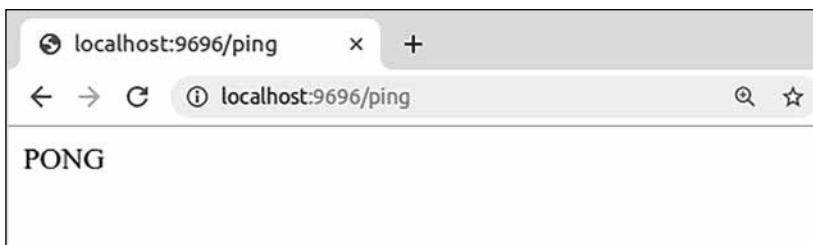
Итак, сервис готов к запуску. Запустим его:

```
python flask_test.py
```

После запуска мы должны увидеть следующее:

```
* Serving Flask app "test" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production
  deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:9696/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 162-129-136
```

Все это означает, что наше приложение Flask запущено и готово к приему запросов. Протестировать его мы можем с помощью нашего браузера: откройте его и введите `localhost:9696/ping` в адресной строке. Если вы запускаете сервис на удаленном сервере, то вам следует заменить `localhost` адресом сервера. (Для AWS EC2 используйте общедоступное имя хоста DNS. Убедитесь, что порт 9696 открыт в группе безопасности вашего экземпляра EC2: перейдите в группу безопасности и добавьте пользовательское правило TCP с портом 9696 и исходным кодом 0.0.0.0/0.) Браузер должен ответить сообщением PONG (рис. 5.3).



**Рис. 5.3.** Самый простой способ проверить, работает ли наше приложение, — это использовать браузер

Flask регистрирует все запросы, которые получает, поэтому мы должны увидеть строку, указывающую на то, что произошел запрос GET по маршруту /ping:

```
127.0.0.1 - - [02/Apr/2020 21:59:09] "GET /ping HTTP/1.1" 200 -
```

Как видите, Flask довольно проста: используя менее десяти строк кода, мы создали веб-сервис.

Далее мы узнаем, как настроить наш сценарий для прогнозирования оттока, а также превратить его в веб-сервис.

### **5.2.3. Обеспечение доступа к модели оттока с помощью Flask**

Мы немного изучили Flask, поэтому теперь мы можем вернуться к нашему сценарию и преобразовать его в приложение Flask.

Чтобы оценить клиента, наша модель должна получить признаки, а это значит, нам требуется механизм передачи определенных данных из одного сервиса (сервиса кампаний) в другой (сервис оттока).

В качестве формата обмена данными веб-сервисы обычно используют JSON (объектную нотацию JavaScript). Это похоже на словари, которые мы определяем в Python:

```
{
    "customerid": "8879-zkjof",
    "gender": "female",
    "seniorcitizen": 0,
    "partner": "no",
    "dependents": "no",
    ...
}
```

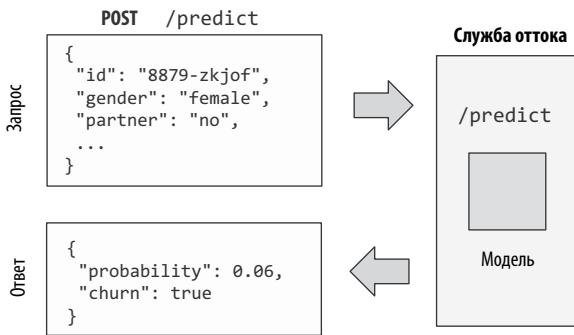
Для отправки данных мы используем запросы POST, а не GET: запросы POST могут включать в запрос данные, тогда как GET нет.

Таким образом, чтобы сервис кампаний мог получать прогнозы от сервиса оттока, нам нужно создать маршрут /predict, который принимает запросы POST. Сервис оттока проанализирует данные о клиенте в формате JSON и также ответит в формате JSON (рис. 5.4).

Теперь мы знаем, что хотим сделать, поэтому приступим к изменению файла `churn_serving.py`.

Сначала мы добавим еще несколько импортированных файлов в начальной части:

```
from flask import Flask, request, jsonify
```



**Рис. 5.4.** Чтобы получить прогнозы, мы выполняем POST с данными о клиенте в формате JSON по маршруту /predict и получаем в ответ вероятность оттока

Если раньше мы импортировали только `Flask`, то теперь нам понадобятся еще два элемента:

- `request` — для получения содержимого запроса POST;
- `jsonify` — чтобы ответить с помощью JSON.

Далее создайте приложение Flask. Назовем его `churn`:

```
app = Flask('churn')
```

Теперь необходимо создать функцию, которая:

- получает данные клиента в запросе;
- вызывает `predict_simple` для оценки клиента;
- отвечает вероятностью оттока в формате JSON.

Мы назовем эту функцию `predict` и назначим ее маршруту `/predict`:

```

@app.route('/predict', methods=['POST']) ← Назначает маршрут
def predict():                                /predict функции predict
    customer = request.get_json() ←           Получает содержимое
                                                запроса в JSON
    prediction = predict_simple(customer, dv, model) ← Оценивает
                                                        клиента
    churn = prediction >= 0.5
    result = {                                 Подготавливает ответ
        'churn_probability': float(prediction),
        'churn': bool(churn),
    }
    return jsonify(result) ← Преобразует ответ в JSON

```

Чтобы назначить маршрут функции, мы используем декоратор `@app.route`, где также предписываем Flask ожидать только POST-запросы.

Основное содержание функции `predict` аналогично тому, что было в сценарии ранее: она принимает клиента, передает его в `predict_single` и выполняет определенную работу с результатом.

Наконец добавим последние две строки, необходимые для запуска приложения Flask:

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=9696)
```

Все готово к запуску:

```
python churn_serving.py
```

После него мы должны увидеть сообщение о том, что приложение запущено и ожидает входящих запросов:

```
* Serving Flask app "churn" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production
  deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:9696/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
```

Протестировать данный код несколько сложнее: на этот раз нам придется использовать POST-запросы и включить информацию о клиенте в тело запроса.

Самый простой способ сделать это – использовать библиотеку `requests` в Python. Она также поставляется с Anaconda, но если вы используете другой дистрибутив, то можете установить ее с помощью `pip`:

```
pip install requests
```

Мы можем открыть тот же блокнот Jupyter, который использовали ранее, и протестировать веб-сервис оттуда.

Первым делом импортируем `requests`:

```
import requests
```

Теперь сделаем POST-запрос к нашему сервису:

```
url = 'http://localhost:9696/predict'
response = requests.post(url, json=customer)
result = response.json()
```

Переменная `results` содержит ответ от сервиса оттока:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Это та же информация, которую мы ранее наблюдали в терминале, но теперь мы получили ее в качестве ответа от веб-сервиса.

#### ПРИМЕЧАНИЕ

Некоторые инструменты, такие как Postman (<https://www.postman.com/>), облегчают тестирование веб-сервисов. В этой книге мы не рассказываем о Postman, но вы можете изучить его самостоятельно.

Если бы сервис кампаний использовал Python, то именно так он мог бы взаимодействовать с сервисом оттока и принимать решение о том, кому отправлять рекламные письма.

С помощью всего нескольких строк кода мы написали веб-сервис, который работает на нашем ноутбуке. В следующем разделе мы увидим, как управлять зависимостями в нашем сервисе и подготовить его к развертыванию.

## 5.3. УПРАВЛЕНИЕ ЗАВИСИМОСТЯМИ

Для локальной разработки дистрибутив Anaconda — идеальный инструмент: в нем есть почти все библиотеки, которые нам когда-либо могут понадобиться. Но есть и обратная сторона: при распаковке он занимает 4 Гбайт, что очень много. Для производственного этапа нам хотелось бы иметь только те библиотеки, которые действительно нужны.

Кроме того, разные сервисы предъявляют разные требования. Часто эти требования конфликтуют, поэтому мы не можем использовать одну и ту же среду для одновременного запуска нескольких сервисов.

В этом разделе мы увидим, как управлять зависимостями нашего приложения изолированным способом, который не мешает работе других сервисов. Мы рассмотрим два инструмента: Pipenv для управления библиотеками Python и Docker для управления системными зависимостями, такими как операционная система и системные библиотеки.

### 5.3.1. Pipenv

Чтобы обслуживать модель оттока, нам нужно всего несколько библиотек: NumPy, Scikit-learn и Flask. Таким образом, вместо того чтобы загружать весь дистрибутив Anaconda со всеми его библиотеками, мы можем взять новую инсталляцию Python и с помощью `pip` установить только нужные библиотеки:

```
pip install numpy scikit-learn flask
```

Но прежде на мгновение задумаемся о том, что происходит, когда мы используем `pip` для установки библиотеки:

- мы запускаем `pip install library` для установки пакета Python под названием `Library` (предположим, такой пакет существует);
- `pip` связывается с PyPI.org (индекс пакетов Python — репозиторий с пакетами Python), после чего получает и устанавливает последнюю версию данной библиотеки. Допустим, это версия 1.0.0.

Когда ее установка будет завершена, мы разрабатываем и тестируем наш сервис, используя именно эту версию. Все работает отлично. Позже наши коллеги решают помочь нам с проектом, поэтому также запускают `pip install`, чтобы настроить все уже на своей машине — за исключением того, что на этот раз последней версией оказывается 1.3.1.

Если нам не повезет, то версии 1.0.0 и 1.3.1 могут быть несовместимы друг с другом, а это означает, что код, который мы написали для версии 1.0.0, не будет работать для версии 1.3.1.

Эту проблему можно решить, указав точную версию библиотеки при ее установке с помощью `pip`:

```
pip install library==1.0.0
```

К сожалению, здесь возможна другая проблема: что если у некоторых наших коллег уже установлена версия 1.3.1 и они уже использовали ее для каких-то других сервисов? В таком случае они не смогут вернуться к использованию версии 1.0.0: это может привести к тому, что их код перестанет работать.

Мы можем решить эти проблемы, создав *виртуальную среду* для каждого проекта — отдельный дистрибутив Python, в котором нет ничего, кроме библиотек, необходимых для этого конкретного проекта.

Pipenv — как раз тот инструмент, который упрощает управление виртуальными средами. Мы можем установить его с помощью `pip`:

```
pip install pipenv
```

После этого мы используем `pipenv` вместо `pip` для установки зависимостей:

```
pipenv install numpy scikit-learn flask
```

При его запуске мы увидим, что сначала он настраивает виртуальную среду, а уже затем устанавливает библиотеки:

```
✓ Successfully created virtual environment!
Virtualenv location: ...
Creating a Pipfile for this project...
Installing numpy...
Adding numpy to Pipfile's [packages]...
```

```

✓ Installation Succeeded
Installing scikit-learn...
Adding scikit-learn to Pipfile's [packages]...
✓ Installation Succeeded
Installing flask...
Adding flask to Pipfile's [packages]...
✓ Installation Succeeded
Pipfile.lock not found, creating...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
*: Locking...

```

После завершения установки будут созданы два файла: `Pipenv` и `Pipenv.lock`.

Файл `Pipenv` выглядит довольно просто:

```

[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]
numpy = "*"
scikit-learn = "*"
flask = "*"

[requires]
python_version = "3.7"

```

Мы видим, что он содержит список библиотек, а также используемую версию Python.

Другой файл — `Pipenv.lock` — содержит конкретные версии библиотек, использованные для проекта. Файл слишком велик, чтобы показать его полностью, так что взглянем на одну из записей:

```

"flask": {
    "hashes": [
        "sha256:4efa1ae2d7c9865af48986de8aeb8504...",
        "sha256:8a4fdd8936eba2512e9c85df320a37e6..."
    ],
    "index": "pypi",
    "version": "==1.1.2"
}

```

Как мы можем видеть, в нем записана точная версия библиотеки, которая использовалась во время установки. Чтобы убедиться, что библиотека не изменится, она также сохраняет хеши — контрольные суммы, которые можно использовать для гарантии того, что в будущем мы скачаем точно такую же версию. Таким

образом, мы «блокируем» зависимости от определенных версий. Так можно гарантировать, что в будущем нас не ждут сюрпризы с двумя несовместимыми версиями одной и той же библиотеки.

Если кому-то нужно поработать над нашим проектом, то этому человеку просто нужно выполнить команду `install`:

```
pipenv install
```

На этом шаге сначала будет создана виртуальная среда, а затем установлены все необходимые библиотеки из `Pipenv.lock`.

### **ВАЖНО**

Блокировка версии библиотеки важна для воспроизводимости в будущем и помогает избежать неприятных сюрпризов, связанных с несовместимостью кода.

После того как все библиотеки установлены, необходимо активировать виртуальную среду — таким образом, наше приложение будет использовать правильные версии библиотек. Это можно проделать, выполнив команду `shell`:

```
pipenv shell
```

Она сообщает нам о том, что работает в виртуальной среде:

```
Launching subshell in virtual environment...
```

Теперь мы можем обеспечить доступ к нашему сервису:

```
python churn_serving.py
```

В качестве альтернативы, вместо того чтобы сначала явно входить в виртуальную среду, а затем запускать сценарий, мы можем проделать эти два шага с помощью всего одной команды:

```
pipenv run python churn_serving.py
```

Команда `run` в `Pipenv` просто запускает указанную программу в виртуальной среде.

Независимо от способа запуска мы должны увидеть точно такой же результат, как и ранее:

```
* Serving Flask app "churn" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production
  deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:9696/ (Press CTRL+C to quit)
```

Выполнив тест с помощью запросов, мы видим тот же результат:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Скорее всего, вы также заметили следующее предупреждение в консоли:

```
* Environment: production
WARNING: This is a development server. Do not use it in a production
deployment.
Use a production WSGI server instead.
```

Встроенный веб-сервер Flask действительно предназначен только для разработки: с его помощью очень просто тестировать наше приложение, но он не сможет обеспечить надежную работу под нагрузкой. Вместо этого придется использовать соответствующий сервер WSGI, что и следует из предупреждения.

WSGI расшифровывается как *интерфейс шлюза веб-сервера* (web server gateway interface) и представляет собой спецификацию, описывающую, как приложения Python должны обрабатывать HTTP-запросы. Тонкости работы WSGI в рамках нашей книги не важны, поэтому мы не будем рассматривать их подробно.

Тем не менее мы уберем данное предупреждение, установив рабочий сервер WSGI. Для этого в Python есть несколько возможных вариантов, мы же используем Gunicorn.

### **ПРИМЕЧАНИЕ**

Gunicorn не работает в Windows: в его основе функции, специфичные для Linux и Unix (включая macOS). Хорошей альтернативой, которая работает и в Windows, является Waitress. Далее мы будем использовать Docker, который решает возникшую проблему и запускает Linux внутри контейнера.

Установим Gunicorn с помощью Pipenv:

```
pipenv install gunicorn
```

Команда устанавливает библиотеку и включает ее в качестве зависимости в проект, добавляя в файлы Pipenv и Pipenv.lock.

Запустим наше приложение с помощью Gunicorn:

```
pipenv run gunicorn --bind 0.0.0.0:9696 churn_serving:app
```

Если все хорошо, то мы должны увидеть в терминале следующие сообщения:

```
[2020-04-13 22:58:44 +0200] [15705] [INFO] Starting gunicorn 20.0.4
[2020-04-13 22:58:44 +0200] [15705] [INFO] Listening at: http://0.0.0.0:9696
(15705)
[2020-04-13 22:58:44 +0200] [15705] [INFO] Using worker: sync
[2020-04-13 22:58:44 +0200] [16541] [INFO] Booting worker with pid: 16541
```

В отличие от встроенного веб-сервера Flask, Gunicorn готов к промышленной эксплуатации, поэтому у него не возникнет никаких проблем с нагрузкой при реальном использовании.

Если мы протестируем его с тем же кодом, что и ранее, то увидим тот же ответ:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Pipenv — отличный инструмент для управления зависимостями: он изолирует необходимые библиотеки в отдельную среду, тем самым помогая избежать конфликтов между различными версиями одного и того же пакета.

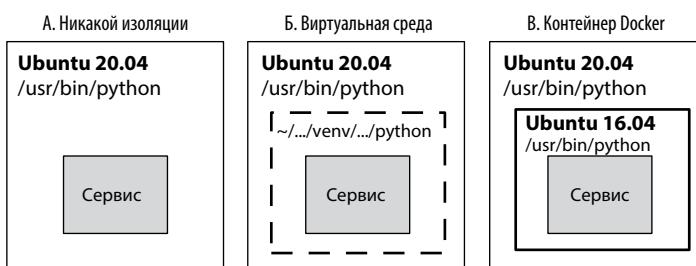
В следующем подразделе мы рассмотрим Docker, позволяющий нам еще больше изолировать приложение и обеспечить его бесперебойную работу, где бы его ни применяли.

### 5.3.2. Docker

Мы уже знаем, как управлять зависимостями Python с помощью Pipenv. Однако некоторые зависимости существуют за пределами Python. Наиболее важно то, что эти зависимости включают операционную систему, а также системные библиотеки.

Например, для разработки нашего сервиса мы могли бы использовать Ubuntu 16.04, но если некоторые из наших коллег используют Ubuntu 20.04, то могут столкнуться с проблемами при попытке запустить сервис на своем ноутбуке.

Docker решает проблему под названием «у меня все работает», упаковывая в контейнер Docker ОС и системные библиотеки — автономную среду, которая работает везде, где установлен Docker (рис. 5.5).

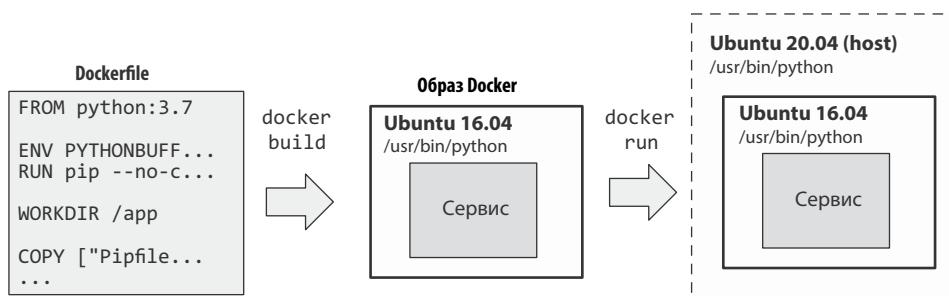


**Рис. 5.5.** В отсутствие изоляции (A) сервис запускается с помощью системного Python. В виртуальных средах (Б) мы изолируем зависимости нашего сервиса внутри среды. В контейнерах Docker (В) мы изолируем всю среду сервиса, включая ОС и системные библиотеки

Как только сервис упакован в контейнер Docker, мы можем запустить его на *хост-компьютере* — нашем ноутбуке (независимо от операционной системы) или любом общедоступном облачном провайдере.

Посмотрим, как применить его для нашего проекта. Мы предполагаем, что Docker уже установлен. Подробную информацию о том, как его установить, можно найти в приложении А.

Сначала нам нужно создать *образ Docker* — описание нашего сервиса, которое включает в себя все настройки и зависимости. Позже Docker будет использовать образ для создания контейнера. Для этого нам понадобится Dockerfile — файл с инструкциями по созданию образа (рис. 5.6).



**Рис. 5.6.** Мы создаем образ, используя инструкции из Dockerfile.  
После этого сможем запустить этот образ на хост-компьютере

Создадим файл с именем Dockerfile и следующим содержимым (обратите внимание, что файл не должен содержать комментарии):

```
FROM python:3.7-slim ←———— Задает базовый образ
ENV PYTHONUNBUFFERED=TRUE ←———— Устанавливает специальные
                                настройки Python для просмотра логов
RUN pip --no-cache-dir install pipenv ←———— Устанавливает Pipenv
WORKDIR /app ←———— Устанавливает в качестве рабочего каталога /app
COPY [\"Pipfile\", \"Pipfile.lock\", \".\"] ←———— Копирует файлы Pipenv
RUN pipenv install --deploy --system && \
    rm -rf /root/.cache ←———— Устанавливает зависимости
                            из файлов Pipenv
COPY [\"*.py\", \"churn-model.bin\", \".\"] ←———— Копирует наш код, а также модель
EXPOSE 9696 ←———— Открывает порт, который используется
                    нашим веб-сервисом      Указывает, как должен
                                            быть запущен сервис
ENTRYPOINT [\"gunicorn\", \"--bind\", \"0.0.0.0:9696\", \"churn_serving:app\"] ←————
```

Это большой объём информации, особенно если вы никогда ранее не сталкивались с файлами Dockerfile. Пройдем по нему строка за строкой.

Сначала мы указываем базовый образ Docker:

```
FROM python:3.7.5-slim
```

Мы используем этот образ в качестве отправной точки и создаем на его основе собственный. Как правило, базовый образ уже содержит ОС и системные библиотеки, такие как сам Python, поэтому нам лишь нужно задать зависимости нашего проекта. В нашем случае мы используем `python:3.7.5-slim`, который основан на Debian 10.2 и содержит Python версии 3.7.5 и pip. Вы можете прочитать больше о базовом образе Python в Docker hub ([https://hub.docker.com/\\_/python](https://hub.docker.com/_/python)) — сервисе для обмена образами Docker.

Все файлы Dockerfile должны начинаться с инструкции `FROM`.

Далее мы устанавливаем для переменной среды `PYTHONUNBUFFERED` значение `TRUE`:

```
ENV PYTHONUNBUFFERED=TRUE
```

Без этого параметра мы не сможем просматривать логи при запуске сценариев Python внутри Docker.

Затем мы используем `pip` для установки Pipenv:

```
RUN pip --no-cache-dir install pipenv
```

Инструкция `RUN` в Docker просто запускает команду оболочки. По умолчанию `pip` сохраняет библиотеки в кэше, чтобы позже можно было ускорить их установку. В контейнере Docker нам это не требуется, поэтому мы используем параметр `--no-cache-dir`.

Затем следует указать рабочий каталог:

```
WORKDIR /app
```

Это примерно эквивалентно команде `cd` в Linux (смена каталога), поэтому все, что мы запустим после этого, будет выполняться в папке `/app`.

Затем мы копируем файлы Pipenv в текущий рабочий каталог (то есть `/app`):

```
COPY ["Pipfile", "Pipfile.lock", "./"]
```

Мы используем эти файлы для установки зависимостей с помощью Pipenv:

```
RUN pipenv install --deploy --system && \
    rm -rf /root/.cache
```

Ранее для этого мы просто использовали `pipenv install`. Здесь включаем два дополнительных параметра: `--deploy` и `--system`. Внутри Docker нам не нужно

создавать виртуальную среду — наш контейнер Docker уже изолирован от остальной системы. Установка этих параметров позволяет пропустить создание виртуальной среды и использовать системный Python для установки всех зависимостей.

После установки библиотек мы очищаем кэш, чтобы наш образ Docker не оказался слишком большим.

Затем копируем файлы нашего проекта, а также сериализованную модель:

```
COPY ["*.py", "churn-model.bin", "./"]
```

Далее мы указываем, какой порт будет использоваться нашим приложением. В нашем случае это 9696:

```
EXPOSE 9696
```

Наконец, мы сообщаем Docker, как запускать наше приложение:

```
ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "churn_serving:app"]
```

Это та же команда, которую мы использовали ранее при локальном запуске Gunicorn.

Создадим образ. Для этого запустим команду `build` в Docker:

```
docker build -t churn-prediction .
```

Флаг `-t` позволяет задать дескриптор для образа, и конечный параметр — точка — указывает каталог с файлом Dockerfile. В нашем случае это текущий каталог.

Первое, что сделает Docker после запуска, — загрузит базовый образ:

```
Sending build context to Docker daemon 51.71kB
Step 1/11 : FROM python:3.7.5-slim
3.7.5-slim: Pulling from library/python
000eee12ec04: Downloading 24.84MB/27.09MB
ddc2d83f8229: Download complete
735b0bee82a3: Downloading 19.56MB/28.02MB
8c69dcfedfc84: Download complete
495e1cccc7f9: Download complete
```

Затем он выполнит каждую строку файла Dockerfile одну за другой:

```
Step 2/9 : ENV PYTHONUNBUFFERED=TRUE
--> Running in d263b412618b
Removing intermediate container d263b412618b
--> 7987e3cf611f
Step 3/9 : RUN pip --no-cache-dir install pipenv
--> Running in e8e9d329ed07
Collecting pipenv
...

```

В конце Docker сообщит нам, что он успешно создал образ и пометил его как `churn-prediction:latest`:

```
Successfully built d9c50e4619a1
Successfully tagged churn-prediction:latest
```

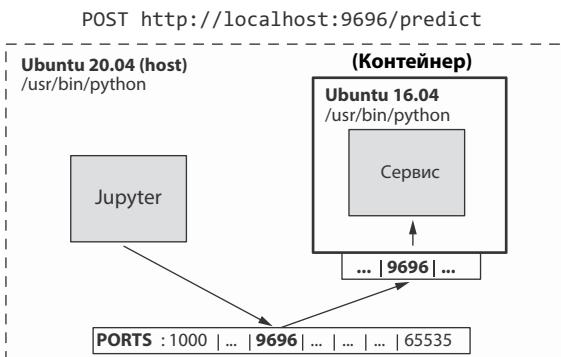
Мы готовы использовать этот образ для запуска контейнера Docker. Используем для этого команду `run`:

```
docker run -it -p 9696:9696 churn-prediction:latest
```

Здесь мы указываем несколько параметров:

- флаг `-it` сообщает Docker, что мы запускаем его с нашего терминала и нам нужно увидеть результаты;
- параметр `-p` задает сопоставление портов. `9696:9696` означает сопоставление порта `9696` в контейнере с портом `9696` на хост-компьютере;
- наконец нам нужно название образа и дескриптор, который в нашем случае `churn-prediction:latest`.

Теперь наш сервис запущен внутри контейнера Docker и мы можем подключиться к нему, используя порт `9696` (рис. 5.7). Это тот же порт, который мы использовали ранее.



**Рис. 5.7.** Порт `9696` на хост-компьютере сопоставлен с портом `9696` контейнера, поэтому, когда мы отправляем запрос на `localhost:9696`, он обрабатывается нашим сервисом в Docker

Проведем тестирование, используя тот же код. В поле запуска мы увидим тот же ответ:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Docker упрощает запуск сервисов воспроизводимым способом. С помощью Docker среда внутри контейнера всегда остается неизменной. Это означает,

что если мы сможем запустить наш сервис на ноутбуке, то он будет работать и в любом другом месте.

Мы уже протестирували наше приложение на своем ноутбуке, так что теперь узнаем, как запустить его в общедоступном облаке и развернуть в AWS.

## 5.4. РАЗВЕРТЫВАНИЕ

Мы не запускаем производственные сервисы на наших ноутбуках; для этого требуются специальные серверы.

В этом разделе мы рассмотрим один из возможных вариантов: Amazon Web Services, или AWS. Мы решили выбрать AWS из-за его популярности и никак не связаны с Amazon или AWS.

Другие популярные общедоступные облачные сервисы включают Google Cloud, Microsoft Azure и Digital Ocean. Мы не рассматриваем их в этой книге, но вы сможете найти аналогичные инструкции в Интернете и развернуть модель у вашего любимого облачного провайдера.

Этот раздел необязателен, поэтому его можно смело пропустить. Чтобы следовать инструкциям в нем, необходимо иметь учетную запись AWS и настроить командную строку AWS (CLI). В приложении А описано, как это сделать.

### 5.4.1. AWS Elastic Beanstalk

AWS предоставляет множество сервисов, и у нас есть много возможных способов развертывания на них веб-сервиса. Например, вы можете арендовать компьютер EC2 (сервер в AWS) и настроить сервис вручную, использовать «бессерверный» подход с AWS Lambda или воспользоваться рядом других сервисов.

В этом разделе мы будем использовать AWS Elastic Beanstalk, который является одним из самых простых способов развертывания модели в AWS. Кроме того, наш сервис достаточно прост, поэтому можно действовать в рамках бесплатного доступа. Другими словами, мы можем пользоваться им бесплатно в течение первого года.

Elastic Beanstalk автоматически берет на себя многие задачи, которые обычно встают при промышленной эксплуатации, включая:

- развертывание нашего сервиса в экземплярах EC2;
- масштабирование — добавление большего количества экземпляров для обработки нагрузки в часы пик;
- уменьшение масштаба — удаление этих экземпляров, когда нагрузка исчезает;

- перезапуск сервиса, если по какой-либо причине возникает сбой;
- балансировку нагрузки между экземплярами.

Использование Elastic Beanstalk также потребует специальной утилиты, а именно интерфейса командной строки Elastic Beanstalk (CLI). Интерфейс написан на Python, поэтому мы можем установить его с помощью pip, как и любой другой инструмент Python.

Однако, поскольку мы используем Pipenv, мы можем добавить его как рабочую зависимость. Таким образом, мы установим его только для нашего проекта, а не для всей системы.

```
pipenv install awsebcli --dev
```

#### **ПРИМЕЧАНИЕ**

Рабочие зависимости — это инструменты и библиотеки, которые мы используем для разработки нашего приложения. Обычно они требуются нам только локально и не нужны в фактическом пакете, развернутом в производстве.

После установки Elastic Beanstalk мы можем войти в виртуальную среду нашего проекта:

```
pipenv shell
```

Теперь CLI должен быть доступен. Проверим:

```
eb --version
```

На экране должна появиться версия:

```
EB CLI 3.18.0 (Python 3.7.7)
```

Далее запустим команду инициализации:

```
eb init -p docker churn-serving
```

Обратите внимание, что мы используем `-p docker`: таким образом мы указываем, что это проект на основе Docker.

Если все в порядке, то будет создана пара файлов, включая файл config.yml в папке .elasticbeanstalk.

Теперь мы можем протестировать наше приложение локально, используя команду local run:

```
eb local run --port 9696
```

Работать все должно так же, как и в предыдущем разделе с Docker: сначала будет создан образ, а затем запущен контейнер.

Чтобы это проверить, мы можем использовать тот же код, что и ранее, получив тот же ответ:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Убедившись, что сервис хорошо работает локально, мы готовы развернуть его в AWS. Это можно сделать с помощью единственной команды:

```
eb create churn-serving-env
```

Эта простая команда настраивает все, что нам нужно, от экземпляров EC2 до правил автоматического масштабирования:

```
Creating application version archive "app-200418_120347".
Uploading churn-serving/app-200418_120347.zip to S3. This may take a while.
Upload Complete.
Environment details for: churn-serving-env
  Application name: churn-serving
  Region: us-west-2
  Deployed Version: app-200418_120347
  Environment ID: e-3xkqdzdjbq
  Platform: arn:aws:elasticbeanstalk:us-west-2::platform/Docker running on
             64bit Amazon Linux 2/3.0.0
  Tier: WebServer-Standard-1.0
  CNAME: UNKNOWN
  Updated: 2020-04-18 10:03:52.276000+00:00
Printing Status:
2020-04-18 10:03:51    INFO    createEnvironment is starting.
-- Events -- (safe to Ctrl+C)
```

На создание всего этого уйдет несколько минут. Мы можем отслеживать процесс, наблюдая за происходящим через терминал.

Когда все будет готово, мы должны увидеть следующую информацию:

```
2020-04-18 10:06:53    INFO    Application available at churn-serving-
env.5w9pp7bkmj.us-west-2.elasticbeanstalk.com.
2020-04-18 10:06:53    INFO    Successfully launched environment: churn-
serving-env
```

URL (`churn-serving-env.5w9pp7bkmj.us-west-2.elasticbeanstalk.com`) в логах важен: именно так мы попадаем в наше приложение. Теперь мы можем использовать этот URL для составления прогнозов (рис. 5.8).

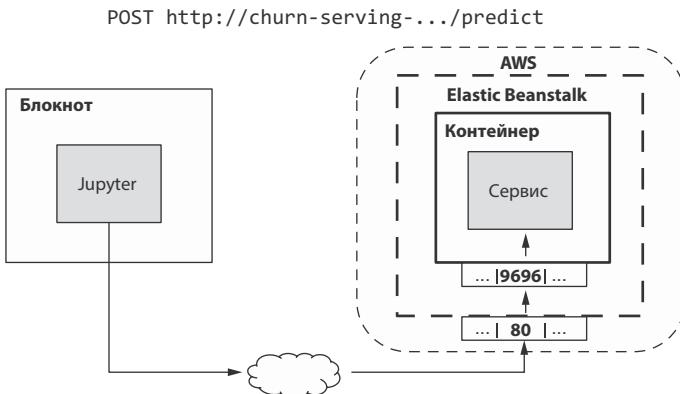
Проведем тест:

```
host = 'churn-serving-env.5w9pp7bkmj.us-west-2.elasticbeanstalk.com'
url = 'http://%s/predict' % host
response = requests.post(url, json=customer)
result = response.json()
result
```

Как и ранее, мы должны увидеть ответ:

```
{'churn': False, 'churn_probability': 0.05960590758316393}
```

Вот и все! Теперь у нас есть работающий сервис.



**Рис. 5.8.** Наш сервис развернут внутри контейнера на AWS Elastic Beanstalk. Чтобы связаться с ним, мы используем его общедоступный URL

### ВНИМАНИЕ

Это игровой пример, и созданный нами сервис доступен любому человеку в мире. Если вы делаете что-то подобное внутри организации, то доступ должен быть максимально ограничен. Расширить данный пример, чтобы обеспечить безопасность, не трудно, но это выходит за рамки данной книги. Проконсультируйтесь с отделом безопасности вашей компании, прежде чем делать подобное на работе.

Мы можем управлять всем из терминала, используя CLI, но управление возможно и из консоли AWS. Для этого мы находим там Elastic Beanstalk и выбираем только что созданную среду (рис. 5.9).

Чтобы отключить ее, выберите Terminate deployment в меню Environment actions с помощью консоли AWS.

### ВНИМАНИЕ

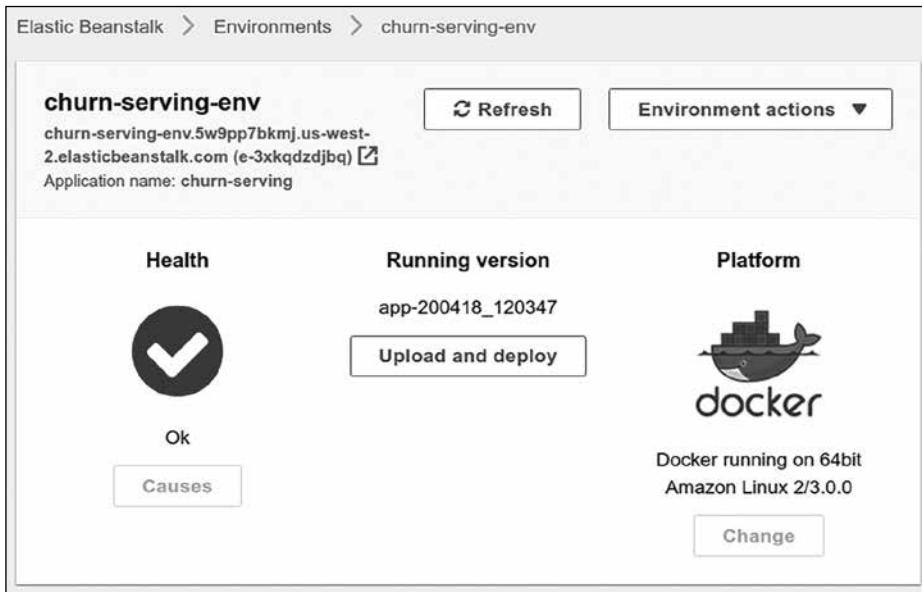
Несмотря на то что работа Elastic Beanstalk для нас ничего не стоит, мы всегда должны проявлять осторожность и отключать его, как только он нам больше не нужен.

В качестве альтернативы используем для этого CLI:

```
eb terminate churn-serving-env
```

Через несколько минут развертывание будет удалено из AWS и URL больше не будет доступен.

AWS Elastic Beanstalk — отличный инструмент для начала работы с моделями машинного обучения. Более продвинутые способы сделать это включают системы оркестровки контейнеров, такие как AWS ECS или Kubernetes, а также «бессерверный» вариант с AWS Lambda. Мы вернемся к этой теме в главах 8 и 9, когда будем изучать развертывание моделей глубокого обучения.



**Рис. 5.9.** Мы можем управлять средой Elastic Beanstalk в консоли AWS

## 5.5. СЛЕДУЮЩИЕ ШАГИ

Мы узнали о Pipenv и Docker и развернули нашу модель в AWS Elastic Beanstalk. Поработайте самостоятельно, чтобы расширить свои навыки.

### 5.5.1. Упражнения

Попробуйте выполнить следующие упражнения, чтобы углубить знания по теме развертывания модели:

- если вы не используете AWS, то попробуйте повторить действия из раздела 5.4 на любом другом облачном провайдере. Например, можно использовать Google Cloud, Microsoft Azure, Heroku или Digital Ocean;
- Flask — не единственный способ создания веб-сервисов на Python. Вы можете попробовать альтернативные фреймворки, такие как FastAPI (<https://fastapi.io>).

[tiangolo.com/](https://tiangolo.com/)), Bottle (<https://github.com/bottlepy/bottle>) или Falcon (<https://github.com/falconry/falcon>).

### 5.5.2. Другие проекты

Вы можете продолжить другие проекты из предыдущих глав и сделать доступными в виде веб-сервиса уже их. Примеры указаны ниже.

- Модель прогнозирования цен на автомобили, которую мы создали в главе 2.
- Проекты для самостоятельного изучения из главы 3: проект по оценке лидов и проект по прогнозированию дефолта.

## РЕЗЮМЕ

- Pickle — библиотека сериализации/десериализации, встроенная в Python. Мы можем использовать ее для сохранения модели, обученной в Jupyter Notebook, и загрузки ее в сценарий Python.
- Самый простой способ сделать модель доступной другим — поместить ее в веб-сервис Flask.
- Pipenv — инструмент управления зависимостями Python путем создания виртуальных сред, поэтому зависимости одного проекта Python не мешают зависимостям другого.
- Docker позволяет полностью изолировать сервис от других сервисов, упаковывая в контейнер Docker не только зависимости Python, но и системные зависимости, а также саму операционную систему.
- AWS Elastic Beanstalk позволяет легко развертывать веб-сервис. Он занимается об управлении экземплярами EC2, масштабировании сервиса вверх и вниз, а также перезапуске в случае сбоя.

В следующей главе мы продолжим изучение классификации, но с другим типом модели — деревьями решений.



# *Деревья решений и ансамблевое обучение*

---

## **В этой главе**

- ✓ Деревья решений и алгоритм обучения дерева решений.
- ✓ Случайные леса: объединение нескольких деревьев в одну модель.
- ✓ Градиентный бустинг как альтернативный способ объединения деревьев решений.

В главе 3 мы описали проблему бинарной классификации и задействовали модель логистической регрессии для прогнозирования оттока клиентов.

В текущей главе мы также решаем задачу бинарной классификации, но используем другое семейство моделей машинного обучения — древовидные модели. Деревья решений, простейшая древовидная модель, представляют собой не что иное, как последовательность правил «если-то-иначе», составленных вместе. Чтобы улучшить производительность, мы можем объединить несколько деревьев решений в ансамбль. Мы рассмотрим две модели ансамбля на основе деревьев: случайный лес и градиентный бустинг.

Проект, который я подготовил для этой главы, называется «прогнозирование дефолта»: мы прогнозируем, сможет ли клиент вернуть кредит. Мы узнаем, как обучать деревья решений и модели случайного леса с помощью

Scikit-learn и изучим XGBoost — библиотеку для реализации моделей градиентного бустинга.

## 6.1. ПРОЕКТ ПО ОЦЕНКЕ КРЕДИТНОГО РИСКА

Представьте, что вы работаете в банке. При получении заявки на кредит нам нужно убедиться, что в случае ее одобрения клиент сможет его вернуть. Каждая заявка несет в себе риск *дефолта* — невозврата денег.

Мы хотели бы минимизировать этот риск: прежде чем соглашаться на предоставление кредита, стоит оценить клиента и вероятность дефолта. Если она слишком высока, то заявка будет отклонена. Этот процесс называется «оценка кредитного риска».

Машинное обучение вполне может быть использовано для расчета этого риска. Нам потребуется набор данных с кредитами, где для каждой заявки мы будем знать, был ли кредит успешно возвращен. Используя эти данные, мы построим модель для прогнозирования вероятности дефолта и сможем с ее помощью оценивать риск того, что будущие заемщики не вернут деньги.

Итак, в этой главе мы используем машинное обучение для расчета риска дефолта. План проекта состоит в следующем:

- сначала мы получим данные и выполним определенную предварительную обработку;
- далее мы обучим модель дерева решений из Scikit-learn прогнозированию вероятности дефолта;
- после этого мы узнаем, как работают деревья принятия решений и какие параметры имеются у модели, после чего поймем, как настроить эти параметры, чтобы достичь наилучшей производительности;
- затем мы объединим несколько деревьев решений в одну модель — случайный лес. Мы рассмотрим ее параметры и настроим их, чтобы достичь наилучшей прогностической производительности;
- наконец мы исследуем другой способ объединения деревьев решений — градиентный бустинг. Мы используем XGBoost, высокоэффективную библиотеку, которая реализует градиентный бустинг. Мы обучим модель и настроим ее параметры.

Оценка кредитного риска — проблема двоичной классификации: цель положительна (1), если клиент не выполняет обязательства, и отрицательна (0) в противном случае. Оценивать наше решение мы будем с помощью AUC (площадь под кривой ROC), которую рассматривали в главе 4. AUC описывает, насколько хорошо модель может разделять случаи на положительные и отрицательные.

Код для этого проекта доступен в репозитории книги на GitHub по адресу <https://github.com/alexeygrigorev/mlbookcamp-code> (в папке chapter-06-trees).

### 6.1.1. Набор данных для оценки кредитоспособности

Для этого проекта мы используем набор данных из курса интеллектуального анализа данных в Политехническом университете Каталонии (<https://www.cs.upc.edu/~belanche/Docencia/mineria/mineria.html>). Набор данных описывает клиентов (трудовой стаж, возраст, семейное положение, доход и другие характеристики), кредит (запрашиваемая сумма, цена товара) и его статус (возвращен или нет).

Мы используем копию этого набора данных, доступную на GitHub по адресу <https://github.com/gastonstat/CreditScoring/>. Скачаем его.

Сначала создайте папку для нашего проекта (например, chapter-06-credit-risk), а затем используйте `wget` для загрузки:

```
wget https://github.com/gastonstat/CreditScoring/raw/master/CreditScoring.csv
```

В качестве альтернативы вы можете ввести ссылку в свой браузер и сохранить результат в папке проекта.

Затем запустите сервер Jupyter Notebook, если он еще не запущен:

```
jupyter notebook
```

Перейдите в папку проекта и создайте новый блокнот (например, chapter-06-credit-risk).

Как обычно, мы начинаем с импорта Pandas, NumPy, Seaborn и Matplotlib:

```
import pandas as pd
import numpy as np

import seaborn as sns
from matplotlib import pyplot as plt
%matplotlib inline
```

После того как мы нажмем `Ctrl+Enter`, библиотеки будут импортированы, а мы будем готовы читать данные с помощью Pandas:

```
df = pd.read_csv('CreditScoring.csv')
```

Данные загружены, поэтому сначала взглянем на них и посмотрим, придется ли нам выполнять какую-либо предварительную обработку, прежде чем использовать их.

### 6.1.2. Очистка данных

Чтобы использовать набор данных для нашей задачи, необходимо выявить любые проблемы в данных и устраниить их.

Начнем с рассмотрения первых строк датафрейма, сгенерированного функцией `df.head()` (рис. 6.1).

df.head()															
	Status	Seniority	Home	Time	Age	Marital	Records	Job	Expenses	Income	Assets	Debt	Amount	Price	
0	1	9	1	60	30	2	1	3	73	129	0	0	800	846	
1	1	17	1	60	58	3	1	1	48	131	0	0	1000	1658	
2	2	10	2	36	46	2	2	3	90	200	3000	0	2000	2985	
3	1	0	1	60	24	1	1	1	63	182	2500	0	900	1325	
4	1	0	1	36	26	1	1	1	46	107	0	0	310	910	

**Рис. 6.1.** Первые пять строк набора данных оценки заемщиков

Мы сразу видим, что все названия столбцов начинаются с заглавной буквы. Прежде чем делать что-либо еще, переведем все названия в нижний регистр, приведя в соответствие с другими проектами (рис. 6.2):

```
df.columns = df.columns.str.lower()
```

df.columns = df.columns.str.lower() df.head()															
	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price	
0	1	9	1	60	30	2	1	3	73	129	0	0	800	846	
1	1	17	1	60	58	3	1	1	48	131	0	0	1000	1658	
2	2	10	2	36	46	2	2	3	90	200	3000	0	2000	2985	
3	1	0	1	60	24	1	1	1	63	182	2500	0	900	1325	
4	1	0	1	36	26	1	1	1	46	107	0	0	310	910	

**Рис. 6.2.** Датафрейм с именами столбцов в нижнем регистре

Мы видим, что датафрейм содержит следующие столбцы:

- `status` — удалось ли клиенту погасить кредит (1) или нет (2);
- `seniority` — стаж работы в годах;

- **home** — тип домовладения: аренда (1), владение (2) и другие;
- **time** — планируемый срок предоставления кредита (в месяцах);
- **age** — возраст клиента;
- **marital [status]** — холост (1), женат (2) и др.;
- **records** — есть ли у клиента какие-либо предыдущие записи: нет (1), да (2) (из описания набора данных неясно, о каких записях здесь идет речь. Учитывая суть проекта, можно предположить, что речь о записях в базе данных банка);
- **job** — тип работы: полный рабочий день (1), неполный рабочий день (2) и другие;
- **expenses** — сколько клиент тратит в месяц;
- **income** — сколько клиент зарабатывает в месяц;
- **assets** — общая стоимость всех активов клиента;
- **debt** — сумма кредитной задолженности;
- **amount** — запрашиваемая сумма кредита;
- **price** — цена товара, который клиент хочет приобрести.

Хотя большинство столбцов числовые, мы видим и категориальные: **status**, **home**, **marital [status]**, **records** и **job**. Однако значения, которые мы наблюдаем в датафрейме, являются числами, а не строками. Это означает, что нам нужно привести их к их настоящим именам. В репозитории GitHub вместе с набором данных имеется сценарий, который декодирует числа по категориям ([https://github.com/gastonstat/CreditScoring/blob/master/Part1\\_CredScoring\\_Processing.R](https://github.com/gastonstat/CreditScoring/blob/master/Part1_CredScoring_Processing.R)). Изначально он был написан на R, поэтому нам нужно перевести его на Pandas. Мы начнем со столбца **status**. Значение 1 означает **OK**, 2 означает **default**, а 0 означает, что значение отсутствует, — заменим его на **unk** (сокращение для **unknown**).

В Pandas для преобразования чисел в строки мы можем использовать **map**. Для этого сначала определим словарь с отображением текущего значения (числа) на желаемое (строку):

```
status_values = {
    1: 'ok',
    2: 'default',
    0: 'unk'
}
```

Теперь можно использовать этот словарь для сопоставления:

```
df.status = df.status.map(status_values)
```

## 230 Глава 6. Деревья решений и ансамблевое обучение

Это создаст новую серию, которую мы немедленно записываем обратно в датагрейм. В результате значения в столбце состояния перезаписываются и выглядят более осмысленными (рис. 6.3).

status_values = { 1: 'ok', 2: 'default', 0: 'unk' }  df.status = df.status.map(status_values) df.head()														
	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price
0	ok	9	1	60	30	2	1	3	73	129	0	0	800	846
1	ok	17	1	60	58	3	1	1	48	131	0	0	1000	1658
2	default	10	2	36	46	2	2	3	90	200	3000	0	2000	2985
3	ok	0	1	60	24	1	1	1	63	182	2500	0	900	1325
4	ok	0	1	36	26	1	1	1	46	107	0	0	310	910

**Рис. 6.3.** Чтобы преобразовать исходные значения в столбце status (числа) в более осмысленное представление (строки), мы используем метод map

Повторим ту же процедуру для всех остальных столбцов. Сначала сделаем это для home:

```
home_values = {  
    1: 'rent',  
    2: 'owner',  
    3: 'private',  
    4: 'ignore',  
    5: 'parents',  
    6: 'other',  
    0: 'unk'  
}  
  
df.home = df.home.map(home_values)
```

Далее проделаем это для столбцов marital, records и job:

```
marital_values = {  
    1: 'single',  
    2: 'married',  
    3: 'widow',  
    4: 'separated',  
    5: 'divorced',  
    0: 'unk'  
}  
  
df.marital = df.marital.map(marital_values)
```

```

records_values = {
    1: 'no',
    2: 'yes',
    0: 'unk'
}

df.records = df.records.map(records_values)

job_values = {
    1: 'fixed',
    2: 'parttime',
    3: 'freelance',
    4: 'others',
    0: 'unk'
}

df.job = df.job.map(job_values)

```

После всех преобразований столбцы с категориальными переменными содержат фактические значения, а не цифры (рис. 6.4).

df.head()																
	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price		
0	ok	9	rent	60	30	married	no	freelance	73	129	0	0	800	846		
1	ok	17	rent	60	58	widow	no	fixed	48	131	0	0	1000	1658		
2	default	10	owner	36	46	married	yes	freelance	90	200	3000	0	2000	2985		
3	ok	0	rent	60	24	single	no	fixed	63	182	2500	0	900	1325		
4	ok	0	rent	36	26	single	no	fixed	46	107	0	0	310	910		

**Рис. 6.4.** Значения категориальных переменных преобразуются из целых чисел в строки

В качестве следующего шага рассмотрим числовые столбцы. Сначала проверим сводную статистику по каждому из столбцов: минимум, среднее, максимум и другие показатели. Для этого мы можем использовать метод `describe` из датафрейма:

```
df.describe().round()
```

### ПРИМЕЧАНИЕ

Выходные данные `describe` могут ввести в заблуждение. В нашем случае это значения в научной системе счисления, такие как  $1,000000e+08$  или  $8,703625e+06$ . Чтобы заставить Pandas использовать другую систему счисления, мы используем `round`: метод удаляет дробную часть и округляет до ближайшего целого числа.

Все это дает нам представление о том, как выглядит распределение значений в каждом столбце (рис. 6.5).

df.describe().round()									
	seniority	time	age	expenses	income	assets	debt	amount	price
count	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0
mean	8.0	46.0	37.0	56.0	763317.0	1060341.0	404382.0	1039.0	1463.0
std	8.0	15.0	11.0	20.0	8703625.0	10217569.0	6344253.0	475.0	628.0
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0	100.0	105.0
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0	700.0	1118.0
50%	5.0	48.0	36.0	51.0	120.0	3500.0	0.0	1000.0	1400.0
75%	12.0	60.0	45.0	72.0	166.0	6000.0	0.0	1300.0	1692.0
max	48.0	72.0	68.0	180.0	99999999.0	99999999.0	99999999.0	5000.0	11140.0

**Рис. 6.5.** Сводка по всем числовым столбцам датафрейма. Мы замечаем, что некоторые из них содержат максимальное значение 99999999

Мы сразу замечаем, что в некоторых случаях максимальное значение равно 99999999. Это довольно подозрительно. Как оказалось, это искусственное значение — именно так кодируются недостающие значения в этом наборе данных.

Эта проблема наблюдается в трех столбцах: `income`, `assets` и `debt`. Заменим для них это большое число на `Nan`:

```
for c in ['income', 'assets', 'debt']:
    df[c] = df[c].replace(to_replace=99999999, value=np.nan)
```

Мы используем метод `replace`, который принимает два значения:

- `to_replace` — исходное значение (в нашем случае 99999999);
- `value` — целевое значение (в нашем случае `Nan`).

После этого преобразования в сводке больше не фигурируют подозрительные суммы (рис. 6.6).

df.describe().round()									
	seniority	time	age	expenses	income	assets	debt	amount	price
count	4455.0	4455.0	4455.0	4455.0	4421.0	4408.0	4437.0	4455.0	4455.0
mean	8.0	46.0	37.0	56.0	131.0	5403.0	343.0	1039.0	1463.0
std	8.0	15.0	11.0	20.0	86.0	11573.0	1246.0	475.0	628.0
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0	100.0	105.0
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0	700.0	1118.0
50%	5.0	48.0	36.0	51.0	120.0	3000.0	0.0	1000.0	1400.0
75%	12.0	60.0	45.0	72.0	165.0	6000.0	0.0	1300.0	1692.0
max	48.0	72.0	68.0	180.0	959.0	300000.0	30000.0	5000.0	11140.0

**Рис. 6.6.** Сводная статистика после замены больших значений на `Nan`

Прежде чем закончить подготовку набора данных, посмотрим на нашу целевую переменную `status`:

```
df.status.value_counts()
```

Метод `value_counts` показывает количество появлений каждого значения:

```
ok        3200  
default   1254  
unk         1  
Name: status, dtype: int64
```

Обратите внимание, что имеется одна строка со статусом `unknown`: мы не знаем, удалось ли этому клиенту погасить кредит. Для нашего проекта она бесполезна, поэтому сразу удалим ее из набора данных:

```
df = df[df.status != 'unk']
```

В данном случае мы не удаляем ее по-настоящему. Вместо этого мы создаем новый датафрейм, в котором нет записей со статусом `unknown`.

Изучив данные, мы выявили в них несколько важных проблем и решили их.

Для этого проекта мы пропустим более подробный предварительный анализ данных (который мы выполняли в главе 2 для проекта прогнозирования цен на автомобили и в главе 3 для проекта прогнозирования оттока), но вы можете повторить шаги оттуда и применительно к текущему проекту.

### 6.1.3. Подготовка набора данных

Теперь, когда наш набор данных очищен, мы практически готовы использовать его для обучения модели. Но прежде необходимо предпринять еще несколько шагов:

- разделить набор данных на обучающие, проверочные и тестовые;
- обработать пропущенные значения;
- использовать прямое кодирование для категориальных переменных;
- создать матрицу признаков  $X$  и целевую переменную  $y$ .

Начнем с разделения данных. Мы разделим их на три части:

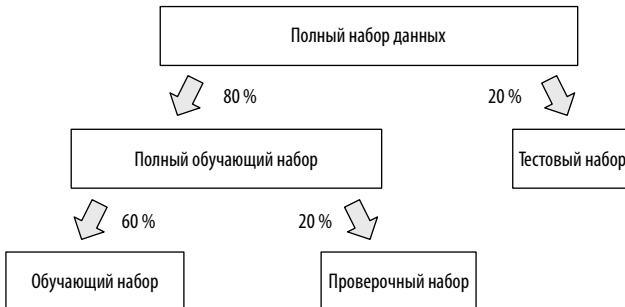
- обучающие (60 % от исходного набора);
- данные для проверки (20 %);
- тестовые (20 %).

Как и ранее, используем для этого `train_test_split` из Scikit-learn. Поскольку мы не можем разделить данные сразу на три набора одновременно, нам придется

делить их дважды (рис. 6.7). Сначала мы оставим 20 % данных для тестирования, а затем разделим оставшиеся 80 % на обучающие и проверочные:

```
from sklearn.model_selection import train_test_split

df_train_full, df_test = train_test_split(df, test_size=0.2, random_state=11)
df_train, df_val = train_test_split(df_train_full, test_size=0.25,
                                     random_state=11)
```



**Рис. 6.7.** Поскольку `train_test_split` может разделить набор данных только на две части (а нам нужно три), мы выполняем разделение дважды

При разделении во второй раз мы отделяем 25 % данных вместо 20 % (`test_size=0.25`). Поскольку `df_train_full` содержит 80 % записей, одна четверть (то есть 25 %) из 80 % соответствует 20 % исходного набора данных.

Чтобы проверить размер наших наборов данных, мы можем использовать функцию `len`:

```
len(df_train), len(df_val), len(df_test)
```

При ее запуске мы получаем следующий вывод:

```
(2672, 891, 891)
```

Итак, мы будем использовать приблизительно 2700 примеров для обучения и почти 900 — для проверки и тестирования.

Результат, который мы хотим спрогнозировать, — это `status`. Мы будем использовать его для обучения модели, так что это и есть наша  $y$  — целевая переменная. Поскольку цель состоит в том, чтобы определить, будет ли кто-то задерживать выплаты, положительным классом является `default`. Это означает, что  $y$  равно 1, если клиент не возвращает заем, и 0 в противном случае. Это довольно просто реализовать:

```
y_train = (df_train.status == 'default').values
y_val = (df_val.status == 'default').values
```

Теперь нам нужно удалить `status` из датафрейма. Если мы этого не сделаем, то можем случайно использовать эту переменную для обучения. Удаляем с помощью оператора `del`:

```
del df_train['status']
del df_val['status']
```

Далее мы позаботимся о  $X$  — матрице признаков.

Из первоначального анализа мы знаем, что наши данные содержат недостающие значения — мы добавляли эти `Nan` самостоятельно. Мы можем заменить недостающие значения нулями:

```
df_train = df_train.fillna(0)
df_val = df_val.fillna(0)
```

Чтобы использовать категориальные переменные, нам придется их закодировать. В главе 3 мы применили для этого метод прямого кодирования. В прямом кодировании каждое значение кодируется как 1, если оно присутствует («горячее») или 0 — если отсутствует («холодное»). Реализовывали мы его с помощью `DictVectorizer` из Scikit-learn.

`DictVectorizer` требуется список словарей, поэтому сначала нам нужно преобразовать датафрейм в этот формат:

```
dict_train = df_train.to_dict(orient='records')
dict_val = df_val.to_dict(orient='records')
```

Каждый словарь в результате представляет собой строку из датафрейма. Например, первая запись в `dict_train` выглядит следующим образом:

```
{'seniority': 10,
 'home': 'owner',
 'time': 36,
 'age': 36,
 'marital': 'married',
 'records': 'no',
 'job': 'freelance',
 'expenses': 75,
 'income': 0.0,
 'assets': 10000.0,
 'debt': 0.0,
 'amount': 1000,
 'price': 1400}
```

Полученный список словарей теперь можно использовать в качестве входных данных для `DictVectorizer`:

```
from sklearn.feature_extraction import DictVectorizer
dv = DictVectorizer(sparse=False)
```

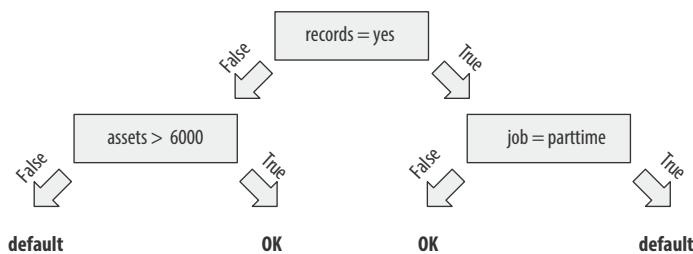
```
X_train = dv.fit_transform(dict_train)
X_val = dv.transform(dict_val)
```

В результате у нас окажутся матрицы признаков как для обучающих, так и для проверочных наборов данных. В главе 3 содержится более подробная информация о выполнении прямого кодирования с помощью Scikit-learn.

Все готово для обучения модели! В следующем разделе мы рассмотрим простейшую древовидную модель: дерево решений.

## 6.2. ДЕРЕВЬЯ РЕШЕНИЙ

*Дерево решений* — это структура данных, которая кодирует ряд правил «если-то-иначе». Каждый узел в дереве содержит условие. Если оно выполнено, то мы переходим на правую сторону дерева; в противном случае — на левую. В конце концов мы приходим к окончательному решению (рис. 6.8).



**Рис. 6.8.** Дерево решений состоит из узлов с условиями. Если условие в узле выполнено, мы идем направо; в противном случае — налево

В Python довольно легко представить дерево решений в виде набора операторов `if-else`. Например:

```
def assess_risk(client):
    if client['records'] == 'yes':
        if client['job'] == 'parttime':
            return 'default'
        else:
            return 'ok'
    else:
        if client['assets'] > 6000:
            return 'ok'
        else:
            return 'default'
```

С помощью машинного обучения мы можем автоматически извлекать эти правила из данных. Посмотрим, как это работает.

## 6.2.1. Классификатор дерева решений

Обучать дерево решений мы будем с помощью Scikit-learn. Поскольку мы решаем задачу классификации, нам нужно использовать `DecisionTreeClassifier` из пакета `tree`. Сразу ее импортируем:

```
from sklearn.tree import DecisionTreeClassifier
```

Обучение модели осуществляется с помощью простого вызова метода `fit`:

```
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)
```

Чтобы оценить качество результата, нам нужно проверить прогностическую производительность модели на проверочном наборе. Используем для этого AUC (площадь под кривой ROC).

Оценка кредитного риска представляет собой проблему бинарной классификации, и для подобных случаев AUC — один из лучших показателей. Как вы, возможно, помните из нашего обсуждения в главе 4, AUC показывает, насколько хорошо модель отделяет положительные примеры от отрицательных. У данного показателя полезная интерпретация: он описывает вероятность того, что случайно выбранный положительный пример (default) наберет более высокий балл, чем случайно выбранный отрицательный пример (OK). Это важный показатель для проекта: мы хотим, чтобы рискованные клиенты имели более высокую оценку, чем безопасные. Более подробную информацию об AUC можно найти в главе 4.

Как и ранее, мы будем использовать реализацию из Scikit-learn, поэтому импортируем эту библиотеку:

```
from sklearn.metrics import roc_auc_score
```

Сначала мы оценим производительность на обучающем наборе. Поскольку мы выбрали AUC в качестве показателя, нам нужны оценки, а не жесткие прогнозы. Как мы знаем из главы 3, для этого нужно использовать метод `predict_proba`:

```
y_pred = dt.predict_proba(X_train)[:, 1]
roc_auc_score(y_train, y_pred)
```

Когда его выполнение будет завершено, мы видим, что результат равен 100 % — он идеальный. Означает ли это, что мы можем предсказывать дефолт без ошибок? Проведем оценку на проверочных данных, прежде чем делать поспешные выводы:

```
y_pred = dt.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

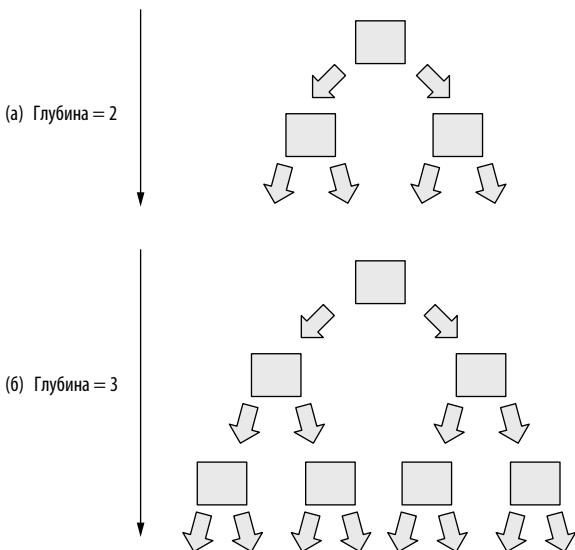
После запуска мы видим, что AUC при проверке составляет всего 65 %.

Мы только что наблюдали случай *переобучения*. Дерево настолько хорошо изучило обучающие данные, что просто запомнило результат для каждого клиента.

Однако когда мы применили его к проверочному набору, модель ошиблась. Правила, которые она извлекла из данных, оказались слишком специфичными для обучающего набора, поэтому она плохо работала применительно к клиентам, которых не встречала во время обучения. В таких случаях мы говорим, что модель не может быть *обобщена*.

Переобучение происходит, когда у нас есть сложная модель, мощность которой позволяет ей попросту запомнить все обучающие данные. Если мы упростим модель, то сможем сделать ее менее мощной и улучшить ее способность к обобщению.

У нас есть несколько способов, позволяющих контролировать сложность дерева. Одним из вариантов является ограничение размера: мы можем указать параметр `max_depth`, который управляет максимальным количеством уровней. Чем больше уровней имеет дерево, тем более сложные правила оно может усвоить (рис. 6.9).



**Рис. 6.9.** Дерево с большим количеством уровней позволяет изучать более сложные правила. Дерево с двумя уровнями менее сложно, чем с тремя, и, следовательно, менее подвержено переобучению

Значение по умолчанию для параметра `max_depth` составляет `None`, что означает, что дерево может вырасти настолько большим, насколько это возможно. Мы можем попробовать меньшее значение и сравнить результаты.

Например, можно изменить его на 2:

```
dt = DecisionTreeClassifier(max_depth=2)
dt.fit(X_train, y_train)
```

Визуализировать полученное дерево можно с помощью функции `export_text` из пакета `tree`:

```
from sklearn.tree import export_text

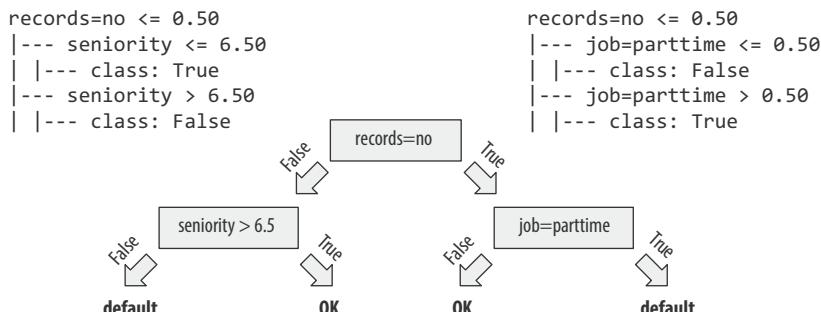
tree_text = export_text(dt, feature_names=dv.feature_names_)
print(tree_text)
```

Нам нужно лишь указать имена признаков, используя `feature_names`. Мы можем получить его из `DictVectorizer`. Вывод показывает следующее:

```
|--- records=no <= 0.50
|   |--- seniority <= 6.50
|   |   |--- class: True
|   |--- seniority > 6.50
|   |   |--- class: False
--- records=no > 0.50
|--- job=parttime <= 0.50
|   |--- class: False
|--- job=parttime > 0.50
|   |--- class: True
```

Каждая строка в выходных данных соответствует узлу с условием. Если условие верно, то мы идем внутрь и повторяем процесс до тех пор, пока не придем к окончательному решению. В конце, если класс имеет значение `True`, то выносится вердикт `default`, а в противном случае — `OK`.

Условие `records=no > 0.50` означает, что у клиента нет записей. Вспомним, что мы используем прямое кодирование для представления `records` с помощью двух признаков: `records=yes` и `records=no`. Для клиента, у которого нет записей, значение `records=no` равно 1, а значение `records=yes` равно 0. Таким образом, `records=no > 0.50` является истинным, когда значение для `records` равно `no` (рис. 6.10).



**Рис. 6.10.** Дерево, которое мы обучили с `max_depth`, установленным в 2

Проверим оценку:

```
y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train auc', auc)

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('validation auc', auc)
```

Мы видим, что оценка после обучения снизилась:

```
train auc: 0.705
val auc: 0.669
```

Раньше производительность на обучающем наборе составляла 100 %, но сейчас данный показатель равен всего 70,5 %. Это означает, что модель больше не может запоминать все результаты из обучающего набора.

Однако оценка по проверочному набору более хорошая: она составляет 66,9 %, что является улучшением по сравнению с предыдущим результатом (65 %). Сделав нашу модель менее сложной, мы улучшили ее способность к обобщению. Теперь она лучше прогнозирует результаты для клиентов, которых раньше не встречала.

Тем не менее у этого дерева возникает новая проблема — оно слишком простое. Чтобы улучшить ситуацию, нам необходимо настроить модель: попробуем разные параметры и посмотрим, какие из них приводят к наилучшему AUC. Чтобы понять, что означают разные параметры и как они влияют на модель, сделаем шаг назад и посмотрим, каким образом деревья решений извлекают правила из данных.

## 6.2.2. Алгоритм обучения дерева решений

Чтобы понять, как дерево решений извлекает закономерности из данных, упростим задачу. Во-первых, мы будем использовать гораздо меньший набор данных с единственным признаком: `assets` (рис. 6.11).

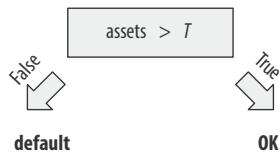
	assets	status
0	8000	default
1	2000	OK
2	0	OK
3	6000	OK
4	6000	default
5	9000	default

**Рис. 6.11.** Меньший набор данных с единственным признаком: `assets`.

Целевая переменная — `status`

Во-вторых, мы вырастим очень маленькое дерево с единственным узлом.

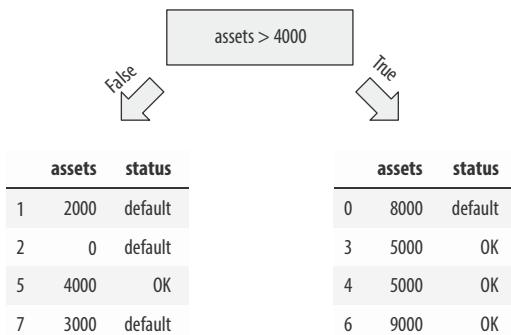
Единственный признак, который есть в нашем наборе данных, — это **assets**. Вот почему условием в узле будет **assets > T**, где  $T$  — пороговое значение, которое нам нужно определить. Если условие истинно, то мы прогнозируем OK, а если ложно, то нашим прогнозом будет default (рис. 6.12).



**Рис. 6.12.** Простое дерево решений с единственным узлом. Узел содержит условие  $\text{assets} > T$ . Нам нужно найти наилучшее значение для  $T$

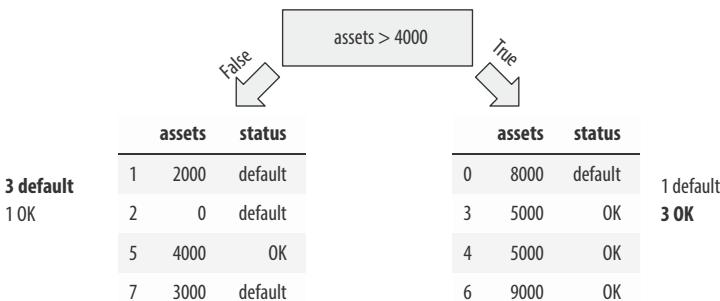
Условие  $\text{assets} > T$  называется *разделением*. Оно разбивает набор данных на две группы: точки данных, которые удовлетворяют условию, и точки данных, которые этого не делают.

Если  $T$  равно 4000, то у нас появятся клиенты с активами более 4000 долларов США (справа) и клиенты с активами менее 4000 долларов США (слева) (рис. 6.13).



**Рис. 6.13.** Условие в узле разбивает набор данных на две части: точки данных, которые удовлетворяют условию (справа), и точки данных, которые ему не удовлетворяют (слева)

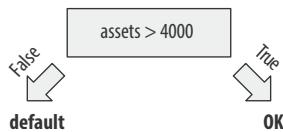
Теперь мы превращаем эти группы в *листья* — узлы принятия решений — взяв наиболее частый статус в каждой группе и использовав его в качестве окончательного решения. В нашем примере default является наиболее частым результатом в левой группе, а OK — в правой (рис. 6.14).



**Рис. 6.14.** Наиболее частым результатом слева является default.

Для группы справа это OK

Таким образом, если активы клиента превышают 4000 долларов, то наше решение — OK, а в противном случае — default assets > 4000 (рис. 6.15).



**Рис. 6.15.** Взяв наиболее частый результат в каждой группе и распределив его по листьям, мы получаем окончательное дерево решений

## Примеси

Эти группы должны быть как можно более однородными. В идеале каждая группа должна содержать только наблюдения одного класса. В таком случае мы называем эти группы *чистыми*.

Например, если у нас есть группа из четырех клиентов с результатами [default, default, default, default], то она чистая, так как содержит лишь клиентов, которые допустили дефолт. Но группа [default, default, default, OK] имеет примеси: есть один клиент, который не допустил дефолт.

При обучении модели дерева решений мы хотим найти такое  $T$ , чтобы *примеси* в обеих группах были минимальными.

Итак, алгоритм нахождения  $T$  довольно прост:

- попробуйте все возможные значения  $T$ ;
- для каждого  $T$  разделите набор данных на левую и правую группы и измерьте их примеси;
- выберите  $T$ , которое дает наименьшую степень примеси.

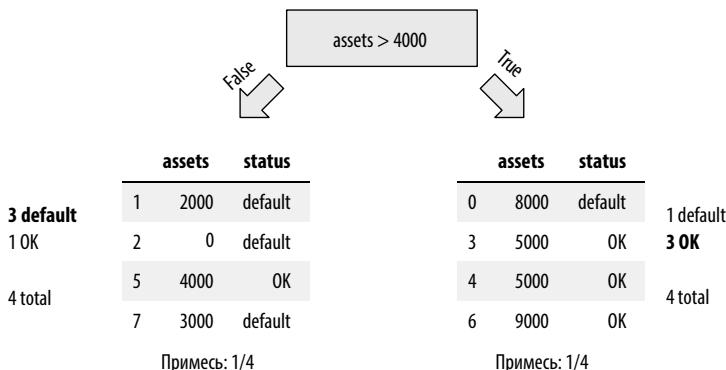
Для измерения примесей можно использовать различные критерии. Самый простой для понимания — это *коэффициент ошибок классификации*, который показывает, сколько наблюдений в группе не принадлежат к классу большинства.

### ПРИМЕЧАНИЕ

Scikit-learn использует более продвинутые критерии разделения, такие как энтропия и примесь Джини. Мы не рассматриваем их в этой книге, но идея та же: они измеряют степень примеси в разделении.

Рассчитаем коэффициент ошибок классификации для разделения  $T = 4000$  (рис. 6.16):

- для левой группы основным классом служит default. Всего есть четыре точки данных, и одна из них не относится к default. Коэффициент ошибок классификации составляет 25 % (1/4);
- для правой группы OK — это класс большинства, и среди них есть один default. Таким образом, коэффициент ошибок классификации тоже составляет 25 % (1/4);
- чтобы рассчитать общую примесь разделения, мы можем взять среднее значение по обеим группам. В этом случае средний показатель составляет 25 %.



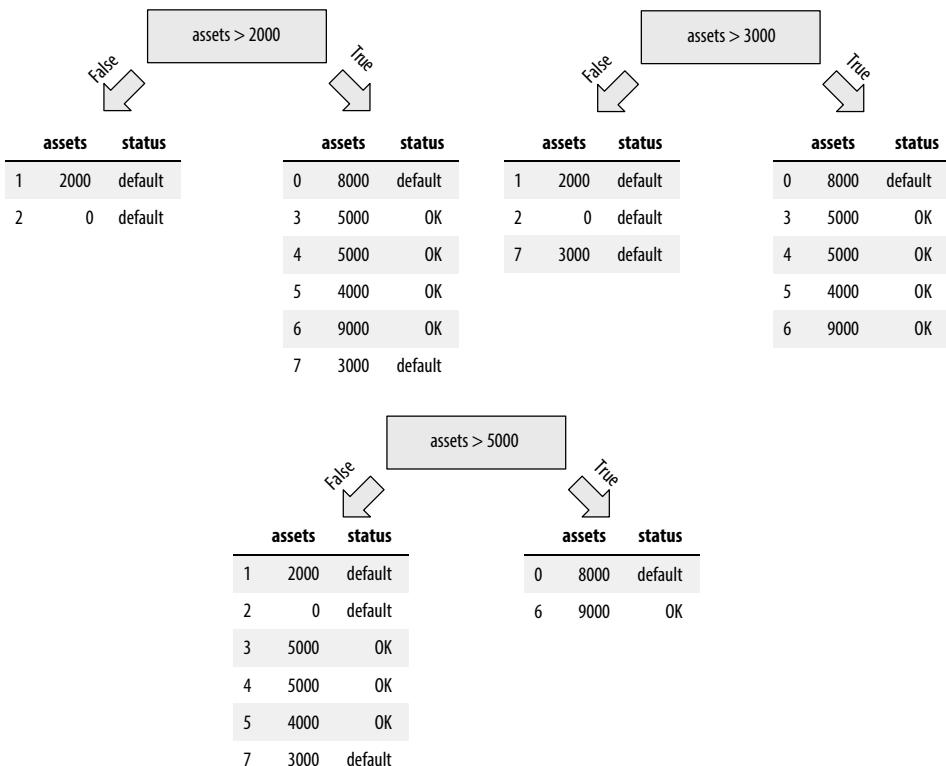
**Рис. 6.16.** Для  $\text{assets} > 4000$  коэффициент ошибок классификации для обеих групп составляет одну четверть

### ПРИМЕЧАНИЕ

В действительности вместо простого среднего значения по обеим группам мы берем средневзвешенное — взвешиваем каждую группу пропорционально ее размеру. Чтобы упростить вычисления, в этой главе мы используем простое среднее значение.

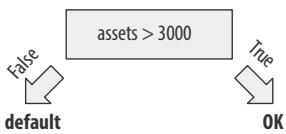
$T = 4000$  не единственное возможное разделение `assets`. Попробуем другие значения  $T$ , такие как 2000, 3000 и 5000 (рис. 6.17):

- для  $T = 2000$  мы имеем 0 % примесей слева ( $0/2$ , все default) и 33,3 % примесей справа ( $2/6$ , 2 из 6 – default, остальные OK). Средний показатель составляет 16,6 %;
- для  $T = 3000$ , 0 % слева и 20 % ( $1/5$ ) справа. Средний показатель составляет 10 %;
- для  $T = 5000$ , 50 % ( $3/6$ ) слева и 50 % ( $1/2$ ) справа. Средний показатель составляет 50 %.



**Рис. 6.17.** В дополнение к `assets > 4000` мы можем попробовать другие значения  $T$ , такие как 2000, 3000 и 5000

Наилучшая средняя примесь составляет 10 % при  $T = 3000$ : мы получили ноль ошибок для левого дерева и только одну (из пяти строк) для правого. Таким образом, нам следует выбрать 3000 в качестве порога для нашей окончательной модели (рис. 6.18).



**Рис. 6.18.** Наилучшее разделение для этого набора данных — assets > 3000

### Выбор наилучшего признака для разделения

Теперь немного усложним задачу и добавим в набор данных еще один признак: **debt** (рис. 6.19).

	assets	debt	status
0	8000	3000	default
1	2000	1000	default
2	0	1000	default
3	5000	1000	OK
4	5000	1000	OK
5	4000	1000	OK
6	9000	500	OK
7	3000	2000	default

**Рис. 6.19.** Набор данных с двумя признаками: assets и debt.

Целевая переменная — status

Раньше у нас был только один признак: **assets**. Мы точно знали, что именно с его помощью будем разделять данные. Теперь у нас их два, поэтому помимо выбора наилучшего порога для разделения нам нужно выяснить, какой признак использовать.

Решение довольно простое: мы перебираем все признаки и для каждого выбираем наилучший порог.

Изменим алгоритм обучения, чтобы включить в него это изменение:

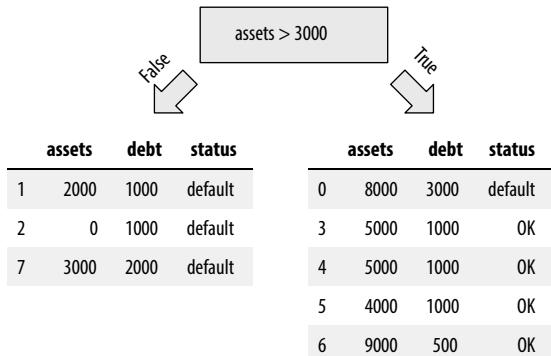
- для каждого признака попробовать все возможные пороговые значения;
- для каждого порогового значения  $T$  измерить примесь разделения;
- выбрать признак и пороговое значение с минимально возможной примесью.

Применим этот алгоритм к нашему набору данных:

- мы уже определили, что для **assets** наилучшее значение  $T$  составляет 3000. Средняя примесь этого разделения 10 %;

## 246 Глава 6. Деревья решений и ансамблевое обучение

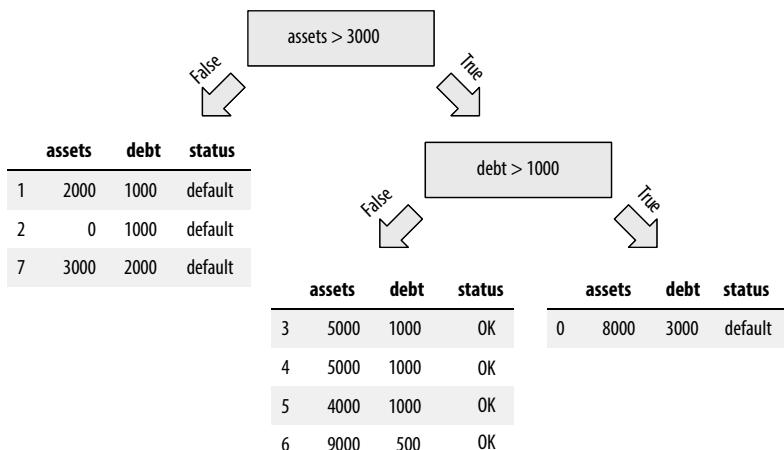
- для `debt` лучшее значение  $T$  равно 1000. В этом случае средняя примесь составляет 17 %. Таким образом, наилучшее разделение — это `asset > 3000` (рис. 6.20).



**Рис. 6.20.** Наилучшее разделение — `asset > 3000`, средняя примесь которого составляет 10 %

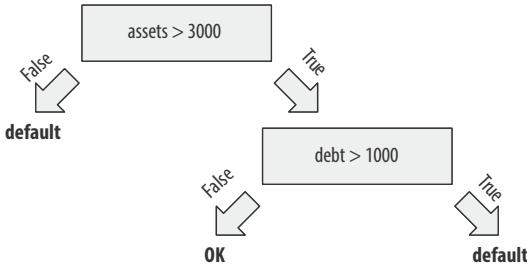
Группа слева уже чистая, а группа справа — пока нет. Мы можем сделать ее менее загрязненной, повторив процесс: разделим ее еще раз!

Применяя тот же алгоритм к набору данных справа, мы обнаруживаем, что наилучшим условием разделения становится `debt > 1000`. Теперь у нас два уровня дерева, или можно сказать, что глубина этого дерева равна 2 (рис. 6.21).



**Рис. 6.21.** Рекурсивно повторив алгоритм для группы справа, мы получаем дерево с двумя уровнями

Прежде чем дерево решений будет готово, предпримем последний шаг: преобразуем группы в узлы принятия решений. Для этого возьмем наиболее частый статус в каждой группе. Таким образом, мы получаем дерево решений (рис. 6.22).



**Рис. 6.22.** Группы уже являются чистыми, поэтому наиболее частый статус — тот единственный статус, который есть у каждой группы.

Мы принимаем его как окончательное решение в каждом листе

### Критерий остановки

При обучении дерева решений мы можем продолжать разбивать данные до тех пор, пока все группы не станут чистыми. Именно это и происходит, когда мы не накладываем никаких ограничений на деревья в Scikit-learn. Как мы уже видели, итоговая модель становится слишком сложной, что приводит к переобучению.

Мы решили эту проблему с помощью параметра `max_depth`, ограничив размер дерева и не позволив ему вырасти слишком большим.

Чтобы решить, хотим ли мы продолжить разделение данных, используем *критерии остановки* — критерии, которые указывают, следует ли нам добавить еще одно разделение к дереву или пора остановиться.

Наиболее распространенными критериями остановки являются следующие:

- группа уже чиста;
- дерево достигло предела глубины (контролируется `max_depth`);
- группа слишком мала, чтобы продолжать разделение (контролируется параметром `min_samples_leaf`).

Применяя эти критерии для более ранней остановки, мы упрощаем нашу модель и, следовательно, снижаем риск переобучения.

Используем эту информацию для корректировки алгоритма обучения:

- поиск лучшего разделения:
  - для каждого признака попробуйте все возможные пороговые значения;
  - используйте тот, при котором меньше всего примесей;

- если достигнута максимально допустимая глубина, то остановитесь;
- если группа слева достаточно велика и еще не чиста, то повторите для левой;
- если группа справа достаточно велика и еще не чиста, то повторите для правой.

Несмотря на то что это упрощенная версия алгоритма обучения дерева решений, она может дать достаточно информации о внутренних особенностях процесса обучения.

Самое главное, что мы теперь знаем два параметра, определяющие сложность модели. Изменяя их, мы можем улучшить производительность модели.

### Упражнение 6.1

У нас есть набор данных с десятью признаками, и нам нужно добавить в него еще один. Что произойдет со скоростью обучения?

- A. При наличии еще одного признака обучение займет больше времени.  
 Б. Количество признаков не влияет на скорость обучения.

### 6.2.3. Настройка параметров для дерева решений

Процесс поиска наилучшего набора параметров называется *настройкой параметров*. Обычно мы проводим ее, изменяя модель и оценивая ее на проверочном наборе данных. В итоге мы оставляем модель, имеющую наилучшую оценку при проверке.

Как мы только что узнали, мы можем настраивать два параметра:

- `max_depth`;
- `min_leaf_size`.

Они наиболее важны, поэтому скорректируем только их. О других параметрах вы можете узнать в официальной документации (<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>).

Обучая нашу модель ранее, мы ограничили глубину дерева двумя уровнями, но не трогали `min_leaf_size`. Таким образом, мы получили AUC 66 % на проверочном наборе.

Поищем наилучшие параметры.

Начнем с настройки `max_depth`. Для этого мы переберем несколько разумных значений и попытаемся понять, какое из них работает лучше всего:

```
for depth in [1, 2, 3, 4, 5, 6, 10, 15, 20, None]:
    dt = DecisionTreeClassifier(max_depth=depth)

    dt.fit(X_train, y_train)
    y_pred = dt.predict_proba(X_val)[:, 1]
    auc = roc_auc_score(y_val, y_pred)
    print('%4s -> %.3f' % (depth, auc))
```

Значение `None` говорит о том, что ограничений на глубину нет, поэтому дерево будет расти настолько большим, насколько это вообще возможно.

Запустив этот код, мы видим, что `max_depth`, равный 5, дает наилучший AUC (76,6 %), а за ним следуют 4 и 6 (рис. 6.23).

```
1 -> 0.606
2 -> 0.669
3 -> 0.739
4 -> 0.761
5 -> 0.766
6 -> 0.754
10 -> 0.685
15 -> 0.671
20 -> 0.657
None -> 0.657
```

] Оптимальные значения  
для `max_depth`

**Рис. 6.23.** Оптимальное значение глубины — 5 (76,6 %),  
за ним следуют 4 (76,1 %) и 6 (75,4 %)

Далее настроим `min_leaf_size`. Для этого переберем три наилучших параметра `max_depth` и для каждого используем разные значения `min_leaf_size`:

```
for m in [4, 5, 6]:
    print('depth: %s' % m)

    for s in [1, 5, 10, 15, 20, 50, 100, 200]:
        dt = DecisionTreeClassifier(max_depth=m, min_samples_leaf=s)
        dt.fit(X_train, y_train)
        y_pred = dt.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)
        print('%s -> %.3f' % (s, auc))

print()
```

После запуска мы видим, что наилучший AUC составляет 78,5 % с параметрами `min_sample_leaf=15` и `max_depth=6` (табл. 6.1).

**Таблица 6.1.** AUC по проверочным данным для различных значений `min_leaf_size` (строки) и `max_depth` (столбцы)

	<code>depth=4</code>	<code>depth=5</code>	<code>depth=6</code>
<b>1</b>	0,761	0,766	0,754
	0,761	0,768	0,760
	0,761	0,762	0,778
	0,764	0,772	<b>0,785</b>
	0,761	0,774	0,774
	0,753	0,768	0,770
	0,756	0,763	0,776
	0,747	0,759	0,768

#### ПРИМЕЧАНИЕ

Как видите, значение, которое мы используем для `min_leaf_size`, влияет на наилучшее значение `max_depth`. Вы можете поэкспериментировать с более широким диапазоном значений `max_depth`, чтобы еще больше повысить производительность.

Мы обнаружили наши наилучшие параметры, поэтому используем их для обучения окончательной модели:

```
dt = DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
dt.fit(X_train, y_train)
```

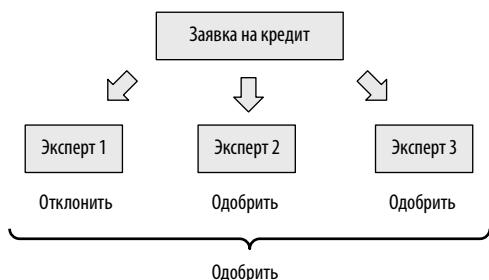
Деревья решений — простые и эффективные модели, но они становятся еще более действенными, когда мы объединяем множество деревьев. Далее мы выясним, как это можно сделать, добившись еще более высокой производительности прогнозирования.

## 6.3. СЛУЧАЙНЫЙ ЛЕС

На мгновение предположим, что у нас нет алгоритма машинного обучения, который может помочь нам оценить кредитный риск. Вместо этого есть группа экспертов.

Каждый эксперт может самостоятельно решить, следует ли одобрить заявку на получение кредита или отклонить. Отдельный эксперт может допустить ошибку. Однако менее вероятно, что все эксперты решат одобрить заявку, а клиент не вернет деньги.

Таким образом, мы можем опросить всех экспертов независимо, а затем объединить их вердикты в окончательное решение, например используя большинство голосов (рис. 6.24).



**Рис. 6.24.** Группа экспертов может принять лучшее решение, нежели один из них

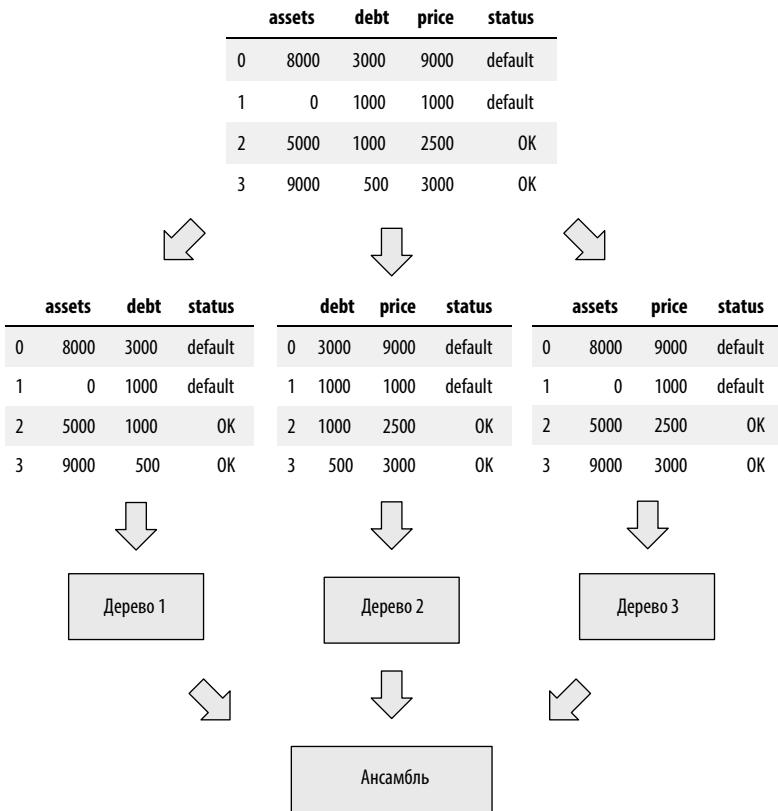
Эта идея применима и к машинному обучению. Одна модель может подвести, но если мы объединим результаты нескольких моделей в одну, то вероятность неправильного ответа будет меньше. Эта концепция называется *ансамблевым обучением*, а комбинация моделей — *ансамблем*.

Чтобы все получилось, модели должны быть разными. Если мы обучим одну и ту же модель дерева решений десять раз, то она будет прогнозировать один и тот же результат, что сделает всю работу совершенно бессмысленной.

Самый простой способ получить разные модели — обучить каждое дерево по различающемуся подмножеству признаков. Например, предположим, что у нас есть три признака: `assets`, `debts` и `price`. Мы можем обучить три модели:

- первая будет использовать `assets` и `debts`;
- вторая — `debts` и `price`;
- последняя — `assets` и `price`.

При таком подходе у нас окажутся разные деревья, каждое из которых принимает собственные решения (рис. 6.25). Но когда мы объединим их прогнозы, их ошибки усредняются, и в совокупности мы получим большую прогностическую эффективность.



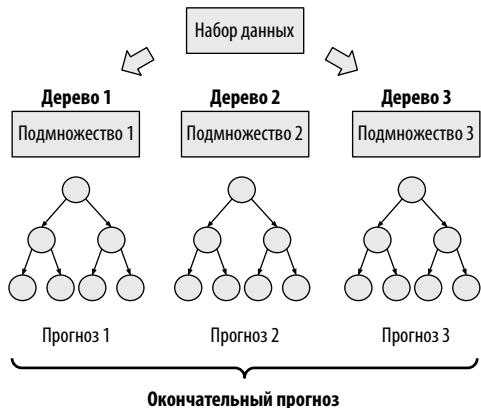
**Рис. 6.25.** Модели, которые мы хотим объединить в ансамбль, не должны быть одинаковыми. Мы можем убедиться, что они отличаются друг от друга, обучив каждое дерево по разному подмножеству признаков

Этот способ объединения нескольких деревьев решений в ансамбль называется *случайным лесом*. Чтобы обучить случайный лес, можно сделать следующее (рис. 6.26):

- обучить  $N$  независимых моделей дерева решений;
- для каждой модели выбрать случайное подмножество признаков и использовать для обучения только их;
- при прогнозировании объединить выходные данные  $N$  моделей в одно.

#### ПРИМЕЧАНИЕ

Здесь приведена крайне упрощенная версия алгоритма. Этого достаточно, чтобы проиллюстрировать основную идею, но в реальности все гораздо сложнее.



**Рис. 6.26.** Обучение модели случайного леса: для обучения каждого дерева случайным образом выберите подмножество признаков. При составлении окончательного прогноза объедините все прогнозы в один

Scikit-learn содержит реализацию случайного леса, поэтому мы можем использовать ее для решения нашей задачи. Так и поступим.

### 6.3.1. Обучение случайного леса

Чтобы использовать случайный лес в Scikit-learn, нам нужно импортировать `RandomForestClassifier` из пакета `ensemble`:

```
from sklearn.ensemble import RandomForestClassifier
```

Первое, что нужно указать при обучении модели, — количество деревьев, которые мы хотим получить в ансамбле. Мы делаем это с помощью параметра `n_estimators`:

```
rf = RandomForestClassifier(n_estimators=10)
rf.fit(X_train, y_train)
```

Когда обучение закончится, мы можем оценить результат:

```
y_pred = rf.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

Он показывает 77,9 %. Однако число, которое вы видите, может оказаться другим. Каждый раз, когда мы переучиваем модель, оценка меняется: она варьируется от 77 до 80 %.

Причиной этого является рандомизация: для обучения дерева мы случайным образом выбираем подмножество объектов. Чтобы результаты были

согласованными, нам нужно зафиксировать начальное значение для генератора случайных чисел, присвоив некое значение параметру `random_state`:

```
rf = RandomForestClassifier(n_estimators=10, random_state=3)
rf.fit(X_train, y_train)
```

Теперь мы можем получить оценку:

```
y_pred = rf.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

На этот раз показатель AUC равен 78 %. Теперь эта оценка не будет меняться, вне зависимости от количества обучений.

Количество деревьев в ансамбле — важный параметр, влияющий на производительность модели. Обычно модель с большим количеством деревьев работает лучше, чем с меньшим. С другой стороны, добавление слишком большого количества деревьев не всегда полезно.

Чтобы понять, сколько деревьев нам требуется, мы можем перебрать различные значения для `n_estimators` и оценить его влияние на AUC:

```
aucs = [] ← Создает список с результатами AUC
for i in range(10, 201, 10):
    rf = RandomForestClassifier(n_estimators=i, random_state=3) | Обучает все больше
    rf.fit(X_train, y_train) | деревьев на каждой
    y_pred = rf.predict_proba(X_val)[:, 1] | итерации
    auc = roc_auc_score(y_val, y_pred) | Считает
    print('%s -> %.3f' % (i, auc)) | оценку
    aucs.append(auc) ← | Добавляет результат в список
                        | с другими результатами
```

В этом коде мы пробуем разное количество деревьев: от 10 до 200 с шагом 10 (10, 20, 30, ...). Каждый раз при обучении модели мы вычисляем ее показатель AUC по проверочному набору и записываем его.

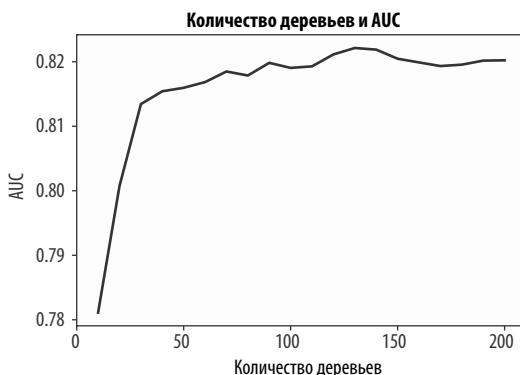
После окончания сможем отобразить результаты:

```
plt.plot(range(10, 201, 10), aucs)
```

На рис. 6.27 можно увидеть результаты.

Производительность быстро растет для первых 25–30 деревьев; затем рост замедляется. После 130 добавление большего количества деревьев больше не влияет на производительность, и она остается примерно на уровне 82 %.

Количество деревьев — не единственный параметр, который мы можем менять, чтобы повысить производительность. Далее мы узнаем, какие еще параметры можно настроить, чтобы улучшить модель.



**Рис. 6.27.** Производительность модели случайного леса с различными значениями n\_estimators

### 6.3.2. Настройка параметров для случайного леса

Ансамбль случайного леса состоит из нескольких деревьев решений, поэтому наиболее важные параметры, которые нам нужно настроить для случайного леса, одинаковы:

- max\_depth;
- min\_leaf\_size.

Мы можем менять и другие параметры, но не будем подробно рассматривать их в этой главе. Получить дополнительную информацию можно в официальной документации (<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>).

Начнем с `max_depth`. Мы уже выяснили, что данный параметр существенно влияет на производительность дерева решений. Это также относится и к случайному лесу: большие деревья, как правило, склонны к переобучению больше, чем деревья поменьше.

Протестируем несколько значений для `max_depth` и посмотрим, как меняется AUC по мере роста количества деревьев:

```
all_aucs = {} ← Создает словарь с результатами AUC
for depth in [5, 10, 20]: ← Выполняет итерации по различным
    print('depth: %s' % depth) | значениям глубины
    aucs = [] ← Создает список с результатами AUC
    for i in range(10, 201, 10): ← для текущего уровня глубины
        rf = RandomForestClassifier(n_estimators=i, max_depth=depth,
```

```

random_state=1)           ← Выполняет итерацию по различным
rf.fit(X_train, y_train)   | значениям n_estimators
y_pred = rf.predict_proba(X_val)[:, 1] |
auc = roc_auc_score(y_val, y_pred) | Оценивает модель
print('%s -> %.3f' % (i, auc))
aucs.append(auc)

all_aucs[depth] = aucs      ← Сохраняет показатели AUC для текущего
print()                     | уровня глубины в словаре

```

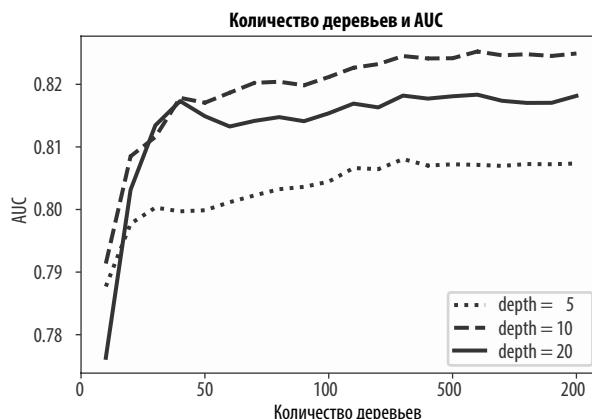
Теперь для каждого значения `max_depth` у нас имеется ряд оценок AUC. Мы можем сразу же нанести их на график:

```

num_trees = list(range(10, 201, 10))
plt.plot(num_trees, all_aucs[5], label='depth=5')
plt.plot(num_trees, all_aucs[10], label='depth=10')
plt.plot(num_trees, all_aucs[20], label='depth=20')
plt.legend()

```

На рис. 6.28 мы видим результат.



**Рис. 6.28.** Производительность случайного леса с различными значениями `max_depth`

При `max_depth=10` AUC превышает 82 %, тогда как при других значениях показатель работает хуже.

Теперь настроим `min_samples_leaf`. Мы устанавливаем значение для `max_depth` из предыдущего шага, а затем следуем тому же подходу, чтобы определить наилучшее значение для `min_samples_leaf`:

```

all_aucs = {}

for m in [3, 5, 10]:
    print('min_samples_leaf: %s' % m)

```

```
aucs = []

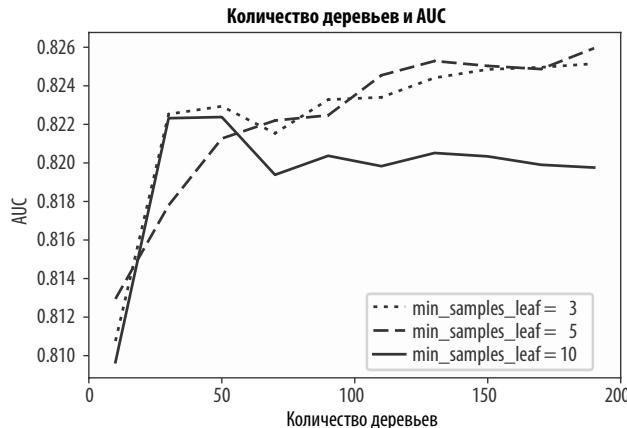
for i in range(10, 201, 20):
    rf = RandomForestClassifier(n_estimators=i, max_depth=10,
    min_samples_leaf=m, random_state=1)
    rf.fit(X_train, y_train)
    y_pred = rf.predict_proba(X_val)[:, 1]
    auc = roc_auc_score(y_val, y_pred)
    print('%s -> %.3f' % (i, auc))
    aucs.append(auc)

all_aucs[m] = aucs
print()
```

Выведем график:

```
num_trees = list(range(10, 201, 20))
plt.plot(num_trees, all_aucs[3], label='min_samples_leaf=3')
plt.plot(num_trees, all_aucs[5], label='min_samples_leaf=5')
plt.plot(num_trees, all_aucs[10], label='min_samples_leaf=10')
plt.legend()
```

После чего обратимся к результатам (рис. 6.29).



**Рис. 6.29.** Производительность случайного леса с различными значениями `min_samples_leaf` (с `max_depth=10`)

Мы видим, что AUC выглядит немного лучше для небольших значений `min_samples_leaf`, а наилучшее значение равно 5.

Таким образом, наилучшими параметрами для случайного леса для нашей задачи являются:

- `max_depth=10`;
- `min_samples_leaf=5`.

Мы достигли наилучшего AUC с 200 деревьями, поэтому следует установить параметр `n_estimators` равным 200.

Обучим окончательную модель:

```
rf = RandomForestClassifier(n_estimators=200, max_depth=10,  
                           min_samples_leaf=5, random_state=1)
```

Случайный лес — не единственный способ объединить несколько деревьев решений. Существует и другой подход: градиентный бустинг. Мы рассмотрим его далее.

### Упражнение 6.2

Чтобы ансамбль оказался полезным, деревья в случайном лесу должны отличаться друг от друга. Какое действие поможет достичь этого?

- А. Выбор различных параметров для каждого отдельного дерева.
- Б. Случайный выбор различающегося подмножества объектов для каждого дерева.
- В. Случайный выбор значений для разделения.

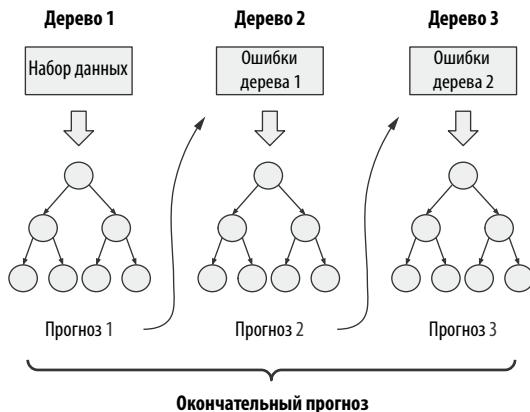
## 6.4. ГРАДИЕНТНЫЙ БУСТИНГ

В случайном лесу каждое дерево независимо: оно обучено на своем наборе признаков. После того как отдельные деревья обучены, мы объединяем все их решения, чтобы получить окончательное.

Однако это не единственный способ объединить несколько моделей в одном ансамбле. В качестве альтернативы мы можем обучать модели последовательно — каждая следующая модель пытается исправить ошибки предыдущей:

- обучить первую модель;
- посмотреть на ошибки, которые она допускает;
- обучить еще одну модель, которая исправляет эти ошибки;
- посмотреть на ошибки еще раз; повторять снова и снова.

Такой способ объединения моделей называется *бустингом*. *Градиентный бустинг* — особая разновидность этого подхода, которая особенно хорошо работает с деревьями (рис. 6.30).



**Рис. 6.30.** При градиентном бустинге мы обучаем модели последовательно, и каждое следующее дерево исправляет ошибки предыдущего

Разберемся, как использовать градиентный бустинг для решения нашей задачи.

### 6.4.1. XGBoost: Экстремальный градиентный бустинг

Есть множество хороших реализаций модели градиентного бустинга: `GradientBoostingClassifier` из Scikit-learn, XGBoost, LightGBM и CatBoost. В этой главе мы используем XGBoost (сокращение от Extreme Gradient Boosting), который является наиболее популярной реализацией.

XGBoost не поставляется с Anaconda, поэтому для того, чтобы его использовать, нам потребуется выполнить установку. Проще всего установить его с помощью `pip`:

```
pip install xgboost
```

Затем откройте блокнот с нашим проектом и импортируйте XGBoost:

```
import xgboost as xgb
```

#### ПРИМЕЧАНИЕ

В некоторых случаях импорт XGBoost может выдать вам предупреждение типа `YMLLoadWarning`. Вам не стоит беспокоиться об этом; библиотека будет работать.

Использование псевдонима `xgb` при импорте XGBoost — еще одно соглашение, как и в случае с другими популярными пакетами машинного обучения в Python.

Прежде чем мы сможем обучить модель XGBoost, нам придется обернуть наши данные в `DMatrix` — специальную структуру данных для эффективного поиска разбиений. Так и сделаем:

```
dtrain = xgb.DMatrix(X_train, label=y_train, feature_names=dv.feature_names_)
```

При создании экземпляра `DMatrix` мы передаем три параметра:

- `X_train` — матрицу признаков;
- `y_train` — целевую переменную;
- `feature_names` — имена признаков в `X_train`.

Проделаем то же самое для проверочного набора данных:

```
dval = xgb.DMatrix(X_val, label=y_val, feature_names=dv.feature_names_)
```

Следующий шаг — указание параметров для обучения. Мы используем только небольшое подмножество параметров XGBoost по умолчанию (полный список параметров можно найти в официальной документации: <https://xgboost.readthedocs.io/en/latest/parameter.html>):

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

Для нас самым важным параметром сейчас является `objective`: он определяет задачу обучения. Мы решаем задачу двоичной классификации — вот почему нам нужно выбрать `binary:logistic`. Чуть позже мы рассмотрим и остальные параметры.

Для обучения модели XGBoost мы используем функцию `train`. Начнем с десяти деревьев:

```
model = xgb.train(xgb_params, dtrain, num_boost_round=10)
```

Мы предоставляем `train` три аргумента:

- `xgb_params` — параметры для обучения;
- `dtrain` — набор данных для обучения (экземпляр `DMatrix`);
- `num_boost_round=10` — количество деревьев для обучения.

Через несколько секунд мы получаем модель. Чтобы оценить ее, нам нужно сделать прогноз на основе проверочного набора данных. Для этого используем метод `predict` с проверочными данными, обернутыми в `DMatrix`:

```
y_pred = model.predict(dval)
```

Результат, `y_pred`, представляет собой одномерный массив NumPy с прогнозами: оценку риска для каждого клиента в проверочном наборе данных (рис. 6.31).

```
y_pred = model.predict(dval)
y_pred[:10]

array([0.08926772, 0.0468099 , 0.09692743, 0.17261842, 0.05435968,
       0.12576081, 0.08033007, 0.61870354, 0.486538 , 0.04056795],
      dtype=float32)
```

**Рис. 6.31.** Прогнозы XGBoost

Далее мы рассчитываем AUC, используя тот же подход, что и ранее:

```
roc_auc_score(y_val, y_pred)
```

В результате мы получаем 81,5 %. Это довольно хороший итог, но он все еще хуже, чем наша лучшая модель случайного леса (82,5 %).

Обучение модели XGBoost упрощается, когда мы можем видеть, как меняется ее производительность при увеличении количества деревьев. Дальше узнаем, как это сделать.

## 6.4.2. Мониторинг производительности модели

Чтобы получить представление о том, как меняется AUC по мере роста количества деревьев, мы можем использовать список наблюдения — встроенную в XGBoost функцию мониторинга производительности модели.

Список наблюдения — это список Python с кортежами. Каждый кортеж содержит DMatrix и ее имя.

Обычно это делается так:

```
watchlist = [(dtrain, 'train'), (dval, 'val')]
```

Кроме того, мы изменим список параметров для обучения: нам требуется указать метрику, которую мы используем для оценки. В нашем случае это AUC:

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc', ←
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

Устанавливает метрику  
оценки в значение AUC

Чтобы использовать список наблюдения во время обучения, нам нужно указать два дополнительных аргумента для функции `train`:

- `evals` — список наблюдения;
- `verbose_eval` — как часто выводить метрику. Если мы установим для него значение 10, то будем видеть результат после каждого десятого шага.

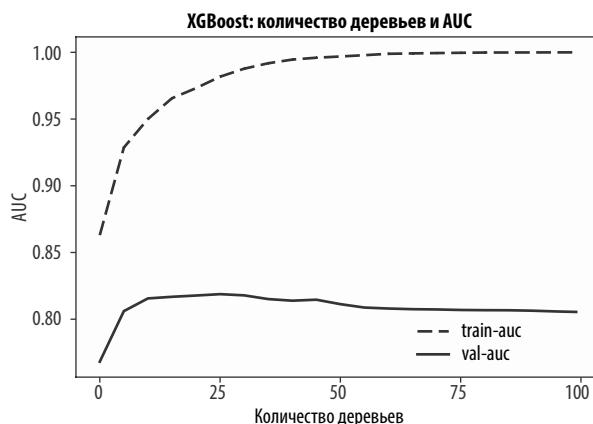
Обучим модель:

```
model = xgb.train(xgb_params, dtrain,
                   num_boost_round=100,
                   evals=watchlist, verbose_eval=10)
```

Во время обучения XGBoost выводит следующие результаты:

```
[0] train-auc:0.862996  val-auc:0.768179
[10] train-auc:0.950021  val-auc:0.815577
[20] train-auc:0.973165  val-auc:0.817748
[30] train-auc:0.987718  val-auc:0.817875
[40] train-auc:0.994562  val-auc:0.813873
[50] train-auc:0.996881  val-auc:0.811282
[60] train-auc:0.998887  val-auc:0.808006
[70] train-auc:0.999439  val-auc:0.807316
[80] train-auc:0.999847  val-auc:0.806771
[90] train-auc:0.999915  val-auc:0.806371
[99] train-auc:0.999975  val-auc:0.805457
```

По мере увеличения количества деревьев увеличивается и оценка на обучающем наборе (рис. 6.32).



**Рис. 6.32.** Влияние количества деревьев на AUC из наборов для обучения и проверки. Чтобы узнать, как отображать эти значения, загляните в блокнот в репозитории GitHub книги

Такое поведение ожидаемо: при повышении каждой следующей модель пытается исправить ошибки предыдущего шага, поэтому оценка каждый раз улучшается.

Однако в случае проверки это не так. Сначала оценка повышается, а затем начинает снижаться. Это эффект переобучения: наша модель становится все более и более сложной, пока попросту не запоминает весь обучающий набор. Прогнозирование для клиентов за пределами обучающего набора становится бесполезным, и оценка при проверке отражает это.

Мы получаем лучший AUC на 30-й итерации (81,7 %), однако он не слишком сильно отличается от результата, полученного на 10-й (81,5 %).

Далее мы выясним, как получить максимальную отдачу от XGBoost, настроив его параметры.

### 6.4.3. Настройка параметров XGBoost

Ранее для обучения модели мы использовали подмножество параметров по умолчанию:

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

Нас больше всего интересуют первые три параметра. Они управляют процессом обучения:

- **eta** — скорость обучения. Деревья решений и случайный лес не имеют этого параметра. Мы рассмотрим его позже в этом разделе, когда будем настраивать;
- **max\_depth** — максимально допустимая глубина каждого дерева; такая же, как **max\_depth** в **DecisionTreeClassifier** из Scikit-learn;
- **min\_child\_weight** — минимальное количество наблюдений в каждой группе; совпадает с **min\_leaf\_size** в **DecisionTreeClassifier** из Scikit-learn.

Ниже представлено описание других параметров:

- **objective** — тип решаемой задачи. Для классификации это **binary:logistic**;
- **eval\_metric** — метрика, которую мы используем для оценки. Для этого проекта это AUC;

- **nthread** — количество потоков, которые мы используем для обучения. XGBoost прекрасно выполняет распараллеливание, поэтому установите параметр согласно количеству ядер вашего компьютера;
- **seed** — начальное значение для генератора случайных чисел; нам нужно установить его, чтобы гарантировать воспроизводимость результатов;
- **silent** — объем вывода. Когда мы устанавливаем его равным 1, выводятся только предупреждения.

Это не полный список параметров, а лишь основные из них. Больше информации обо всех параметрах вы можете найти в официальной документации (<https://xgboost.readthedocs.io/en/latest/parameter.html>).

Мы уже изучили **max\_depth** и **min\_child\_weight** (**min\_leaf\_size**), но еще не сталкивались с **eta** — параметром скорости обучения. Поговорим о нем и посмотрим, как можно его оптимизировать.

### Скорость обучения

При бустинге каждое дерево пытается исправить ошибки предыдущих итераций. Скорость обучения определяет вес такой корректировки. Если задано большое значение для **eta**, то корректировка значительно перевешивает предыдущие прогнозы. С другой стороны, если значение невелико, используется лишь небольшая часть этой корректировки.

На практике это означает следующее:

- если **eta** слишком велик, то модель начинает переобучаться довольно рано, так и не реализовав весь свой потенциал;
- если же он слишком мал, то нам придется обучить слишком много деревьев, прежде чем модель сможет показать достойный результат.

Значение по умолчанию 0,3 подойдет для больших наборов данных, но для небольших, подобных нашему, стоит попробовать меньшие значения, такие как 0,1 или даже 0,05.

Так и поступим, а затем посмотрим, помогает ли это улучшить производительность:

```
xgb_params = {
    'eta': 0.1,           ← Изменяет eta
    'max_depth': 6,      | с от 0.3 до 0.1
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

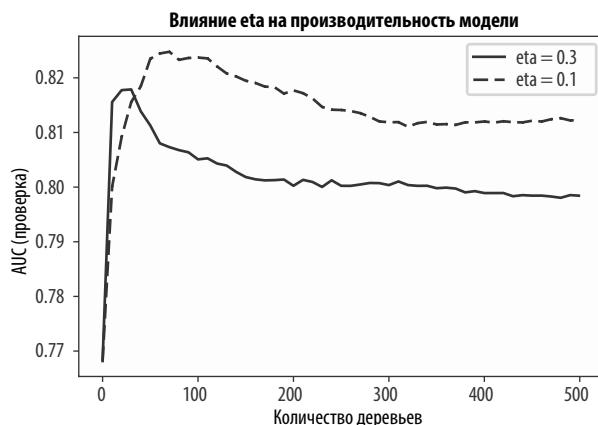
Поскольку теперь есть возможность использовать список наблюдения для мониторинга производительности нашей модели, мы можем обучаться столько итераций, сколько захотим. Ранее мы использовали 100 итераций, но для меньшего `eta` этого может оказаться недостаточно. Так что проведем 500 циклов обучения:

```
model = xgb.train(xgb_params, dtrain,
                   num_boost_round=500, verbose_eval=10,
                   evals=watchlist)
```

При запуске мы видим, что лучший результат проверки составляет 82,4 %:

```
[60] train-auc:0.976407 val-auc:0.824456
```

Ранее нам удавалось достичь AUC 81,7 %, когда `eta` было присвоено значение по умолчанию 0,3. Сравним эти две модели (рис. 6.33).



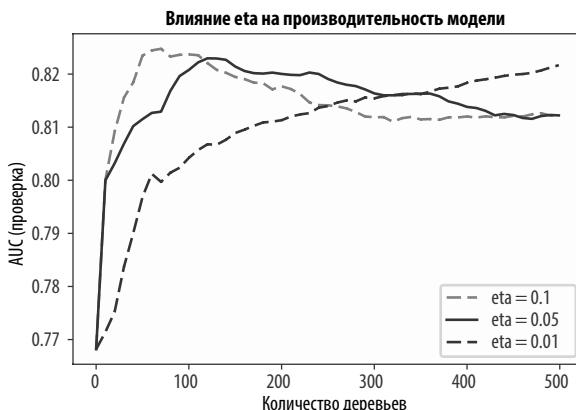
**Рис. 6.33.** Влияние параметра `eta` на оценку при проверке

Когда `eta` составляет 0,3, мы довольно быстро получаем наилучший AUC, но затем начинается переобучение. После 30-й итерации производительность на проверочном наборе падает.

Когда `eta` составляет 0,1, AUC растет медленнее, но достигает максимума при более высоком значении. При меньшей скорости обучения достижение пика требует большего количества деревьев, но зато можно добиться лучшей производительности.

Для сравнения мы также можем попробовать другие значения `eta` (рис. 6.34):

- для 0,05 наилучший AUC составляет 82,2 % (после 120 итераций);
- для 0,01 наилучший AUC составляет 82,1 % (после 500 итераций).



**Рис. 6.34.** Модель требует большего количества деревьев, если параметр eta небольшой

Когда `eta` равен 0,05, производительность аналогична 0,1, но для достижения пика требуется на 60 итераций больше.

Для `eta` 0,01 она растет слишком медленно и даже после 500 итераций не достигает пика. Если бы мы провели еще больше итераций, то она потенциально могла бы достичь того же уровня AUC, что и при других значениях. Но даже если все так, это непрактично: с точки зрения вычислений будет дорого оценивать все эти деревья во время прогнозирования.

Таким образом, мы используем для `eta` значение 0,1. Далее настроим другие параметры.

### Упражнение 6.3

У нас есть модель градиентного бустинга с `eta=0.1`. Достижение максимальной производительности требует 60 деревьев. Что произойдет, если мы увеличим `eta` до 0,5?

- А. Количество деревьев не изменится.
- Б. Достижение максимальной производительности модели потребует большего количества деревьев.
- В. Достижение максимальной производительности модели потребует меньшего количества деревьев.

### Настройка других параметров

Следующий параметр, который мы настраиваем, — `max_depth`. Значение по умолчанию равно 6, поэтому мы можем попробовать:

- меньшее значение, например 3;
- большее значение, например 10.

Результат должен дать представление о том, находится ли наилучшее значение для `max_depth` между 3 и 6 или между 6 и 10.

Сначала проверим значение 3:

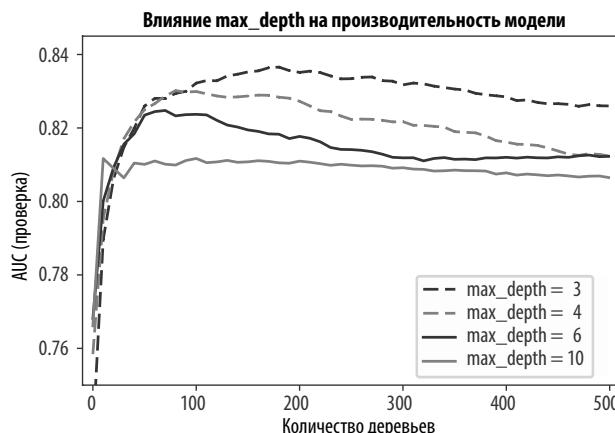
```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,           ← Изменяет максимальную
    'min_child_weight': 1,     глубину с 6 на 3

    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

Лучший AUC, который мы получаем с его помощью, составляет 83,6 %.

Далее попробуем значение 10. В этом случае наилучшее значение составит 81,1 %.

Это означает, что оптимальный параметр `max_depth` должен находиться в диапазоне от 3 до 6. Однако, пробуя 4, мы видим, что наилучший AUC составляет 83 %, что немного хуже, чем AUC, который мы получили при глубине 3 (рис. 6.35).

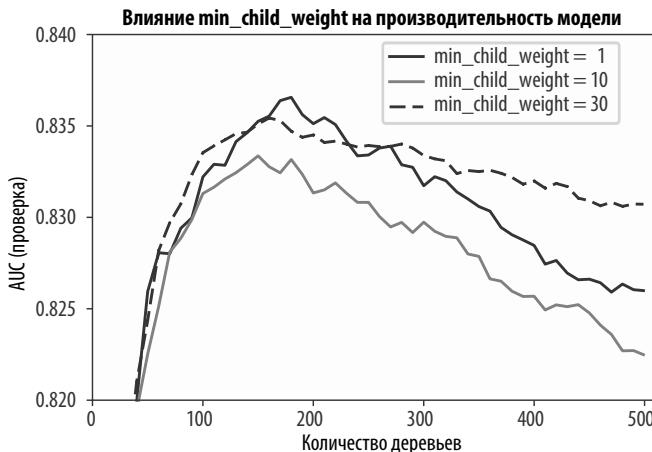


**Рис. 6.35.** Оптимальное значение `max_depth` равно 4: с его помощью мы можем достичь AUC 83,6 %

Следующий параметр, который мы настроим, — это `min_child_weight`. Это то же самое, что `min_leaf_size` в деревьях принятия решений из Scikit-learn:

он управляет минимальным количеством наблюдений, которые дерево может содержать в одном листе.

Опробуем диапазон значений и посмотрим, какое из них работает лучше всего. В дополнение к значению по умолчанию (1) мы можем установить 10 и 30 (рис. 6.36).



**Рис. 6.36.** Оптимальное значение `min_child_weight` равно 1, однако оно не слишком отличается от других значений этого параметра

Из рис. 6.36 мы видим, что:

- для `min_child_weight=1` AUC составляет 83,6 %;
- для `min_child_weight=10` AUC составляет 83,3 %;
- для `min_child_weight=30` AUC составляет 83,5 %.

Разница между этими параметрами незначительна, поэтому мы оставим значение по умолчанию.

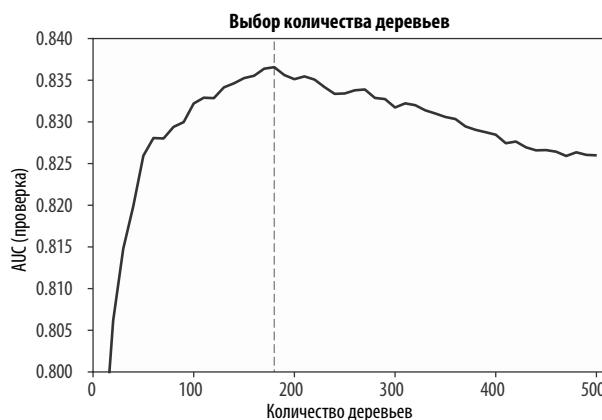
Параметрами для нашей окончательной модели будут:

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

Нам предстоит сделать последний шаг, прежде чем мы закончим модель: необходимо выбрать оптимальное количество деревьев. Это довольно просто: найдите итерацию, при которой оценка проверки достигла максимума, и используйте это число.

В нашем случае нам нужно обучить 180 деревьев для окончательной модели (рис. 6.37):

```
[160] train-auc:0.935513    val-auc:0.835536
[170] train-auc:0.937885    val-auc:0.836384
[180] train-auc:0.93971     val-auc:0.836565 <- best
[190] train-auc:0.942029    val-auc:0.835621
[200] train-auc:0.943343    val-auc:0.835124
```



**Рис. 6.37.** Оптимальное количество деревьев для окончательной модели — 180

Лучшее, чего смогла добиться модель случайного леса, — 82,5 % AUC, тогда как лучшим показателем модели градиентного бустинга стали 83,6 %, что на 1 % больше.

Это лучшая модель, поэтому будем использовать ее в качестве окончательной, то есть для оценки заявок на получение кредита.

#### 6.4.4. Тестирование окончательной модели

Мы почти готовы использовать модель для оценки рисков. Нам осталось сделать две вещи:

- снова обучить окончательную модель на основе обучающего и проверочного наборов данных, вместе взятых. Нам больше не требуется проверочный набор данных, поэтому мы можем использовать больше данных для обучения, что слегка улучшит модель;

- протестируйте модель на тестовом наборе. Это та часть данных, которую мы отложили в самом начале. Пришло время использовать ее, чтобы убедиться, что модель не переобучена и хорошо работает на совершенно неизвестных данных.

Следующие шаги представлены ниже:

- проделайте с `df_full_train` и `df_test` ту же предварительную обработку, что и с `df_train` и `df_val`. В результате мы получим матрицы признаков `X_train` и `X_test`, а также целевые переменные `y_train` и `y_test`;
- обучите модель на объединенном наборе данных с параметрами, которые мы выбрали ранее;
- примените модель к тестовым данным, чтобы получить тестовые прогнозы;
- убедитесь, что модель работает хорошо и не переобучена.

Сделаем все это. Сначала создадим целевую переменную:

```
y_train = (df_train_full.status == 'default').values
y_test = (df_test.status == 'default').values
```

Поскольку мы используем весь датафрейм для создания матрицы признаков, нам необходимо удалить целевую переменную:

```
del df_train_full['status']
del df_test['status']
```

Далее мы преобразуем датафреймы в списки словарей, а затем используем прямое кодирование для получения матриц признаков:

```
dict_train = df_train_full.fillna(0).to_dict(orient='records')
dict_test = df_test.fillna(0).to_dict(orient='records')

dv = DictVectorizer(sparse=False)
X_train = dv.fit_transform(dict_train)
X_test = dv.transform(dict_test)
```

Наконец, мы обучим модель XGBoost, используя эти данные и оптимальные параметры, которые мы выяснили ранее:

```
dtrain = xgb.DMatrix(X_train, label=y_train, feature_names=dv.feature_names_)
dtest = xgb.DMatrix(X_test, label=y_test, feature_names=dv.feature_names_)

xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
```

```
'silent': 1  
}  
  
num_trees = 160  
  
model = xgb.train(xgb_params, dtrain, num_boost_round=num_trees)
```

Затем оценим производительность на тестовом наборе:

```
y_pred_xgb = model.predict(dtest)  
roc_auc_score(y_test, y_pred_xgb)
```

Результат составляет 83,2 %, что сопоставимо с 83,6 % — производительностью на проверочном наборе. Это означает, что наша модель не переобучена и может хорошо работать с клиентами, с которыми до этого не встречалась.

### Упражнение 6.4

В чем заключается основное различие между случайным лесом и градиентным бустингом?

- А. Деревья при градиентном бустинге обучаются последовательно, и каждое следующее дерево улучшает предыдущее. В случайному лесу все деревья обучаются независимо.
- Б. Градиентный бустинг намного быстрее, чем случайный лес.
- В. Деревья в случайному лесу обучаются последовательно, и каждое следующее дерево улучшает предыдущее. При градиентном бустинге все деревья обучаются независимо.

## 6.5. СЛЕДУЮЩИЕ ШАГИ

Мы коснулись основ деревьев решений, случайному леса и градиентного бустинга. Мы многому научились, но всю информацию в одну главу, к сожалению, не вместить. Вы можете изучить эту тему более основательно, выполнив упражнения.

### 6.5.1. Упражнения

- Конструирование признаков представляет собой процесс создания новых признаков из существующих. Для этого проекта мы не создавали никаких признаков; мы просто использовали те, которые предоставлены в наборе данных. Добавление дополнительных признаков может повысить производительность нашей модели. Например, мы можем добавить соотношение запрашиваемых

денег к общей цене товара. Поэкспериментируйте с созданием дополнительных признаков.

- При обучении случайного леса мы получаем разные модели, выбирая случайное подмножество объектов для каждого дерева. Чтобы управлять размером подмножества, мы используем `max_features`. Попробуйте настроить этот параметр и посмотрите, изменит ли он значение AUC при проверке.
- Экстремальные рандомизированные деревья (или сокращенно экстра-деревья) — разновидность случайного леса, в котором идея рандомизации доведена до крайности. Вместо поиска наилучшего возможного разделения условие разделения выбирается случайным образом. Такой подход имеет ряд преимуществ: экстра-деревья быстрее обучаются и менее подвержены переобучению. С другой стороны, обеспечение адекватной производительности требует большего количества деревьев. В Scikit-learn это реализуется с помощью `ExtraTreesClassifier` из пакета `ensemble`. Поэкспериментируйте с ними в рамках проекта.
- В XGBoost параметр `colsample_bytree` управляет количеством признаков, которые мы выбираем для каждого дерева, — это похоже на `max_features` для случайного леса. Поэкспериментируйте с этим параметром и посмотрите, улучшает ли он производительность: попробуйте значения от 0,1 до 1,0 с шагом 0,1. Обычно оптимальные значения находятся в диапазоне от 0,6 до 0,8, но иногда 1,0 дает наилучший результат.
- В дополнение к случайному выбору столбцов (признаков) мы можем выбрать подмножество строк (клиентов). Это называется *подвыборкой* (*subsampling*) и помогает предотвратить переобучение. В XGBoost параметр `subsample` управляет долей примеров, которые мы выбираем для обучения каждого дерева в ансамбле. Попробуйте значения от 0,4 до 1,0 с шагом 0,1. Обычно оптимальные значения находятся в диапазоне от 0,6 до 0,8.

## 6.5.2. Другие проекты

- Все древовидные модели могут решить проблему регрессии — спрогнозировать число. В Scikit-learn, `DecisionTreeRegressor` и `RandomForestRegressor` реализуют регрессионную вариацию моделей. В XGBoost нам потребуется изменить цель на `reg:squarederror`. Используйте эти модели для прогнозирования цены автомобиля, а также попытайтесь решить другие регрессионные задачи.

## РЕЗЮМЕ

- Дерево решений — это модель, представляющая последовательность решений «если-то-иначе». Ее легко понять, и она довольно хорошо работает на практике.
- Мы обучаем деревья решений, выбирая наилучшее разделение с помощью показателей примеси. Основные параметры, которые мы контролируем, — глубина дерева и максимальное количество образцов в каждом листе.

- Случайный лес — способ объединить множество деревьев решений в одну модель. Подобно команде экспертов, отдельные деревья могут совершать ошибки, но когда они работают вместе, вероятность того, что они придут к неверному решению, уменьшается.
- Чтобы получить хорошие прогнозы, случайный лес должен состоять из разнообразного набора моделей. Вот почему каждое дерево в модели использует для обучения собственный набор признаков.
- Основные параметры, которые нам нужно изменить для случайного леса, те же, что и для деревьев принятия решений: глубина и максимальное количество примеров в каждом листе. Кроме того, нам необходимо выбрать количество деревьев, которые мы хотим получить в ансамбле.
- В то время как в случайному лесу деревья независимы, при градиентном бустинге деревья последовательны, и каждая следующая модель исправляет ошибки предыдущей. В некоторых случаях это приводит к повышению эффективности прогнозирования.
- Параметры, которые нам нужно настроить для градиентного бустинга, аналогичны параметрам для случайному леса: глубина, максимальное количество примеров в листе и количество деревьев. В дополнение к этому у нас есть `eta` — скорость обучения. Она определяет вклад каждого отдельного дерева в ансамбль.

Древовидные модели легко интерпретировать и понимать, и зачастую они работают довольно хорошо. Градиентный бустинг работает отлично и часто обеспечивает наилучшую возможную производительность для структурированных данных (данных в табличном формате).

В следующей главе мы рассмотрим нейронные сети: другой тип модели, который, напротив, обеспечивает наилучшую производительность при работе с неструктурированными данными, такими как изображения.

## ОТВЕТЫ К УПРАЖНЕНИЯМ

- Упражнение 6.1. Ответ А. При наличии еще одного признака обучение занимает больше времени.
- Упражнение 6.2. Ответ Б. Случайный выбор различающегося подмножества признаков для каждого дерева.
- Упражнение 6.3. Ответ В. Достижение максимальной производительности модели потребует меньшего количества деревьев.
- Упражнение 6.4. Ответ А. Деревья при градиентном бустинге обучаются последовательно. В случайному лесу деревья обучаются независимо.



# *Нейронные сети и глубокое обучение*

## **В этой главе**

- ✓ Сверточные нейронные сети для классификации изображений.
- ✓ TensorFlow и Keras — фреймворки для построения нейронных сетей.
- ✓ Использование предварительно обученных нейронных сетей.
- ✓ Сверточная нейронная сеть изнутри.
- ✓ Обучение с переносом опыта.
- ✓ Расширение данных — процесс генерации большего количества обучающих данных.

Ранее мы имели дело только с табличными данными в CSV-файлах. В этой главе будем работать с совершенно другим типом данных — изображениями.

Проект, который мы подготовили для данной главы, — классификация одежды. Мы спрогнозируем, является ли элемент одежды на изображении футболкой, рубашкой, юбкой, платьем или чем-то еще.

Это одна из задач классификации изображений. Чтобы ее решить, мы выясним, как обучить глубокую нейронную сеть распознавать типы одежды с помощью

TensorFlow и Keras. Материалы этой главы помогут вам начать использовать нейронные сети и выполнить любой подобный проект по классификации изображений.

Приступим!

## 7.1. МОДНАЯ КЛАССИФИКАЦИЯ

Представьте, что мы работаем в интернет-магазине по продаже модной одежды. В попытках продать свою одежду наши пользователи загружают тысячи изображений ежедневно. Мы хотим помочь им быстрее создавать списки, автоматически рекомендуя нужную категорию для их товаров.

Нам требуется модель для классификации изображений. Ранее мы уже рассматривали ряд моделей классификации: логистическую регрессию, деревья решений, случайные леса и градиентный бустинг. Эти модели отлично работают с табличными данными, но их довольно сложно использовать для изображений.

Для нашей задачи нам нужен другой тип модели: сверточная нейронная сеть, специальная модель, используемая для изображений. Такие нейронные сети состоят из множества слоев, и именно поэтому их часто называют «глубокими». Глубокое обучение — часть машинного обучения, которая работает с глубокими нейронными сетями.

Фреймворки для обучения этих моделей также отличаются от тех, что мы встречали ранее, поэтому в данной главе мы используем TensorFlow и Keras вместо Scikit-learn.

План нашего проекта таков:

- сначала мы загрузим набор данных и используем предварительно подготовленную модель для классификации изображений;
- затем поговорим о нейронных сетях и рассмотрим их внутреннее устройство;
- после этого настроим предварительно обученную нейронную сеть под решение собственных задач;
- наконец, расширим наш набор данных, сгенерировав множество других изображений из имеющихся.

Для оценки качества наших моделей будем использовать достоверность: процент вещей, которые мы классифицировали правильно.

Всю теорию, лежащую в основе глубокого обучения, невозможно изложить в одной главе. В данной книге мы сосредоточимся на самых фундаментальных частях, которых достаточно для завершения проекта этой главы и других

подобных проектов по классификации изображений. Подробную информацию о концепциях, которые не требуются для завершения этого проекта, мы можем получить в CS231n — курсе о нейронных сетях от Стэнфордского университета. Конспекты курса доступны онлайн по адресу [cs231n.github.io](https://cs231n.github.io).

Код для этого проекта доступен в репозитории книги на GitHub по адресу <https://github.com/alexeygrigorev/mlbookcamp-code>, в папке chapter-07-neural-nets. В этой папке находится несколько блокнотов. Для большей части главы нам понадобится `07-neural-nets-train.ipynb`. Для раздела 7.5 мы используем `07-neural-nets-test.ipynb`.

### 7.1.1. GPU vs CPU

Обучение нейронной сети — процесс, требующий больших вычислительных затрат, и его ускорение требует мощного аппаратного обеспечения. Чтобы ускорить обучение, мы обычно используем GPU — графические процессоры, или, проще говоря, графические карты.

Для данной главы графический процессор не потребуется. Вы сможете сделать все на своем ноутбуке, но без графического процессора все будет примерно в восемь раз медленнее, чем с ним.

Если у вас есть графическая карта, то ее использование потребует установки специальных драйверов от TensorFlow. (Проверьте официальную документацию TensorFlow для получения более подробной информации: <https://www.tensorflow.org/install/gpu>.) В качестве альтернативы вы можете арендовать предварительно сконфигурированный GPU-сервер. Например, мы можем использовать AWS SageMaker для аренды экземпляра Jupyter Notebook, в котором все уже настроено. Подробную информацию о том, как использовать SageMaker, можно найти в приложении Д. У других облачных провайдеров тоже есть серверы с графическими процессорами, но мы не рассматриваем их в этой книге. Независимо от используемой вами среды код работает в любом месте, где возможно установить Python и TensorFlow.

Приняв решение о том, где запускать код, мы можем перейти к следующему шагу: загрузке набора данных.

### 7.1.2. Загрузка набора данных

Сначала создадим папку для этого проекта и назовем ее `07-neural-nets`.

Для этого проекта нам потребуется набор данных. Мы будем использовать подмножество набора данных (для получения дополнительной информации проверьте <https://github.com/alexeygrigorev/clothing-dataset>), который содержит около

3800 изображений одежды десяти различных классов. Данные доступны в репозитории GitHub. Клонируем их:

```
git clone https://github.com/alexeygrigorev/clothing-dataset-small.git
```

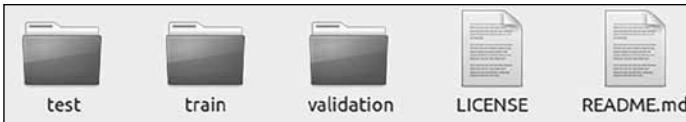
Если вы делаете это в AWS SageMaker, то можете выполнить такую команду в ячейке блокнота. Просто добавьте восклицательный знак (!) перед командой (рис. 7.1).

```
!git clone https://github.com/alexeygrigorev/clothing-dataset-small.git
Cloning into 'clothing-dataset-small'...
remote: Enumerating objects: 3839, done.
remote: Counting objects: 100% (400/400), done.
remote: Compressing objects: 100% (400/400), done.
remote: Total 3839 (delta 9), reused 384 (delta 0), pack-reused 3439
Receiving objects: 100% (3839/3839), 100.58 MiB | 1.21 MiB/s, done.
Resolving deltas: 100% (10/10), done.
Checking out files: 100% (3783/3783), done.
```

**Рис. 7.1.** Выполнение команды сценария оболочки в Jupyter: просто добавьте восклицательный знак (!) перед командой

Набор данных уже разбит на папки (рис. 7.2):

- **train** — изображения для обучения модели (3068 изображений);
- **validation** — изображения для проверки (341 изображение);
- **test** — изображения для тестирования (372 изображения).



**Рис. 7.2.** Набор данных уже разделен на обучающий, проверочный и тестовый

Каждая из этих папок имеет десять вложенных папок: по одной для каждого типа одежды (рис. 7.3).



**Рис. 7.3.** Изображения в наборе данных организованы в виде вложенных папок

Как мы видим, набор данных содержит десять классов одежды, от платьев и шляп до шорт и обуви.

Каждая вложенная папка содержит изображения только одного класса (рис. 7.4).



**Рис. 7.4.** Содержимое папки «Штаны»

Предметы одежды на этих фотографиях имеют разные цвета и различный фон. Одни предметы лежат на полу, вторые разложены на кровати или столе, а третьи развешаны на нейтральном фоне.

При подобном разнообразии изображений невозможно использовать методы, которые мы изучали ранее. Нам нужен особый тип модели: нейронные сети. Эта модель потребует использования различных инструментов, и мы рассмотрим их далее.

### 7.1.3. TensorFlow и Keras

Если вы используете AWS SageMaker, то вам не нужно ничего устанавливать: все необходимые библиотеки там уже присутствуют.

Но если вы используете свой ноутбук с Anaconda или запускаете код где-то еще, то вам необходимо установить TensorFlow — библиотеку для построения нейронных сетей.

Используйте pip:

```
pip install tensorflow
```

TensorFlow — низкоуровневый фреймворк, и им не всегда легко пользоваться. В этой главе мы используем Keras — библиотеку более высокого уровня, построенную поверх TensorFlow. Keras значительно упрощает обучение нейронных сетей. Он поставляется вместе с TensorFlow, поэтому нам не нужно устанавливать ничего дополнительного.

### **ПРИМЕЧАНИЕ**

Раньше Keras не был частью TensorFlow, и в Интернете вы можете найти много примеров, где это все еще отдельная библиотека. Однако интерфейс Keras существенно не изменился, поэтому большинство примеров, которые вы найдете, по-прежнему работают в новом Keras.

На момент написания статьи последней версией TensorFlow была 2.3.0, а AWS SageMaker использовал TensorFlow версии 2.1.0. Разница в версиях не является проблемой; код из этой главы работает для обеих версий и, скорее всего, будет работать для всех версий TensorFlow 2.

Мы готовы начинать и в первую очередь создадим блокнот под названием `chapter-07-neural-nets`. Как обычно, начинаем с импорта NumPy и Matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Далее импортируем TensorFlow и Keras:

```
import tensorflow as tf
from tensorflow import keras
```

Подготовительная работа завершена, и теперь можно более пристально взглянуть на имеющиеся у нас изображения.

#### **7.1.4. Загрузка изображений**

Keras предлагает специальную функцию для загрузки изображений — `load_img`. Сразу ее импортируем:

```
from tensorflow.keras.preprocessing.image import load_img
```

### **ПРИМЕЧАНИЕ**

Когда Keras был отдельным пакетом, импорт выглядел следующим образом:

```
from keras.preprocessing.image import load_img
```

Если вы обнаружили какой-то старый код Keras в Интернете и хотите использовать его с последними версиями TensorFlow, то просто добавьте `tensorflow`.

в начале при его импорте. Скорее всего, этого хватит, чтобы заставить его работать.

Воспользуемся данной функцией, чтобы взглянуть на одно из изображений:

```
path = './clothing-dataset-small/train/t-shirt'  
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'  
fullname = path + '/' + name  
load_img(fullname)
```

Когда ячейка будет выполнена, мы должны увидеть изображение футболки (рис. 7.5).

```
path = './clothing-dataset-small/train/t-shirt'  
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'  
fullname = path + '/' + name  
load_img(fullname)
```



**Рис. 7.5.** Изображение футболки из обучающего набора

Чтобы использовать это изображение в нейронной сети, нам нужно изменить его размер, поскольку модели всегда ожидают изображения определенного размера.

Например, для сети, которую мы используем в текущей главе, требуется изображение размером  $150 \times 150$  или  $299 \times 299$ .

Чтобы изменить размер изображения, укажите параметр `target_size`:

```
load_img(fullname, target_size=(299, 299))
```

В результате изображение становится квадратным и немного сплющенным (рис. 7.6).



**Рис. 7.6.** Чтобы изменить размер изображения, используйте параметр `target_size`

Теперь применим нейронную сеть, чтобы классифицировать это изображение.

## 7.2. СВЕРТОЧНЫЕ НЕЙРОННЫЕ СЕТИ

Нейронные сети — класс моделей машинного обучения, предназначенных для решения задач классификации и регрессии. Перед нами стоит задача классификации, и нам нужно выяснить категорию изображения.

Однако наша задача особенная: мы имеем дело с изображениями. Вот почему нам нужен особый тип нейронной сети — сверточная нейронная сеть, которая может извлекать визуальные закономерности из изображения и использовать их для прогнозов.

Предварительно обученные нейронные сети доступны в Интернете, поэтому посмотрим, как мы можем использовать одну из них для этого проекта.

### 7.2.1. Использование предварительно обученной модели

Обучение сверточной нейронной сети с нуля — трудоемкий процесс, требующий большого объема данных и мощного оборудования. Работа с большими наборами данных, такими как ImageNet с 14 миллионами изображений, может потребовать недель непрерывного обучения. (На [image-net.org](http://image-net.org) можно получить дополнительную информацию.)

К счастью, нам не нужно делать это самостоятельно: нам доступны предварительно обученные модели. Обычно они обучаются в ImageNet и могут быть использованы для классификации изображений общего назначения.

Все очень просто, и нам даже не потребуется ничего скачивать — Keras позабочится об этом автоматически. Мы можем использовать множество различных типов моделей (называемых *архитектурами*). В официальной документации Keras вы можете найти хороший обзор доступных предварительно обученных моделей (<https://keras.io/api/applications/>).

В этой главе мы будем использовать Xception, относительно небольшую модель, обладающую хорошей производительностью. В первую очередь нам нужно импортировать саму модель и некоторые полезные функции:

```
from tensorflow.keras.applications.xception import Xception
from tensorflow.keras.applications.xception import preprocess_input
from tensorflow.keras.applications.xception import decode_predictions
```

Мы импортировали три вещи:

- `Xception` — фактическую модель;
- `preprocess_input` — функцию для подготовки изображения к использованию моделью;
- `decode_prediction` — функцию для декодирования предсказания модели.

Загрузим эту модель:

```
model = Xception(
    weights='imagenet',
    input_shape=(299, 299, 3)
)
```

Здесь мы указываем два параметра:

- `weights` — мы хотим использовать предварительно обученную модель из ImageNet;
- `input_shape` — размер входных изображений: высота, ширина и количество каналов. Мы изменяем размер изображений до  $299 \times 299$ , и каждое изображение имеет три канала: красный, зеленый и синий.

При первой загрузке скачивается актуальная модель из Интернета. После этого мы сможем ею пользоваться.

Протестируем ее на изображении, которое мы видели ранее. Сначала мы загружаем его с помощью функции `load_img`:

```
img = load_img(fullname, target_size=(299, 299))
```

Переменная `img` — это объект `Image`, который нам нужно преобразовать в массив NumPy. Это легко:

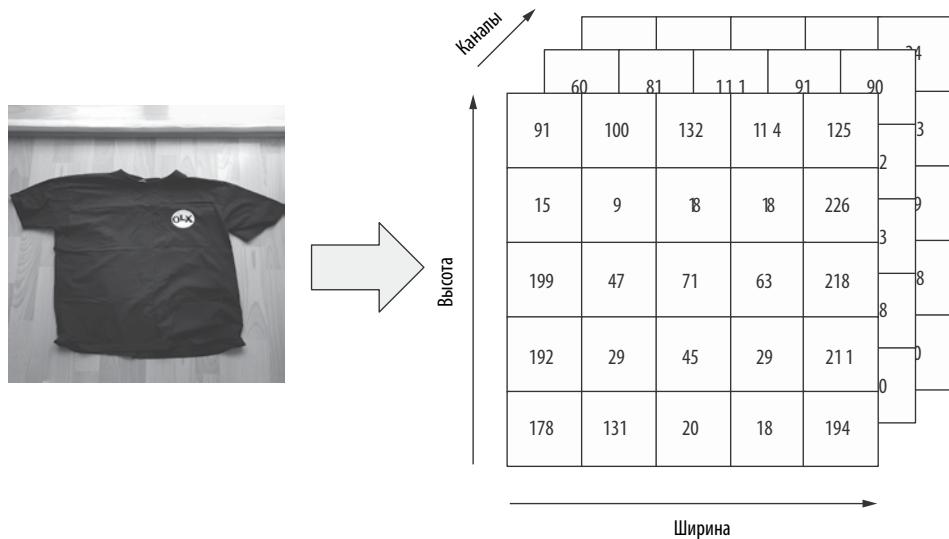
```
x = np.array(img)
```

Этот массив должен получить ту же форму, что и изображение. Проверим:

```
x.shape
```

Мы видим `(299, 299, 3)`. Он содержит три измерения (рис. 7.7):

- ширина изображения — 299;
- высота изображения — 299;
- количество цветовых каналов — красный, зеленый, синий.



**Рис. 7.7.** После преобразования изображение становится числовым массивом формы ширина × высота × количество каналов

Это соответствует форме ввода, которую мы указали при загрузке нейронной сети. Однако модель не ожидает только одного изображения. Она получает

пакет — несколько изображений, собранных в один массив. Такой массив должен иметь четыре измерения:

- количество изображений;
- ширину;
- высоту;
- количество каналов.

Например, для десяти изображений форма равна `(10, 299, 299, 3)`. Поскольку у нас есть только одно изображение, нам нужно создать пакет именно с ним:

```
X = np.array([x])
```

#### ПРИМЕЧАНИЕ

Если бы у нас было несколько изображений, например `x, y` и `z`, то мы бы написали

```
X = np.array([x, y, z])
```

Проверим, что у нас получилось:

```
X.shape
```

Мы видим `(1, 299, 299, 3)` — это одно изображение размером  $299 \times 299$  с тремя каналами.

Прежде чем мы сможем применить модель к нашему изображению, нам нужно подготовить его с помощью функции `preprocess_input`:

```
X = preprocess_input(X)
```

Эта функция преобразует целые числа от 0 до 255 в исходном массиве в числа от -1 до 1.

Теперь мы готовы использовать модель.

### 7.2.2. Получение прогнозов

Чтобы применить модель, используйте метод `predict`:

```
pred = model.predict(X)
```

Взглянем на этот массив:

```
pred.shape
```

Он довольно большой — содержит 1000 элементов (рис. 7.8).

```

pred = model.predict(X)

pred.shape
(1, 1000)

pred[0, :10]

array([0.0003238 , 0.00015736, 0.00021406, 0.00015296, 0.00024657,
       0.00030446, 0.00032349, 0.00014726, 0.00020487, 0.00014866],
      dtype=float32)

```

**Рис. 7.8.** Выходные данные предварительно обученной модели Xception

Модель Xception предсказывает, принадлежит ли изображение к одному из 1000 классов, поэтому каждый элемент в массиве прогнозов представляет собой вероятность принадлежности к одному из этих классов.

Мы не знаем, что это за классы, поэтому трудно понять смысл прогноза, просто взглянув на цифры. К счастью, мы можем использовать функцию `decode_predictions`, которая декодирует предсказание в осмысленные имена классов:

```
decode_predictions(pred)
```

Функция показывает пять наиболее вероятных классов для этого изображения:

```
[[('n02667093', 'abaya', 0.028757658),
  ('n04418357', 'theater_curtain', 0.020734021),
  ('n01930112', 'nematode', 0.015735716),
  ('n03691459', 'loudspeaker', 0.013871926),
  ('n03196217', 'digital_clock', 0.012909736)]]
```

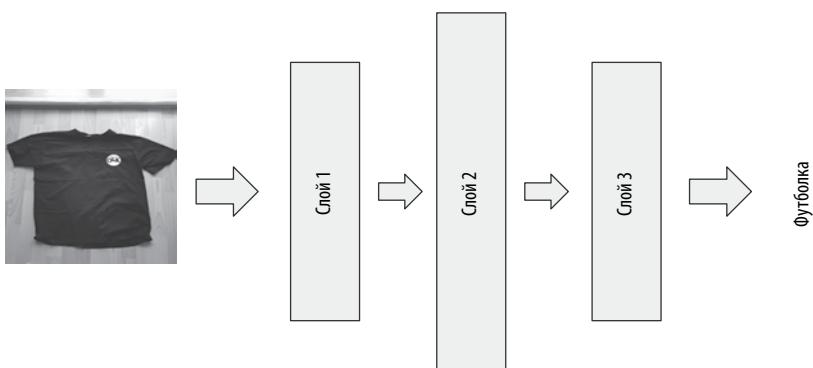
Не совсем тот результат, которого мы ожидали. Скорее всего, изображения, подобные этой футболке, не распространены в ImageNet, и именно поэтому результат бесполезен для нашей задачи.

Несмотря на то что эти результаты не слишком полезны, мы можем использовать эту нейронную сеть в качестве базовой модели для решения своей задачи.

Чтобы понять, как это сделать, мы должны сначала получить представление о том, как вообще работают сверточные нейронные сети. Посмотрим, что происходит внутри модели, когда мы вызываем метод `predict`.

## 7.3. ВНУТРИ МОДЕЛИ

Все нейронные сети организованы в слои. Мы берем изображение, пропускаем его через все слои и в конце концов получаем прогнозы (рис. 7.9).



**Рис. 7.9.** Нейронная сеть состоит из нескольких слоев

Обычно модель состоит из множества слоев. Например, у модели Xception, которую мы используем, 71 слой. Вот почему эти нейронные сети называются «глубокими» нейронными сетями — потому что у них множество слоев.

Для сверточной нейронной сети наиболее важными слоями являются:

- сверточные;
- плотные.

Сначала посмотрим на сверточные слои.

### 7.3.1. Сверточные слои

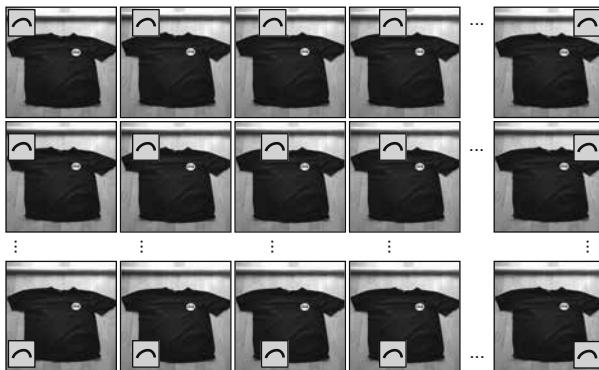
Несмотря на то что «сверточный слой» звучит загадочно, это не более чем набор *фильтров* — небольших «изображений» с простыми формами, такими как полосы (рис. 7.10).



**Рис. 7.10.** Примеры фильтров для сверточного слоя (не из реальной сети)

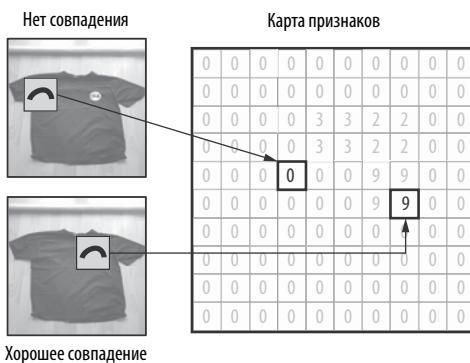
Фильтры в сверточном слое изучаются моделью во время обучения. Однако, поскольку мы используем предварительно обученную нейронную сеть, нам не нужно беспокоиться об этом; у нас фильтры уже готовы.

Чтобы применить сверточный слой к изображению, мы перемещаем каждый фильтр по этому изображению. Например, мы можем перемещать его слева направо и сверху вниз (рис. 7.11).



**Рис. 7.11.** Чтобы применить фильтр, мы перемещаем его по изображению

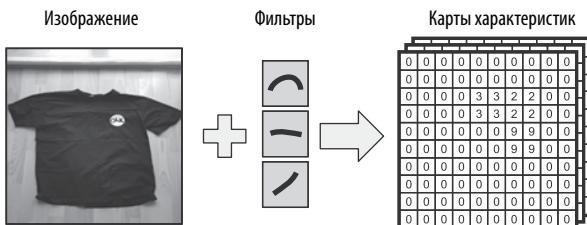
При движении мы сравниваем содержимое фильтра с содержимым изображения под ним. Для каждого сравнения записываем степень сходства. Так мы получаем *карту характеристик* — массив с числами, где большое число означает совпадение между фильтром и изображением, а малое — отсутствие совпадения (рис. 7.12).



**Рис. 7.12.** Карта характеристик — результат применения фильтра к изображению. Высокое значение на карте соответствует областям с высокой степенью сходства между изображением и фильтром

Таким образом, карта характеристик сообщает нам, где на изображении мы можем обнаружить фигуру из фильтра.

Один сверточный слой состоит из множества фильтров, поэтому мы фактически получаем несколько карт характеристик — по одной для каждого фильтра (рис. 7.13).

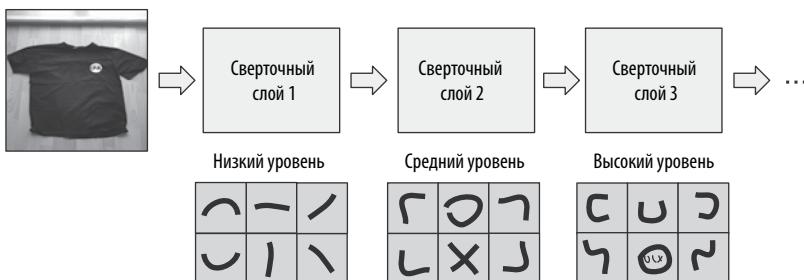


**Рис. 7.13.** Каждый сверточный слой содержит множество фильтров, поэтому мы получаем набор карт характеристик: по одной для каждого используемого фильтра

Теперь можно взять выходные данные одного сверточного слоя и использовать их в качестве входных данных для следующего.

Из предыдущего слоя мы знаем расположение различных полос и других простых форм. Две простые фигуры при встрече образуют более сложные узоры: кресты, углы или круги.

Это то, что делают фильтры следующего слоя: объединяют формы из предыдущего в более сложные структуры. Чем глубже мы погружаемся в сеть, тем более сложные паттерны может распознать сеть (рис. 7.14).



**Рис. 7.14.** Более глубокие сверточные слои могут обнаруживать все более усложняющиеся характеристики изображения

Мы повторяем этот процесс, чтобы обнаружить все более и более сложные формы. Таким образом сеть «усваивает» некоторые отличительные особенности изображения. Что касается одежды, то это могут быть короткие или длинные рукава или тип горловины. Для животных это могут быть заостренные или висячие уши или наличие усов.

В конце мы получаем векторное представление изображения: одномерный массив, каждая позиция которого соответствует некоторым визуальным особенностям высокого уровня.

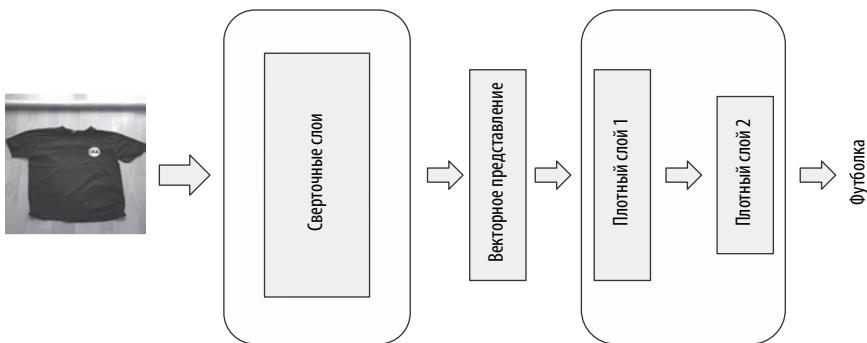
Одни части массива могут соответствовать рукавам, в то время как другие части представляют уши и усы. На данном уровне обычно трудно разобраться в этих

признаках, но они различаются достаточно, чтобы отделять футбольку от брюк или кошку от собаки.

Теперь нам нужно использовать это векторное представление, чтобы объединить высокоуровневые характеристики и прийти к окончательному решению. Для этого мы используем другой вид слоев — плотные слои.

### 7.3.2. Плотные слои

Плотные слои обрабатывают векторное представление изображения и переводят эти визуальные особенности в реальный класс: футболку, платье, куртку или что-то другое (рис. 7.15).



**Рис. 7.15.** Сверточные слои преобразуют изображение в его векторное представление, а плотные преобразуют векторное представление в фактическую метку

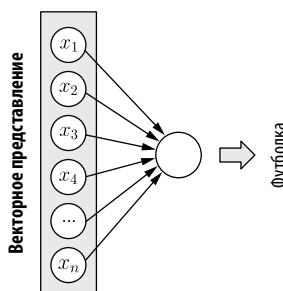
Чтобы понять, как это работает, сделаем шаг назад и поразмышляем над тем, как можно с помощью логистической регрессии классифицировать изображение.

Предположим, мы хотим построить модель бинарной классификации для прогнозирования того, является ли изображение футболкой. В этом случае входными данными для логистической регрессии является векторное представление изображения — вектор признаков  $x$ .

Из главы 3 мы знаем, что для получения прогноза необходимо объединить признаки в  $x$  с вектором весов  $w$ , а затем применить сигмоидальную функцию для получения окончательного прогноза:

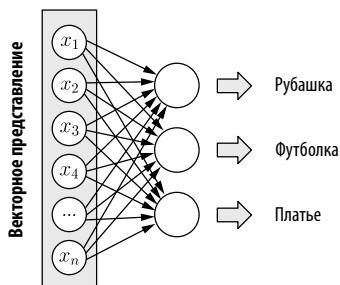
$$\text{sigmoid}(x^T w)$$

Это можно представить визуально, взяв все компоненты вектора  $x$  и соединив их с результатом (вероятностью прогноза «футболка») (рис. 7.16).



**Рис. 7.16.** Логистическая регрессия: мы берем все компоненты вектора признаков  $x$  и объединяем для получения прогноза

Что если нам нужны прогнозы для нескольких классов? Например, нам требуется выяснить, какой предмет перед нами: футболка, рубашка или платье. В этом случае мы можем построить несколько логистических регрессий — по одной для каждого класса (рис. 7.17).



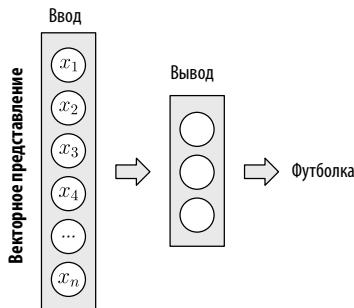
**Рис. 7.17.** Для прогнозирования нескольких классов мы обучаем несколько моделей логистической регрессии

Объединив несколько моделей логистической регрессии, мы только что создали небольшую нейронную сеть!

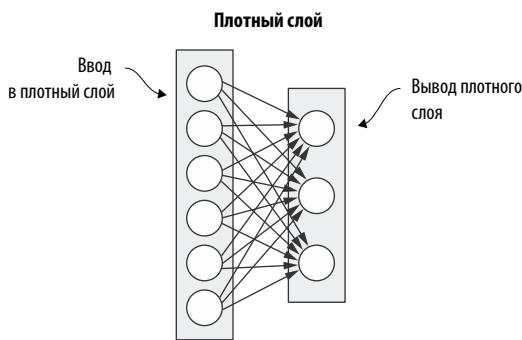
Чтобы визуально это упростить, мы можем собрать выходные данные в один слой — выходной (рис. 7.18).

Имея для прогнозирования десять классов, мы получаем десять элементов в выходном слое. Чтобы сделать прогноз, мы смотрим на каждый элемент выходного слоя и выбираем тот, который получил наибольшую оценку.

В данном случае у нас сеть с одним слоем, который преобразует входные данные в выходные. Такой слой называется *плотным*, поскольку соединяет каждый элемент ввода со всеми элементами своего вывода. Именно поэтому такие слои иногда называют «полносвязными» (рис. 7.19).

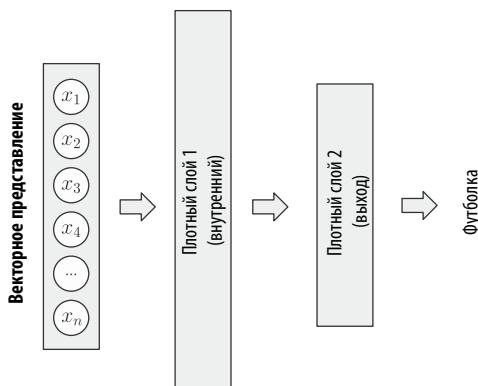


**Рис. 7.18.** Несколько логистических регрессий, собранных воедино, образуют небольшую нейронную сеть



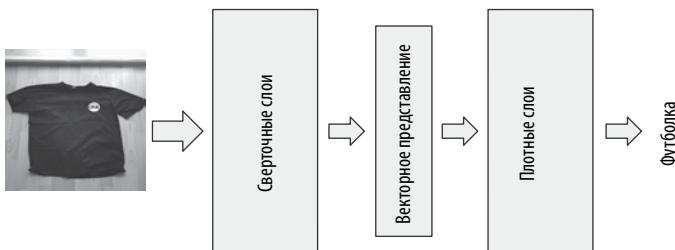
**Рис. 7.19.** Плотный слой соединяет каждый элемент своего ввода с каждым элементом своего вывода

Однако мы можем не останавливаться лишь на одном выходном слое. Можно добавить больше слоев между вводом и конечным результатом (рис. 7.20).



**Рис. 7.20.** Нейронная сеть с двумя слоями: одним внутренним и одним выходным

Итак, после вызова `predict` изображение сначала проходит через череду сверточных слоев. Таким образом мы извлекаем векторное представление этого изображения. Далее уже векторное представление проходит через ряд плотных слоев, и мы получаем окончательный прогноз (рис. 7.21).



**Рис. 7.21.** В сверточной нейронной сети изображение сначала проходит через ряд сверточных слоев, а затем через ряд плотных

#### ПРИМЕЧАНИЕ

В этой книге мы даем лишь упрощенный и высокоуровневый обзор внутренней работы сверточных нейронных сетей. В дополнение к сверточным и плотным слоям существует множество других. Более полно с данной темой можно ознакомиться в примечаниях CS231n ([cs231n.github.io/convolutional-networks](https://cs231n.github.io/convolutional-networks)).

Теперь вернемся к коду и посмотрим, как можно настроить предварительно обученную нейронную сеть для нашего проекта.

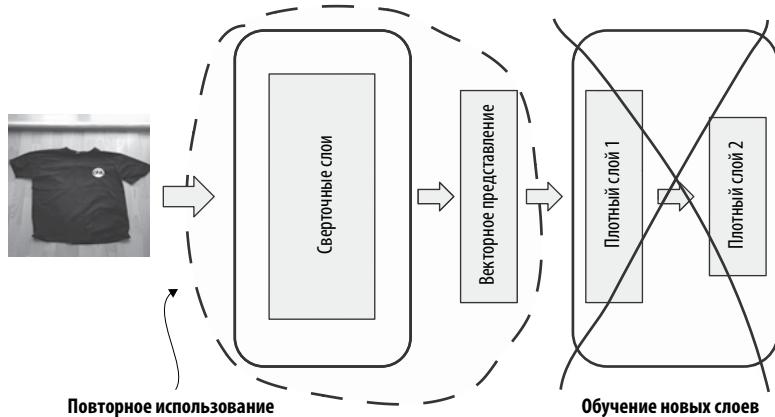
## 7.4. ОБУЧЕНИЕ МОДЕЛИ

Обучение сверточной нейронной сети занимает много времени и требует большого количества данных. Но есть короткий путь: мы можем использовать *обучение с переносом опыта* — подход, при котором адаптируем предварительно подготовленную модель к своей задаче.

### 7.4.1. Обучение с переносом

Сложность в обучении обычно возникает из-за сверточных слоев. Чтобы извлечь хорошее векторное представление из изображения, фильтры должны изучать хорошие шаблоны. Для этого сеть должна увидеть множество разных изображений — чем больше, тем лучше. Но как только мы получаем хорошее векторное представление, тренировать плотные слои становится относительно легко. Это означает, что мы можем взять нейронную сеть, предварительно

обученную в ImageNet, и использовать ее для решения нашей задачи. Данная модель уже усвоила хорошие фильтры. Итак, мы берем эту модель, оставляем сверточные слои и отбрасываем плотные — вместо них обучим новые (рис. 7.22).



**Рис. 7.22.** Чтобы адаптировать предварительно обученную модель к новой области, мы сохраняем старые сверточные слои, но обучаем новые плотные

В данном разделе этим и займемся. Но прежде чем начать, мы должны подготовить наш набор данных.

### 7.4.2. Загрузка данных

В предыдущих главах мы загрузили весь набор данных в память и использовали его для получения  $X$  — матрицы с признаками. С изображениями задача несколько усложняется: у нас может не хватить памяти, чтобы сохранить их все.

Keras уже содержит решение этой проблемы — `ImageDataGenerator`. Вместо всего набора данных изображения загружаются в память небольшими пакетами. Воспользуемся этим:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input )
```

Применяет функцию `preprocess_input` к каждому изображению

Мы уже знаем, что изображения необходимо предварительно обработать с помощью функции `preprocess_input`. Вот почему нам нужно сообщить `ImageDataGenerator`, как требуется подготовить данные.

Теперь у нас имеется генератор, поэтому нам просто нужно указать ему на каталог с данными. Для этого используем метод `flow_from_directory`:

```
train_ds = train_gen.flow_from_directory(  
    "clothing-dataset-small/train", ← Загружает все изображения  
    target_size=(150, 150), ← из обучающего каталога  
    batch_size=32, ← Изменяет размер изображений  
) ← до 150 × 150  
    Загружает изображения  
    пакетами по 32 шт.
```

Для наших первых экспериментов мы используем небольшие изображения размером  $150 \times 150$ . Таким образом, обучение модели пройдет быстрее. Кроме того, небольшой размер позволяет использовать для обучения ноутбук.

В нашем наборе данных десять классов одежды, и изображения каждого хранятся в отдельном каталоге. Например, все футболки хранятся в папке `t-shirt`. Генератор может использовать структуру папок, чтобы определить метку для каждого изображения.

При выполнении ячейки он сообщает нам, сколько изображений и классов нашлось в обучающем наборе:

```
Found 3068 images belonging to 10 classes.
```

Теперь осталось повторить тот же процесс для проверочного набора данных:

```
validation_gen = ImageDataGenerator(  
    preprocessing_function=preprocess_input  
)  
  
val_ds = validation_gen.flow_from_directory(  
    "clothing-dataset-small/validation",  
    target_size=image_size,  
    batch_size=batch_size,  
)
```

Как и ранее, мы используем обучающий набор для обучения модели и проверочный для выбора наилучших параметров.

Итак, данные загружены, и теперь можно обучать модель.

### 7.4.3. Создание модели

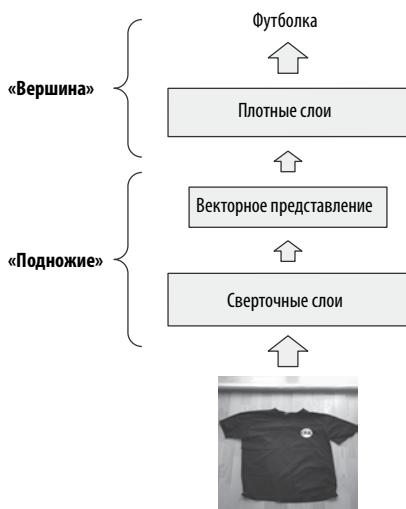
Сначала нам нужно загрузить базовую модель — это предварительно обученная модель, которую мы используем для извлечения векторного представления из изображений. Как и ранее, воспользуемся Xception, однако на сей раз

ограничимся только частью с предварительно обученными сверточными слоями. После этого добавим собственные плотные слои.

Итак, создадим базовую модель:

```
base_model = Xception(  
    weights='imagenet', ← Использует модель,  
    include_top=False ← предварительно обученную  
    input_shape=(150, 150, 3), ← в ImageNet  
) ← Сохраняет только  
          Изображения должны  
          быть размером  
          150 × 150 с тремя  
          каналами
```

Обратите внимание на параметр `include_top`: с его помощью мы явно указываем, что нас интересуют не плотные слои предварительно обученной нейронной сети, а только сверточные. В терминологии Keras «вершина» — это набор конечных уровней сети (рис. 7.23).



**Рис. 7.23.** В Keras вход в сеть расположен внизу, а выход — вверху, поэтому `include_top=False` означает «не включать конечные плотные слои»

Мы не будем обучать базовую модель, так как попытка это сделать приведет к уничтожению всех фильтров. Итак, мы «замораживаем» базовую модель, установив параметр `trainable` в `False`:

```
base_model.trainable = False
```

Теперь построим модель классификации одежды:

```
inputs = keras.Input(shape=(150, 150, 3)) ← Входные изображения должны быть
                                             размером 150 × 150 с тремя каналами
base = base_model(inputs, training=False) ← Использует base_model для извлечения
                                             высокоровневых признаков
vector = keras.layers.GlobalAveragePooling2D()(base) ←
                                                 Извлекает векторное
                                                 представление:
                                                 преобразует
                                                 выходные данные
                                                 base_model в вектор

outputs = keras.layers.Dense(10)(vector) ← Добавляет плотный слой
                                             размером 10: по одному
                                             элементу для каждого
                                             класса

model = keras.Model(inputs, outputs) ← Объединяет входные и выходные
                                             данные в модель Keras
```

То, как мы строим модель, называется функциональным стилем. Поначалу он может показаться непонятным, поэтому мы рассмотрим каждую строку в отдельности.

Сначала мы указываем входные данные и размер ожидаемых массивов:

```
inputs = keras.Input(shape=(150, 150, 3))
```

Далее создаем базовую модель:

```
base = base_model(inputs, training=False)
```

Несмотря на то что `base_model` уже является моделью, мы используем ее как функцию и присваиваем ей два параметра — `inputs` и `training=False`:

- первый указывает, что послужит вводом для `base_model`. Данные поступят из `inputs`;
- второй является необязательным и говорит о том, что нам не требуется обучать базовую модель.

Результатом станет `base`, что представляет собой *функциональный компонент* (как `base_model`), который мы можем комбинировать с другими компонентами. Мы используем его в качестве входных данных для следующего слоя:

```
vector = keras.layers.GlobalAveragePooling2D()(base)
```

Здесь мы создаем объединяющий слой — специальную конструкцию, которая позволяет преобразовывать выходные данные сверточного слоя (трехмерный массив) в вектор (одномерный массив).

Когда он будет создан, мы немедленно его вызываем с `base` в качестве аргумента. Так мы сообщаем, что входные данные для этого слоя поступают из `base`.

Это может несколько сбить с толку, поскольку мы создаем слой и сразу же соединяем его с `base`. Мы можем слегка переписать код, чтобы упростить понимание:

```
pooling = keras.layers.GlobalAveragePooling2D() ← Сначала создает
vector = pooling(base) ← Соединяет его с base объединяющий слой
```

В результате получаем `vector`. Это еще один функциональный компонент, который мы подключим к следующему слою — плотному:

```
outputs = keras.layers.Dense(10)(vector)
```

Аналогично сначала мы создаем слой, а затем подключаем его к `vector`. На данный момент создаем сеть только с одним плотным слоем. Для начала этого вполне достаточно.

Теперь результатом служит `outputs` — конечный результат, который мы хотим получить от сети.

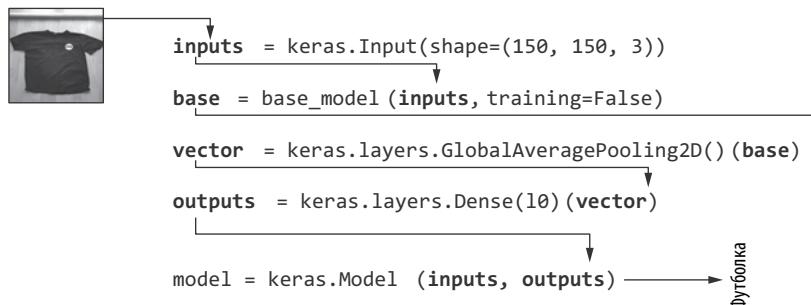
Итак, в нашем случае данные поступают в `inputs` и выходят из `outputs`. Осталось сделать заключительный шаг — обернуть `inputs` и `outputs` в класс `Model`:

```
model = keras.Model(inputs, outputs)
```

Здесь нам потребуется указать два параметра:

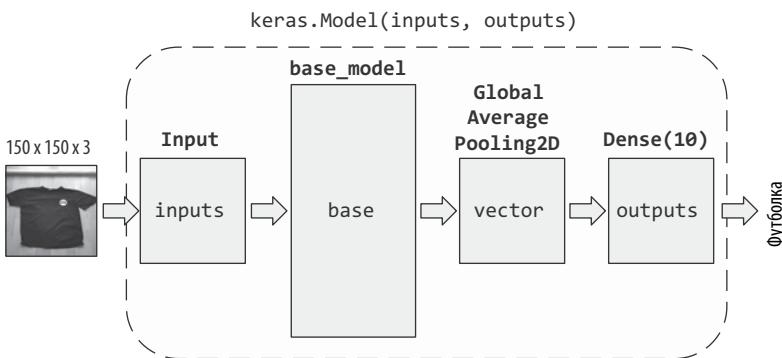
- что модель получит в качестве входных данных (в нашем случае это `inputs`);
- что станет результатом работы модели (в нашем случае `outputs`).

Сделаем шаг назад и еще раз посмотрим на код определения модели, проследив за потоком данных от `inputs` к `outputs` (рис. 7.24).



**Рис. 7.24.** Поток данных: изображение поступает в `inputs`, затем `base_model` преобразует его в `base`, затем объединение преобразует его в `vector`, а затем плотный слой преобразует его в `output`. В конце концов `inputs` и `outputs` передаются в модель Keras

Для простоты понимания мы можем представить каждую строку кода как блок, который получает данные из предыдущего блока, преобразует их и передает следующему (рис. 7.25).



**Рис. 7.25.** Поток данных: каждая строка кода Keras в виде блока

Итак, мы создали модель, которая умеет принимать изображение, получать векторное представление, используя базовую модель, и делать окончательный прогноз с помощью плотного слоя.

Обучим ее.

#### 7.4.4. Обучение модели

Мы определили модель: входные данные, элементы модели (базовая модель, объединяющий слой) и конечный выходной слой.

Теперь нам нужно ее обучить. Для этого нам потребуется *оптимизатор*, который регулирует веса сети, чтобы она лучшеправлялась со своей задачей.

Мы не будем подробно описывать работу оптимизаторов — это выходит за рамки данной книги и не требуется для завершения проекта. Но если вы хотите узнать о них больше, то ознакомьтесь с примечаниями CS231n (<https://cs231n.github.io/neural-networks-3/>). Со списком доступных оптимизаторов вы можете ознакомиться в официальной документации для Keras (<https://keras.io/api/optimizers/>).

Для нашего проекта мы будем использовать алгоритм оптимизации Adam — хороший выбор по умолчанию, и в большинстве случаев его вполне достаточно.

Создадим его:

```
learning_rate = 0.01
optimizer = keras.optimizers.Adam(learning_rate)
```

Adam требует лишь одного параметра: скорости обучения, которая определяет, насколько быстро обучается сеть.

Скорость обучения может существенно влиять на качество нашей сети. Если мы установим слишком высокую скорость, то сеть обучится слишком быстро

и может случайно упустить некоторые важные детали. В этом случае прогностическая производительность окажется не оптимальной. Если мы установим слишком низкую скорость, то обучение сети займет слишком много времени, поэтому процесс обучения окажется крайне неэффективным.

Позже мы скорректируем этот параметр. Пока же установим его равным **0.01** — хорошее начальное значение по умолчанию.

Чтобы обучить модель, оптимизатору необходимо знать, хорошо ли она работает. Для этого он использует функцию потерь, которая уменьшается по мере улучшения сети. Цель оптимизатора — минимизировать эти потери.

Пакет `keras.losses` содержит множество различных оценок потерь. Ниже представлен список наиболее важных из них:

- `BinaryCrossentropy` — для обучения двоичного классификатора;
- `CategoricalCrossentropy` — для обучения модели классификации с несколькими классами;
- `MeanSquaredError` — для обучения регрессионной модели.

Поскольку нам нужно разделить одежду на десять различных классов, мы используем функцию категориальной перекрестной энтропии (кросс-энтропии):

```
loss = keras.losses.CategoricalCrossentropy(from_logits=True)
```

Для данной потери мы указываем только один параметр: `from_logits=True`. Благодаря этому последний уровень нашей сети сможет выводить необработанные оценки (называемые «логитами»), а не вероятности. Официальная документация рекомендует делать это, чтобы сделать численное решение устойчивым ([https://www.tensorflow.org/api\\_docs/python/tf/keras/losses/CategoricalCrossentropy](https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy)).

### ПРИМЕЧАНИЕ

В качестве альтернативы мы могли бы определить последний уровень сети следующим образом: `outputs = keras.layers.Dense(10, activation='softmax')(vector)`

В этом случае мы явно даем сети указание выводить вероятности: многопараметрическая логистическая функция (`softmax`) аналогична сигмоидальной, но для нескольких классов. Тогда вывод больше не будет представлять собой «логиты», поэтому мы можем отказаться от этого параметра:

```
loss = keras.losses.CategoricalCrossentropy()
```

Теперь объединим оптимизатор и потери. Для этого мы используем метод `compile` нашей модели:

```
model.compile(
    optimizer=optimizer,
```

```

    loss=loss,
    metrics=["accuracy"]
)

```

В дополнение к оптимизатору и потерям мы указываем показатели, которые хотим отслеживать во время обучения. Нас интересует достоверность: процент изображений с правильными прогнозами.

Наша модель готова к обучению! Чтобы к нему приступить, используем метод `fit`:

```
model.fit(train_ds, epochs=10, validation_data=val_ds)
```

Мы указываем три параметра:

- `train_ds` — набор данных для обучения;
- `epochs` — количество проходов по обучающим данным;
- `validation_data` — набор данных для проверки.

Одна итерация по всему набору обучающих данных называется *эпохой*. Чем больше итераций мы выполняем, тем лучше сеть усваивает обучающий набор данных.

В какой-то момент она может изучить набор данных настолько хорошо, что начнет переобучаться. Чтобы понять, когда это происходит, нам нужно отслеживать производительность нашей модели на проверочном наборе данных. Вот почему мы указываем параметр `validation_data`.

После начала обучения Keras информирует нас о прогрессе:

```

Train for 96 steps, validate for 11 steps
Epoch 1/10
96/96 [=====] - 22s 227ms/step - loss: 1.2372 -
    accuracy: 0.6734 - val_loss: 0.8453 - val_accuracy: 0.7713
Epoch 2/10
96/96 [=====] - 16s 163ms/step - loss: 0.6023 -
    accuracy: 0.8194 - val_loss: 0.7928 - val_accuracy: 0.7859
...
Epoch 10/10
96/96 [=====] - 16s 165ms/step - loss: 0.0274 -
    accuracy: 0.9961 - val_loss: 0.9342 - val_accuracy: 0.8065

```

Это позволяет нам видеть:

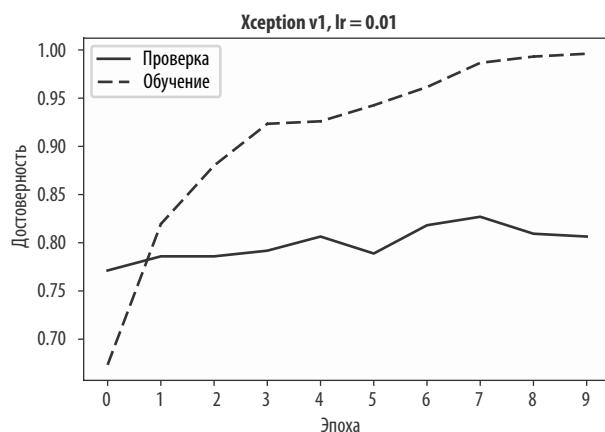
- скорость обучения — сколько времени занимает каждая эпоха;
- достоверность на наборах данных для обучения и проверки. Мы должны следить за достоверностью на проверочном наборе, чтобы точно знать, что модель не начала переобучаться. Например, если достоверность при проверке снижается в течение нескольких эпох, то это признак переобучения;

- потери при обучении и проверке. Нас не интересуют потери — они менее интуитивно понятны, а значения труднее интерпретировать.

### ПРИМЕЧАНИЕ

Ваши результаты, скорее всего, будут другими. Общая прогностическая эффективность модели должна быть аналогичной, но точные цифры будут разными. Применительно к нейронным сетям намного сложнее обеспечить идеальную воспроизведимость, даже при фиксации начальных случайных значений.

Как вы можете видеть, модель быстро достигает достоверности в 99 % на обучающем наборе данных, но оценка при проверке остается в районе 80 % для всех эпох (рис. 7.26).



**Рис. 7.26.** Точность на обучающем и проверочном наборах данных с оценкой после каждой эпохи

Идеальная достоверность на обучающих данных не обязательно означает, что наша модель переобучается, но это верный признак того, что следует скорректировать параметр скорости обучения. Ранее мы упоминали, что это важный параметр, поэтому настроим его прямо сейчас.

### Упражнение 7.1

Обучение с переносом представляет собой процесс использования предварительно обученной (базовой) модели для преобразования изображения в векторное представление, а затем обучения другой модели поверх него.

- Да.
- Нет.

### 7.4.5. Настройка скорости обучения

Мы начали со скорости обучения 0,01. Это хорошая отправная точка, но не обязательно лучшая. Мы увидели, что наша модель обучается слишком быстро и через несколько эпох дает прогноз со 100%-ной точностью на обучающем наборе.

Поэкспериментируем и попробуем другие значения этого параметра.

Для начала, чтобы упростить задачу, мы должны переместить логику создания модели в отдельную функцию. Она принимает скорость обучения в качестве параметра (листинг 7.1).

#### Листинг 7.1. Функция для создания модели

```
def make_model(learning_rate):
    base_model = Xception(
        weights='imagenet',
        input_shape=(150, 150, 3),
        include_top=False
    )

    base_model.trainable = False

    inputs = keras.Input(shape=(150, 150, 3))

    base = base_model(inputs, training=False)
    vector = keras.layers.GlobalAveragePooling2D()(base)

    outputs = keras.layers.Dense(10)(vector)

    model = keras.Model(inputs, outputs)

    optimizer = keras.optimizers.Adam(learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)

    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=["accuracy"],
    )

    return model
```

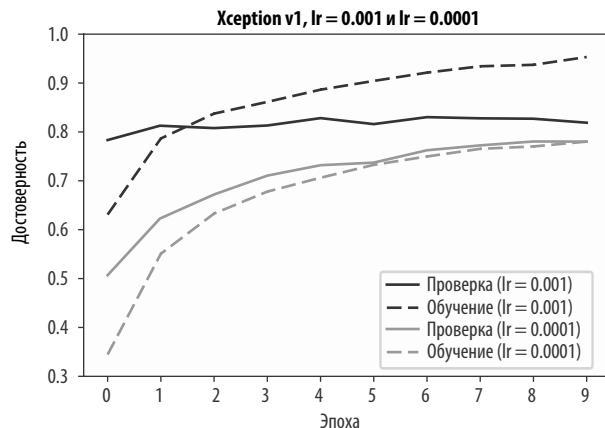
Мы уже пробовали 0,01, так что попробуем 0,001:

```
model = make_model(learning_rate=0.001)
model.fit(train_ds, epochs=10, validation_data=val_ds)
```

Мы также можем использовать и меньшее значение, например 0,0001:

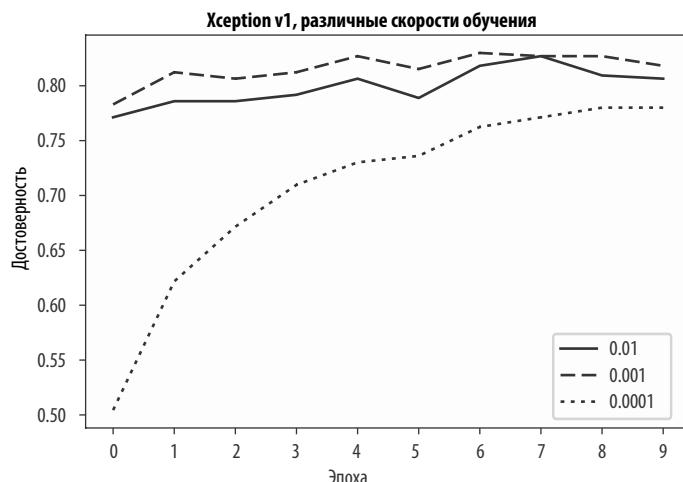
```
model = make_model(learning_rate=0.0001)
model.fit(train_ds, epochs=10, validation_data=val_ds)
```

Как мы видим (рис. 7.27), при 0,001 достоверность при обучении повышается не так быстро, как при 0,01, но при 0,0001 повышается очень медленно. Сеть в этом случае обучается слишком медленно — она *недообучается*.



**Рис. 7.27.** Производительность нашей модели с коэффициентами обучения 0,001 и 0,0001

Если мы посмотрим на показатели при проверке для всех скоростей обучения (рис. 7.28), то увидим, что скорость обучения 0,001 является наилучшей.



**Рис. 7.28.** Достоверность нашей модели на наборе проверки для трех различных скоростей обучения

При скорости обучения 0,001 наилучшая достоверность составляет 83 % (табл. 7.1).

**Таблица 7.1.** Достоверность при различных значениях коэффициента отсева

Скорость обучения	0,01	0,001	0,0001
Достоверность при проверке	82,7 %	83,0 %	78,0 %

#### ПРИМЕЧАНИЕ

Ваши значения могут немного различаться. Кроме того, возможно, что в ваших экспериментах скорость обучения 0,01 даст несколько лучшие результаты, чем 0,001.

Разница между 0,01 и 0,001 несущественна. Но если мы посмотрим на достоверность на обучающих данных, полученную при значении 0,01, то увидим, что переобучение происходит гораздо быстрее. В какой-то момент модель даже достигает точности 100 %. Когда расхождение в производительности между обучающим и проверочным наборами велико, риск переобучения тоже высок. Таким образом, нам стоит предпочесть скорость обучения 0,001.

После завершения обучения необходимо сохранить модель. Далее мы узнаем, как это сделать.

#### 7.4.6. Сохранение модели и контрольная точка

Как только модель обучена, мы можем сохранить ее, используя метод `save_weights`:

```
model.save_weights('xception_v1_model.h5', save_format='h5')
```

Нам нужно указать следующее:

- выходной файл: `'xception_v1_model.h5'`;
- формат: h5, являющийся форматом сохранения двоичных данных.

Возможно, вы заметили, что во время обучения производительность нашей модели на проверочном наборе скачет вверх и вниз. Таким образом, после десяти итераций у нас неизбежно окажется лучшая модель — возможно, лучшая производительность была достигнута на пятой или шестой итерации.

Мы можем сохранять модель после каждой итерации, но она генерирует слишком много данных. И если мы арендует сервер в облаке, то она может быстро занять все доступное пространство.

Вместо этого мы можем сохранить модель, только когда она при проверке показывает результат лучше предыдущего. Например, если предыдущая наилучшая достоверность равна 0,8, но мы улучшили ее до 0,91, то сохраняется модель. В противном случае продолжаем процесс обучения, не прибегая к сохранению.

Данный процесс называется *установкой контрольных точек модели*. В Keras для этого имеется специальный класс: `ModelCheckpoint`. Воспользуемся им:

```
checkpoint = keras.callbacks.ModelCheckpoint(
    "xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5", ← Задает шаблон имени
    save_best_only=True, ← Сохраняет модель, только когда она лучше,
    monitor="val_accuracy" ← Использует чем на предыдущих итерациях
)
)                                            достоверность при
                                                проверке для выбора
                                                наилучшей модели
```

Первый параметр — шаблон для имени файла. Взглянем на него еще раз:

```
"xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5"
```

Он содержит два параметра:

- `{epoch:02d}` заменяется номером эпохи;
- `{val_accuracy:.3f}` заменяется на достоверность при проверке.

Поскольку мы устанавливаем `save_best_only` в значение `True`, `ModelCheckpoint` отслеживает наилучшую достоверность и сохраняет результаты на диск каждый раз, когда она повышается.

Мы реализуем `ModelCheckpoint` как обратный вызов — выполнение чего-либо после завершения каждой эпохи. В этом конкретном случае обратный вызов оценивает модель и сохраняет результат, если достоверность повышается.

Мы можем использовать его, передав в аргумент `callbacks` метода `fit`:

```
model = make_model(learning_rate=0.001) ← Создает новую модель
model.fit(
    train_ds,
    epochs=10,
    validation_data=val_ds,
    callbacks=[checkpoint] ← Определяет список обратных
)
)                                         вызовов, которые будут
                                                использоваться во время обучения
```

После нескольких итераций у нас появятся несколько моделей, сохраненных на диске (рис. 7.29).

		Name	Last Modified	File size
□	0	/		
□	clothing-dataset-small		2 days ago	
□	chapter-07-neural-nets.ipynb		Running seconds ago	549 kB
□	xception_v1_01_0.765.h5		2 minutes ago	84 MB
□	xception_v1_02_0.789.h5		2 minutes ago	84 MB
□	xception_v1_03_0.809.h5		2 minutes ago	84 MB
□	xception_v1_06_0.830.h5		a minute ago	84 MB

**Рис. 7.29.** Поскольку обратный вызов ModelCheckpoint сохраняет модель, только когда она улучшается, у нас есть лишь четыре файла с нашей моделью вместо десяти

Мы научились сохранять лучшую модель. Теперь улучшим ее, добавив в сеть больше слоев.

#### 7.4.7. Добавление дополнительных слоев

Ранее мы обучали модель с одним плотным слоем:

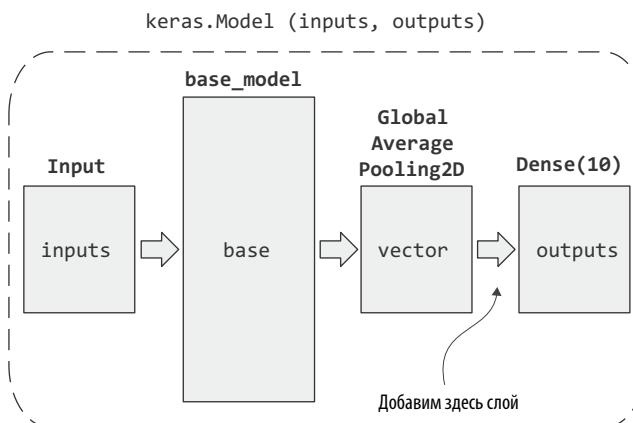
```
inputs = keras.Input(shape=(150, 150, 3))

base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

outputs = keras.layers.Dense(10)(vector)

model = keras.Model(inputs, outputs)
```

У нас нет ограничения в один слой, поэтому добавим еще один между базовой моделью и последним слоем с прогнозами (рис. 7.30).



**Рис. 7.30.** Мы добавляем еще один плотный слой между векторным представлением и выводом

Например, мы можем добавить плотный слой размером 100:

```
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

inner = keras.layers.Dense(100, activation='relu')(vector) ← Добавляет еще
                                                               один плотный слой
                                                               размером 100

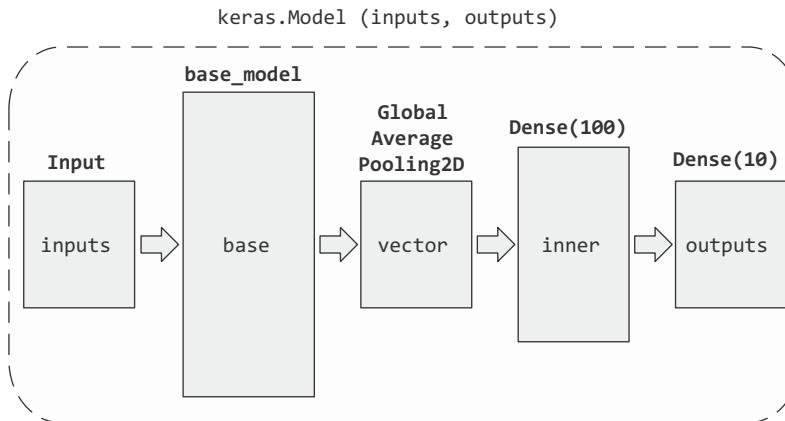
outputs = keras.layers.Dense(10)(inner) ← Вместо подключения выходов к вектору
model = keras.Model(inputs, outputs)    подключает его к внутреннему
```

### ПРИМЕЧАНИЕ

У нас не было особой причины для выбора числа 100 в качестве размера внутреннего плотного слоя. Это просто параметр: как и в случае со скоростью обучения, мы можем попробовать разные значения и выяснить, какое из них приводит к лучшей производительности при проверке. В текущей главе мы не будем экспериментировать с изменением размера внутреннего слоя, но вы можете сделать это самостоятельно.

Таким образом, мы добавили слой между базовой моделью и выходными данными (рис. 7.31). Еще раз взглянем на строку с новым плотным слоем:

```
inner = keras.layers.Dense(100, activation='relu')(vector)
```



**Рис. 7.31.** Новый слой `inner` добавлен между `vector` и `outputs`

Здесь мы устанавливаем параметру `activation` значение `relu`.

Вспомните, что мы получаем нейронную сеть, объединяя несколько логистических регрессий. В логистической регрессии сигмоидальная функция используется для преобразования необработанной оценки в вероятность.

Но для внутренних слоев нам не нужны вероятности, и мы можем заменить сигмоидальную функцию другими. Такие функции называются *функциями активации*. ReLU (Rectified Linear Unit, блок линейной ректификации) является одной из них и для внутренних слоев подходит гораздо больше.

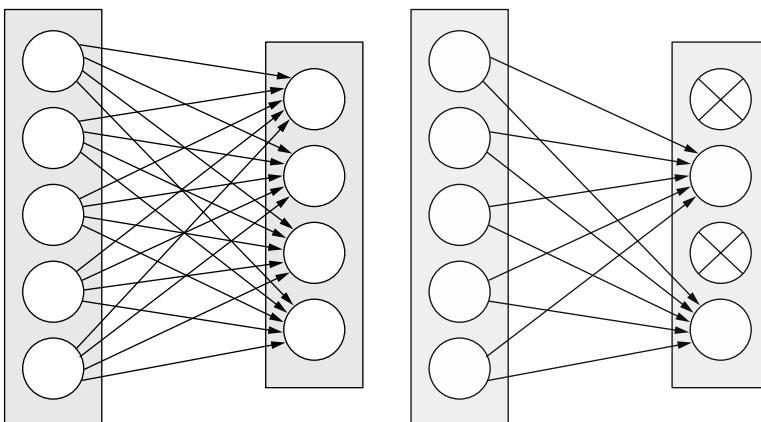
Сигмоидальная функция подвержена проблеме исчезающего градиента, что делает невозможным обучение глубоких нейронных сетей. ReLU решает данную проблему. Больше информации об этой проблеме и о функциях активации в целом можно найти в примечаниях CS231n (<https://cs231n.github.io/neural-networks-1/>).

С новым слоем наши шансы на переобучение значительно возрастают. Чтобы его избежать, нам требуется добавить в нашу модель регуляризацию. Далее мы выясним, как это сделать.

#### 7.4.8. Регуляризация и отсев

*Отсев* — специальная техника борьбы с переобучением в нейронных сетях. Основная идея отсева заключается в «замораживании» части плотного слоя во время обучения. На каждой итерации «замороженная» часть выбирается случайным образом. Обучается только оставшаяся часть, а «замороженная» вообще не меняется.

Если некоторые части сети игнорируются, то вероятность того, что модель в целом окажется переобучена, меньше. Когда сеть просматривает пакет изображений, «замороженная» часть слоя не видит эти данные, поскольку она отключена. Таким образом, сети становится сложнее запоминать изображения (рис. 7.32).



А. Два плотных слоя без отсева

Б. Два плотных слоя с отсевом

**Рис. 7.32.** При отсеве соединения с «замороженными» узлами выпадают

Для каждого пакета часть, подлежащая «заморозке», выбирается случайным образом, поэтому сеть учится извлекать шаблоны из неполной информации, что делает ее более надежной и снижает вероятность переобучения.

Мы можем контролировать частоту отсева — долю элементов в слое, подлежащую «заморозке» на каждом шаге.

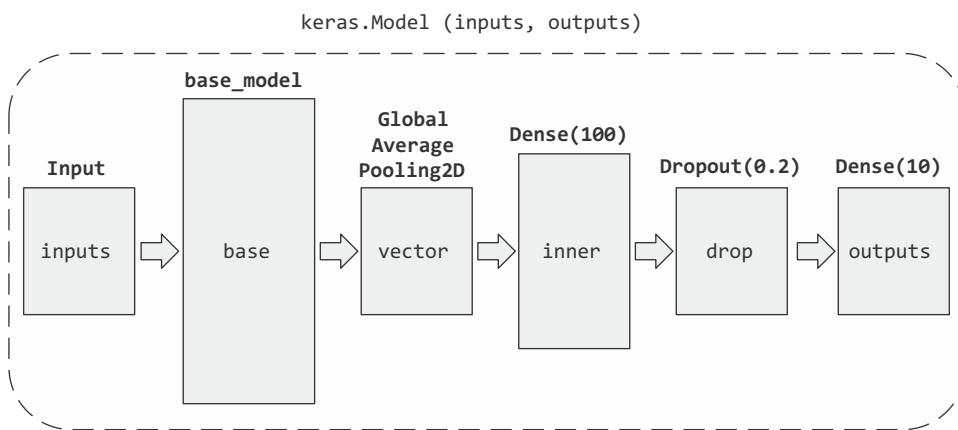
Чтобы сделать это в Keras, мы добавляем слой `Dropout` после первого слоя `Dense` и настраиваем частоту отсева:

```
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

inner = keras.layers.Dense(100, activation='relu')(vector)
drop = keras.layers.Dropout(0.2)(inner)
outputs = keras.layers.Dense(10)(drop)

model = keras.Model(inputs, outputs)
```

Таким образом, мы добавляем еще один блок к нашей сети — блок отсева (рис. 7.33).



**Рис. 7.33.** `Dropout` — это еще один блок между слоями `inner` и `outputs`

Обучим получившуюся модель. Чтобы упростить задачу, сначала стоит обновить функцию `make_model` и добавить туда еще один параметр для управления частотой отсева (листинг 7.2).

#### Листинг 7.2. Функция для создания модели с отсевом

```
def make_model(learning_rate, drop_rate):
    base_model = Xception(
```

## 310 Глава 7. Нейронные сети и глубокое обучение

```
weights='imagenet',
input_shape=(150, 150, 3),
include_top=False
)

base_model.trainable = False

inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vector = keras.layers.GlobalAveragePooling2D()(base)

inner = keras.layers.Dense(100, activation='relu')(vector)
drop = keras.layers.Dropout(droprate)(inner)

outputs = keras.layers.Dense(10)(drop)

model = keras.Model(inputs, outputs)

optimizer = keras.optimizers.Adam(learning_rate)
loss = keras.losses.CategoricalCrossentropy(from_logits=True)

model.compile(
    optimizer=optimizer,
    loss=loss,
    metrics=["accuracy"],
)
return model
```

Попробуем четыре разных значения для `droprate`, чтобы увидеть, как меняется производительность нашей модели:

- **0.0** — ничего не «замораживается», так что это эквивалентно тому, чтобы вообще не включать слой отсева;
- **0.2** — «замораживается» только 20 % слоя;
- **0.5** — половина слоя «замораживается»;
- **0.8** — большая часть слоя (80 %) «замораживается».

При отсеве обучение модели занимает больше времени: на каждом шаге обучается только часть сети, поэтому нам придется сделать больше шагов в целом. Это означает, что нам придется увеличить количество эпох при обучении.

Итак, обучим модель:

```
model = make_model(learning_rate=0.001, droprate=0.0) ←
model.fit(train_ds, epochs=30, validation_data=val_ds) ←
                                                | Изменяет отсев, чтобы
                                                | поэкспериментировать
                                                | с различными значениями
                                                |
                                                | Обучает модель большее
                                                | количество эпох, чем раньше
```

Когда процесс завершится, повторите его для других значений `droprate`, скопировав код в другую ячейку и изменив значение на 0,2, 0,5 и 0,8.

Из результатов на проверочном наборе данных мы видим, что нет существенной разницы между 0,0, 0,2 и 0,5. Однако с 0,8 ситуация хуже — сети стало действительно трудно чему-либо научиться (рис. 7.34).



**Рис. 7.34.** Достоверность на проверочном наборе аналогична для коэффициентов отсева 0,0, 0,2 и 0,5. Однако для 0,8 она хуже

Наилучшая достоверность, которой мы смогли достичь, составляет 84,5 % при коэффициенте отсева 0,5 (табл. 7.2).

**Таблица 7.2.** Достоверность при различных значениях коэффициента отсева

Частота отсева	0,0	0,2	0,5	0,8
Достоверность при проверке	84,2 %	84,2 %	84,5 %	82,4 %

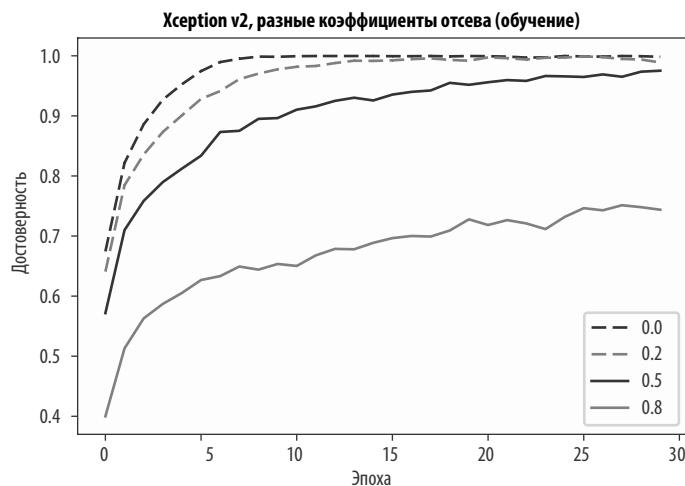
#### ПРИМЕЧАНИЕ

У вас могут получаться другие результаты, и вполне возможно, что другое значение коэффициента отсева даст лучшую достоверность.

В подобных случаях, когда нет видимой разницы между достоверностями на проверочном наборе, полезно принять в расчет достоверность на обучающем (рис. 7.35).

Без отсева модель быстро запоминает весь обучающий набор данных и через десять эпох становится достоверной на 99,9 %.

При коэффициенте отсева 0,2 требуется больше времени для переобучения на обучающем наборе данных, тогда как при 0,5 модель не достигает идеальной достоверности даже после 30 итераций. Установив коэффициент равным 0,8, мы существенно затрудняем обучение сети, поэтому достоверность невелика даже на обучающем наборе данных.



**Рис. 7.35.** При коэффициенте отсева 0,0 сеть быстро переобучается, в то время как коэффициент 0,8 существенно затрудняет обучение

Мы видим, что при коэффициенте отсева 0,5 сеть уже не переобучается настолько быстро, сохраняя при этом тот же уровень достоверности на проверочном наборе, что и при 0,0 и 0,2. Таким образом, следует предпочесть модель, которую мы обучили на коэффициенте отсева 0,5.

Добавив еще один слой и отсев, мы увеличили достоверность с 83 до 84 %. Даже при том, что прирост незначителен для данного конкретного случая, отсев служит эффективным инструментом для борьбы с переобучением, и мы должны использовать его при усложнении наших моделей.

В дополнение к отсеву мы можем применять и другие способы борьбы с переобучением. Например, сгенерировать больше данных. В следующем подразделе мы посмотрим, как это сделать.

### Упражнение 7.2

Что мы делаем при отсеве?

- А. Полностью удаляем часть модели.
- Б. «Замораживаем» случайную часть модели, чтобы она не обновлялась в течение одной итерации обучения.
- В. «Замораживаем» случайную часть модели, чтобы она не использовалась в течение всего процесса обучения.

### 7.4.9. Расширение данных

Получать большое количество данных — всегда хорошо. Обычно это лучшее, что мы можем сделать для улучшения качества нашей модели. К сожалению, не всегда есть возможность получить больший объем данных.

Однако в случае изображений мы можем генерировать больше данных из уже существующих. Например:

- отразить изображение по вертикали и горизонтали;
- повернуть изображение;
- немного увеличить или уменьшить масштаб;
- изменить изображение другими способами.

Процесс генерации дополнительных данных из существующего набора называется *расширением данных* (рис. 7.36).



**Рис. 7.36.** Мы можем генерировать больше обучающих данных, изменения существующие изображения

Самый простой способ создать новое изображение из существующего — отразить его по горизонтали, вертикали или сделать и то и другое сразу (рис. 7.37).



**Рис. 7.37.** Отражение изображения по горизонтали и вертикали

В нашем случае отражение по горизонтали может не иметь особого смысла, а вот отражение по вертикали выглядит полезным.

#### ПРИМЕЧАНИЕ

Информацию о том, как генерируются эти изображения, можно найти в блокноте 07-augmentations.ipynb в репозитории GitHub для этой книги.

Поворот — еще один метод изменения изображений, который можно использовать: новое изображение можно создать, повернув существующее на какое-то количество градусов (рис. 7.38).



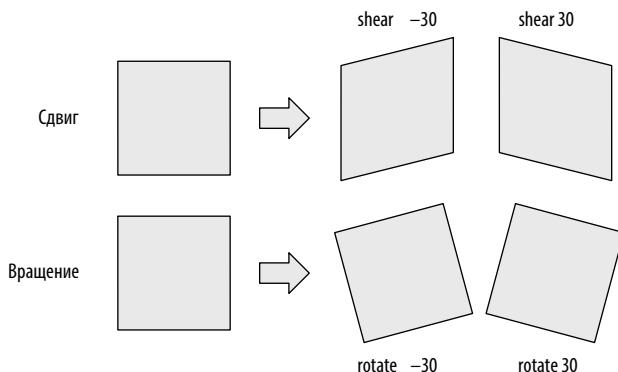
**Рис. 7.38.** Поворот изображения. Если угол поворота отрицательный, то изображение поворачивается против часовой стрелки

Сдвиг — еще одна возможная трансформация. Мы искажаем изображение, «потянув» за одну из сторон. Когда сдвиг положительный, мы тянем правую сторону вниз, а когда отрицательный — вверх (рис. 7.39).



**Рис. 7.39.** Трансформация с помощью сдвига.  
Мы тянем изображение вверх или вниз за его правую сторону

На первый взгляд, поворот и сдвиг могут выглядеть похожими, однако на самом деле это не так. Сдвиг изменяет геометрическую форму изображения, а вращение этого не делает: изображение просто поворачивается (рис. 7.40).



**Рис. 7.40.** Сдвиг изменяет геометрическую форму изображения, вытягивая его, так что квадрат становится параллелограммом. Вращение не изменяет форму, поэтому квадрат остается квадратом

Далее мы можем сместить изображение по горизонтали (рис. 7.41) или по вертикали (рис. 7.42).



**Рис. 7.41.** Смещение изображения по горизонтали.

Положительные значения смещают изображение влево, в то время как отрицательные — вправо



**Рис. 7.42.** Смещение изображения по вертикали. Положительные значения смещают изображение вверх, в то время как отрицательные — вниз

Наконец, мы можем увеличивать или уменьшать масштаб изображения (рис. 7.43).



**Рис. 7.43.** Увеличение или уменьшение масштаба. Когда коэффициент масштабирования меньше 1, мы увеличиваем изображение. Если оно больше 1, то уменьшаем

Более того, при расширении объема данных мы можем комбинировать несколько методов. Например, мы можем взять изображение, отразить его по горизонтали, уменьшить масштаб, а затем повернуть.

Применяя различные изменения к одному и тому же изображению, мы можем сгенерировать множество новых (рис. 7.44).



**Рис. 7.44.** Десять новых изображений, созданных на основе одного

Keras предоставляет встроенный способ расширения набора данных. Он основан на `ImageDataGenerator`, который мы уже использовали для чтения изображений.

Генератор принимает множество аргументов. Ранее мы использовали только `preprocessing_function`, которая необходима для предварительной обработки изображений. Но есть и другие, и многие из них отвечают за приращение набора данных.

Например, мы можем создать новый генератор:

```
train_gen = ImageDataGenerator(
    rotation_range=30,
    width_shift_range=30.0,
    height_shift_range=30.0,
    shear_range=10.0,
    zoom_range=0.2,
    horizontal_flip=True,
    vertical_flip=False,
    preprocessing_function=preprocess_input
)
```

Подробнее рассмотрим его параметры:

- `rotation_range=30` — повернуть изображение на произвольный градус от  $-30$  до  $30$ ;
- `width_shift_range=30` — сместить изображение по горизонтали на величину от  $-30$  до  $30$  пикселей;
- `height_shift_range=30` — сместить изображение по вертикали на величину от  $-30$  до  $30$  пикселей;
- `shear_range=10` — применить сдвиг на значение от  $-10$  до  $10$  (также в пикселях);

- `zoom_range=0.2` — применить масштабирование, используя коэффициент масштабирования от 0,8 до 1,2 ( $1 - 0,2$  и  $1 + 0,2$ );
- `horizontal_flip=True` — случайно отразить изображение по горизонтали;
- `vertical_flip=False` — не отражать изображение вертикально.

Для нашего проекта возьмем небольшой набор этих изменений:

```
train_gen = ImageDataGenerator(
    shear_range=10.0,
    zoom_range=0.1,
    horizontal_flip=True,
    preprocessing_function=preprocess_input,
)
```

Далее мы используем генератор так же, как и ранее:

```
train_ds = train_gen.flow_from_directory(
    "clothing-dataset-small/train",
    target_size=(150, 150),
    batch_size=32,
)
```

Методы расширения следует применить только к обучающим данным. Мы не будем использовать их для проверки: нам требуется последовательность в оценке и возможность сравнивать модель, обученную на расширенном наборе данных, с моделью, обученной без дополнений.

Следовательно, мы загружаем проверочный набор данных, используя точно такой же код, как и раньше:

```
validation_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

val_ds = validation_gen.flow_from_directory(
    "clothing-dataset-small/validation",
    target_size=image_size,
    batch_size=batch_size,
)
```

Теперь мы готовы обучить новую модель:

```
model = make_model(learning_rate=0.001, droprate=0.2)
model.fit(train_ds, epochs=50, validation_data=val_ds)
```

## **ПРИМЕЧАНИЕ**

Для краткости мы опускаем здесь код для сохранения контрольных точек модели. Добавьте его самостоятельно, если хотите сохранить лучшую модель.

Чтобы обучить эту модель, нам потребуется больше эпох, чем раньше. Расширение данных также служит стратегией регуляризации. Вместо того чтобы тренироваться на одном и том же изображении снова и снова, сеть изучает разные варианты одного и того же изображения в каждой эпохе. Это затрудняет запоминание данных и снижает вероятность переобучения.

После обучения модели нам удалось повысить точность на 1 %, с 84 до 85 %.

Само по себе такое улучшение не является значительным. Но мы много экспериментировали, и нам удалось сделать это относительно быстро, поскольку мы использовали небольшие изображения размером  $150 \times 150$ . Теперь мы можем применить все изученное к более крупным изображениям.

### Упражнение 7.3

Почему расширение данных помогает бороться с переобучением?

- А. Модель не видит одни и те же изображения снова и снова.
- Б. Оно добавляет в набор данных разнообразие: повороты и другие преобразования изображений.
- В. Генерируются примеры изображений, которые могут существовать, но модель не увидела бы их в ином случае.
- Г. Все вышеперечисленное.

### 7.4.10. Обучение более крупной модели

Даже людям бывает сложно понять, какой предмет изображен на маленькой картинке размером  $150 \times 150$ . Это не менее сложно и для компьютера: трудно разглядеть важные детали, поэтому модель может путать брюки и шорты или футболки и майки.

При увеличении размера изображений с  $150 \times 150$  до  $299 \times 299$  сети будет легче рассмотреть детали и, следовательно, достичь большей достоверности.

#### ПРИМЕЧАНИЕ

Обучение модели на больших изображениях занимает примерно в четыре раза больше времени, чем на маленьких. Если у вас нет доступа к компьютеру с графическим процессором, то вам необязательно запускать код в этом подразделе. Концептуально процесс тот же, и единственная разница заключается в размере входных данных.

Прежде всего изменим нашу функцию создания модели. Для этого необходимо скорректировать код `make_model` (представленный в листинге 7.2) в двух местах:

- аргумент `input_shape` для `Xception`;
- аргумент `C` для ввода.

В обоих случаях нам нужно заменить `(150, 150, 3)` на `(299, 299, 3)`.

Далее настроим параметр `target_size` для обучающего и проверочного генераторов. Мы заменим `(150, 150)` на `(299, 299)`, а все остальное оставим без изменений.

Вот теперь мы готовы обучить модель!

```
model = make_model(learning_rate=0.001, droprate=0.2)
model.fit(train_ds, epochs=20, validation_data=val_ds)
```

#### ПРИМЕЧАНИЕ

Чтобы сохранить модель, включите в код добавление контрольных точек.

Данная модель обеспечивает достоверность на проверочных данных около 89 %. Это уже значительное улучшение по сравнению с предыдущей моделью.

Мы обучили модель, поэтому настало время ее использовать.

## 7.5. ИСПОЛЬЗОВАНИЕ МОДЕЛИ

Ранее мы обучили несколько моделей. Лучшая из них — обученная на больших изображениях модель с достоверностью 89 %. Вторая по качеству модель имеет достоверность 85 %.

Используем эти модели для получения прогнозов. Чтобы применить какую-либо модель, ее сначала нужно загрузить.

### 7.5.1. Загрузка модели

Вы можете либо использовать самостоятельно обученную модель, либо скачать модель, которую мы обучили для книги, и задействовать ее.

Для скачивания перейдите в раздел релизов репозитория GitHub книги и найдите *Models for Chapter 7: Deep learning* (рис. 7.45). Или перейдите по URL: <https://github.com/alexeygrigorev/mlbookcamp-code/releases/tag/chapter7-model>.

The screenshot shows a GitHub repository page for 'Models for Chapter 7: Deep learning'. The repository was released by user 'alexeygrigorev' 13 hours ago, with 3 commits to the 'master' branch. The description states: 'Pre-trained models for chapter 7 - detecting types of clothes'. Below this, there is a section titled 'Assets' with 4 items:

Asset	Size
xception_v3_44_0.853.h5	82.2 MB
xception_v4_large_08_0.894.h5	82.2 MB
Source code (zip)	
Source code (tar.gz)	

**Рис. 7.45.** Вы можете скачать модели, которые мы подготовили для этой главы, из репозитория книги на GitHub

Затем скачайте большую модель, обученную на изображениях  $299 \times 299$  (xception\_v4\_large). Чтобы использовать модель, загрузите модель с помощью функции `load_model` из пакета `models`:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

Мы уже использовали как обучающий, так и проверочный набор данных. Процесс обучения завершен, поэтому теперь пришло время оценить модель с помощью тестовых данных.

## 7.5.2. Оценка модели

Чтобы загрузить тестовые данные, мы следуем тому же подходу: используем `ImageDataGenerator`, но укажем тестовый каталог. Сделаем это:

```
test_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

test_ds = test_gen.flow_from_directory(
    "clothing-dataset-small/test",
    shuffle=False,
    target_size=(299, 299), ←
    batch_size=32,           | Используйте (150, 150)
                           | для малой модели
)
```

Оценить модель в Keras можно с помощью простого вызова метода `evaluate`:

```
model.evaluate(test_ds)
```

Он применяет модель ко всем данным в тестовой папке и показывает оценочные показатели потерь и достоверности:

```
12/12 [=====] - 70s 6s/step - loss: 0.2493 -
accuracy: 0.9032
```

Наша модель показывает точность в 90 % на тестовом наборе данных, что сопоставимо с производительностью на проверочном наборе (89 %).

Если мы повторим тот же процесс для малого набора данных, то увидим, что производительность ухудшилась:

```
12/12 [=====] - 15s 1s/step - loss: 0.6931 -
accuracy: 0.8199
```

Достоверность составляет 82 %, тогда как при проверке она равнялась 85 %. Модель показала худшие результаты на тестовом наборе данных.

Это могло произойти из-за случайных колебаний: размер наборов для проверки и теста невелик, всего 300 примеров. Таким образом, модели могло повезти при проверке и не повезти при тестировании.

Однако это может быть и признаком переобучения. Многократно оценив модель на проверочном наборе данных, мы выбрали модель, которой очень повезло. Возможно, ее «удачливость» не поддается обобщению, и именно поэтому модель хуже работает с данными, которые не видела ранее.

Теперь посмотрим, как применять модель к отдельным изображениям, чтобы получать прогнозы.

### 7.5.3. Получение прогнозов

Если мы хотим применить модель к одному изображению, то нам нужно сделать то же самое, что `ImageDataGenerator` выполняет внутри:

- загрузить изображение;
- предварительно обработать его.

Мы уже знаем, как загружать изображения. Для этого мы можем использовать `load_img`:

```
path = 'clothing-dataset-small/test/pants/c8d21106-bbdb-4e8d-83e4-
bf3d14e54c16.jpg'
img = load_img(path, target_size=(299, 299))
```

Это изображение брюк (рис. 7.46).

```
img = load_img(path, target_size=(299, 299))  
img
```



**Рис. 7.46.** Изображение брюк из обучающего набора данных

Далее мы предварительно обрабатываем изображение:

```
x = np.array(img)  
X = np.array([x])  
X = preprocess_input(X)
```

И наконец, получаем прогнозы:

```
pred = model.predict(X)
```

Мы можем увидеть прогноз для изображения, проверив первую строку прогнозов: `pred[0]` (рис. 7.47).

```
pred = model.predict(X)  
pred[0]  
  
array([-2.8609202, -4.234048 , -1.5732546, -1.907885 , 10.247051 ,  
-2.2489133, -4.297381 , 4.43905 , -4.4588056, -3.9616938],  
dtype=float32)
```

**Рис. 7.47.** Прогнозы нашей модели. Это массив из десяти элементов, по одному для каждого класса

Результатом служит массив из десяти элементов, где каждый элемент содержит оценку. Чем она выше, тем больше вероятность того, что изображение будет принадлежать к соответствующему классу.

Чтобы получить элемент с наибольшей оценкой, мы можем использовать метод `argmax`. Он возвращает индекс элемента с наибольшей оценкой (рис. 7.48).

```
pred[0].argmax()
4
```

**Рис. 7.48.** Функция `argmax` возвращает элемент с наибольшей оценкой

Чтобы узнать, какая метка соответствует классу 4, нам нужно получить отображение. Его можно извлечь из генератора данных. Но мы вручную занесем его в словарь:

```
labels = {
    0: 'dress',
    1: 'hat',
    2: 'longsleeve',
    3: 'outwear',
    4: 'pants',
    5: 'shirt',
    6: 'shoes',
    7: 'shorts',
    8: 'skirt',
    9: 't-shirt'
}
```

Чтобы получить метку, просто отыщите ее в словаре:

```
labels[pred[0].argmax()]
```

Мы видим метку «брюки», что правильно. Обратите также внимание, что метка «шорты» имеет высокую положительную оценку: брюки и шорты довольно похожи визуально. Но «брюки» явно выигрывают.

Мы будем использовать этот код в следующей главе, где запустим нашу модель в работу.

## 7.6. СЛЕДУЮЩИЕ ШАГИ

Мы изучили основы, с помощью которых можно подготовить классификационную модель к прогнозированию типа одежды. Мы рассмотрели много материала, но общее количество информации значительно превышает объем, который можно вместить в эту главу. Вы можете подробнее изучить данную тему, выполнив упражнения.

### 7.6.1. Упражнения

- Что касается глубокого обучения, то чем больше у нас данных, тем лучше. Но набор данных, который мы использовали для этого проекта, невелик: мы

обучили нашу модель всего на 3068 изображениях. Чтобы улучшить его, мы можем добавить больше обучающих данных. Вы можете найти больше фотографий одежды в других источниках, например на <https://www.kaggle.com/dqmmonn/zalando-store-crawl>, <https://www.kaggle.com/paramagarwal/fashion-product-images-dataset> или <https://www.kaggle.com/c/imaterialist-fashion-2019-FGVC6>. Попробуйте добавить больше изображений к обучающим данным и посмотрите, повысит ли это достоверность на проверочном наборе данных.

- Расширение данных помогает нам обучать более совершенные модели. В текущей главе мы использовали для этого только самые базовые методы. Вы можете продолжить изучать эту тему и попробовать другие способы изменения изображений. Например, добавьте повороты и смещения и посмотрите, поможет ли это добиться лучшей производительности модели.
- В дополнение к встроенному способу расширения набора данных существуют и специальные библиотеки. Одна из них — Albumentations (<https://github.com/albumentations-team/albumentations>) — содержит много дополнительных алгоритмов обработки изображений. Вы также можете поэкспериментировать с ней и посмотреть, какие методы расширения позволят вам решить нашу задачу.
- Доступно и множество различных предварительно обученных моделей. Мы использовали Xception, но существуют и многие другие. Вы можете попробовать их и выяснить, показывают ли они лучшую производительность. С Keras довольно просто использовать любую другую модель: просто импортируйте из другого пакета. Например, можно попробовать ResNet50 и сравнить ее результаты с результатами Xception. Чтобы получить дополнительную информацию, ознакомьтесь с документацией (<https://keras.io/api/applications/>).

## 7.6.2. Другие проекты

Есть много проектов классификации изображений, над которыми можно поработать. Вот пара примеров.

- Кошки или собаки (<https://www.kaggle.com/c/dogs-vs-cats>).
- Хот-дог или не хот-дог (<https://www.kaggle.com/dansbecker/hot-dog-not-hot-dog>).
- Прогнозирование категории изображений из набора данных Avito (онлайн- объявлений) (<https://www.kaggle.com/c/avito-duplicate-ads-detection>). Обратите внимание, что в этом наборе данных много дубликатов, поэтому будьте осторожны при разделении данных для проверки. Возможно, неплохой идеей будет использовать подготовленное организаторами разделение «обучение/тест» и провести небольшую дополнительную очистку, чтобы убедиться в отсутствии дубликатов.

## РЕЗЮМЕ

- TensorFlow — фреймворк для построения и использования нейронных сетей. Keras — надстройка над TensorFlow, которая упрощает обучение моделей.
- Для обработки изображений требуется особый вид нейронной сети: сверточная нейронная сеть. Она состоит из серии сверточных слоев, за которыми следует серия плотных слоев.
- Сверточные слои в нейронной сети преобразуют изображение в его векторное представление. Это представление содержит высокоуровневые признаки. Плотные слои используют эти признаки для прогнозирования.
- Нам не нужно обучать сверточную нейронную сеть с нуля. Мы можем использовать предварительно обученные модели в ImageNet для классификации общего назначения.
- Обучение с переносом опыта — процесс адаптации предварительно обученной модели к нашей проблеме. Мы сохраняем исходные сверточные слои, а плотные создаем заново. Это значительно сокращает время, необходимое нам для обучения модели.
- Мы используем отсев, чтобы предотвращать переобучение. На каждой итерации «замораживается» случайная часть сети, а для обучения используется только оставшаяся. Это позволяет сети лучше обобщать данные.
- Мы можем создавать дополнительные обучающие данные из существующих изображений, поворачивая их, отражая по вертикали и горизонтали и выполняя другие преобразования. Этот процесс называется расширением данных, и он увеличивает вариативность данных, снижая риск переобучения.

В текущей главе мы обучали сверточную нейронную сеть, предназначенную для классификации изображений одежды. Мы можем сохранять ее, загружать и использовать в блокноте Jupyter. Этого недостаточно, чтобы использовать модель в производстве.

В следующей главе мы покажем, как внедрить модель в производство, а также поговорим о двух способах создания моделей глубокого обучения: TensorFlow Lite в AWS Lambda и TensorFlow Serving в Kubernetes.

## ОТВЕТЫ К УПРАЖНЕНИЯМ

- Упражнение 7.1. Ответ А. Да.
- Упражнение 7.2. Ответ Б. «Замораживаем» случайную часть модели, чтобы она не обновлялась в течение одной итерации обучения.
- Упражнение 7.3. Ответ Г. Все вышеперечисленное.



# *Бессерверное глубокое обучение*

---

## **В этой главе**

- ✓ Запуск моделей с помощью TensorFlow Lite — облегченной среды для применения моделей TensorFlow.
- ✓ Развертывание моделей глубокого обучения с помощью AWS Lambda.
- ✓ Предоставление лямбда-функции в качестве веб-сервиса через API Gateway.

В предыдущей главе мы натренировали модель глубокого обучения классифицировать изображения одежды по категориям. Теперь нам нужно ее развернуть, сделав доступной для других сервисов. Существует множество способов сделать это. Мы уже рассмотрели основы развертывания модели в главе 5, где говорили об использовании Flask, Docker и AWS Elastic Beanstalk для развертывания модели логистической регрессии.

В этой главе мы поговорим о бессерверном подходе к развертыванию моделей и используем AWS Lambda.

## **8.1. БЕССЕРВЕРНО: AWS LAMBDA**

AWS Lambda — сервис от Amazon. Его главный лозунг заключается в том, что вы можете «запускать код, не думая о серверах».

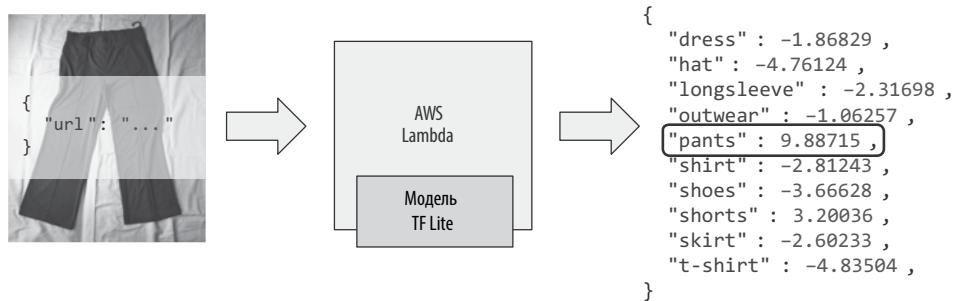
Обещание реализовано в полной мере: в AWS Lambda нам просто нужно загрузить код. Сервис заботится о его запуске и масштабирует его в обе стороны в зависимости от нагрузки.

Кроме того, вы платите только за время, когда функция используется на самом деле. Когда никто не применяет модель и не обращается к нашему сервису, вы ни за что не платите.

В этой главе мы используем AWS Lambda для развертывания модели, которую обучили ранее. Для этого мы также будем использовать TensorFlow Lite — облегченную версию TensorFlow, которая содержит только самые необходимые функции.

Мы хотим создать веб-сервис, который:

- получает URL в запросе;
- загружает изображение из этого URL;
- с помощью TensorFlow Lite применяет модель к изображению и получает прогнозы;
- выдает результаты (рис. 8.1).



**Рис. 8.1.** Обзор сервиса: он получает URL изображения, применяет модель и возвращает прогнозы

Для его создания нам потребуется:

- преобразовать модель из Keras в формат TensorFlow Lite;
- предварительно обработать изображения — изменить их размер и применить функцию предварительной обработки;
- упаковать код в образ Docker и загрузить в ECR (реестр Docker от AWS);
- создать и протестировать лямбда-функцию на AWS;
- сделать лямбда-функцию доступной для всех с помощью AWS API Gateway.

Мы предполагаем, что у вас уже есть учетная запись AWS и вы настроили инструмент AWS CLI. Подробную информацию можно найти в приложении A.

**ПРИМЕЧАНИЕ**

На момент написания книги на AWS Lambda распространяется бесплатный уровень AWS. Это означает, что все эксперименты, описанные в данной главе, можно проводить бесплатно. Условия использования представлены в документации AWS (<https://aws.amazon.com/free/>).

В нашем случае мы используем AWS, но подход также сработает и для других бессерверных платформ.

Код для этой главы доступен в репозитории книги на GitHub (<https://github.com/alexeygrigorev/mlbookcamp-code/>) в папке chapter-08-serverless.

Начнем с обсуждения TensorFlow Lite.

### 8.1.1. TensorFlow Lite

TensorFlow – отличный фреймворк с богатым набором функций. Однако большинство из этих функций не нужны для развертывания модели, а кроме того, занимают много места: после сжатия TensorFlow занимает более 1,5 Гбайт места.

А вот библиотека TensorFlow Lite (сокращенно TF Lite) занимает всего 50 Мбайт. Она оптимизирована для мобильных устройств и содержит только самое необходимое. TF Lite позволяет использовать модель лишь для составления прогнозов и ни для чего другого, включая обучение новых моделей.

Несмотря на то что изначально TF Lite была создана для мобильных устройств, она применима для множества других случаев. Мы можем использовать ее всякий раз, когда у нас есть модель TensorFlow, но мы не можем позволить себе взять весь пакет TensorFlow.

**ПРИМЕЧАНИЕ**

Библиотека TF Lite находится в стадии активной разработки и меняется довольно быстро. Возможно, способ ее установки изменился с момента публикации книги. Актуальные инструкции представлены в официальной документации (<https://www.tensorflow.org/lite/guide/python>).

Установим библиотеку прямо сейчас. Мы можем сделать это с помощью pip:

```
pip install --extra-index-url https://google-coral.github.io/py-repo/  
    tflite_runtime
```

При запуске `pip install` мы добавляем параметр `extra-index-url`. Эта библиотека недоступна в центральном репозитории с пакетами Python, но доступна в другом. Нам нужно указать этот репозиторий.

**ПРИМЕЧАНИЕ**

Для дистрибутивов Linux, не основанных на Debian (таких как CentOS, Fedora или Amazon Linux), библиотека, установленная таким образом, может не работать: вы можете получить сообщение об ошибке при попытке импорта. Если это ваш случай, то вам придется скомпилировать эту библиотеку самостоятельно. Получить инструкции и более подробную информацию можно по ссылке <https://github.com/alexeygrigorev/serverless-deep-learning>. Для macOS и Windows все должно работать ожидаемым образом.

TF Lite использует специальный оптимизированный формат для хранения моделей. Чтобы использовать нашу модель с TF Lite, нам придется преобразовать ее в этот формат. Этим мы сейчас и займемся.

### **8.1.2. Преобразование модели в формат TF Lite**

В предыдущей главе для хранения модели мы использовали формат h5. Он подходит для хранения моделей Keras, но не будет работать с TF Lite. Следовательно, нам нужно преобразовать модель в формат TF-Lite.

Если у вас нет модели из предыдущей главы, то скачайте ее:

```
wget https://github.com/alexeygrigorev/mlbookcamp-code/releases/download/
      chapter7-model/xception_v4_large_08_0.894.h5
```

Теперь создадим простой скрипт для преобразования этой модели — `convert.py`.

Начнем с импорта:

```
import tensorflow as tf
from tensorflow import keras
```

Затем загрузим модель Keras:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

И наконец, конвертируем ее в TF Lite:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)

tflite_model = converter.convert()

with tf.io.gfile.GFile('clothing-model-v4.tflite', 'wb') as f:
    f.write(tflite_model)
```

Запустим код:

```
python convert.py
```

После запуска в нашем каталоге должен появиться файл с именем `clothing-model-v4.tflite`. Теперь мы готовы с помощью модели классифицировать

изображения одежды, чтобы понять, является ли то или иное изображение футболкой, брюками, юбкой или чем-то еще. Однако мы помним, что прежде чем использовать модель для классификации изображений, их нужно предварительно обработать. Сейчас мы узнаем, как это делается.

### 8.1.3. Подготовка изображений

Ранее, при тестировании модели в Keras, мы предварительно обрабатывали каждое изображение с помощью функции `preprocess_input`. Вот как мы импортировали ее в предыдущей главе:

```
from tensorflow.keras.applications.xception import preprocess_input
```

А затем мы применяли эту функцию к изображениям.

Однако использовать ту же функцию при развертывании нашей модели уже не получится. Эта функция — часть пакета TensorFlow, и в TF Lite нет ее эквивалента. Мы не хотим зависеть от TensorFlow только из-за этой простой функции предварительной обработки.

Вместо этого мы используем специальную библиотеку, которая содержит только нужный нам код: `keras_image_helper`. Эта библиотека была написана для того, чтобы упростить объяснения в данной книге. Если вы хотите в подробностях узнать, как выполняется предварительная обработка изображений, то ознакомьтесь с исходным кодом. Он доступен по адресу <https://github.com/alexeygrigorev/keras-image-helper>. Эта библиотека может загружать изображение, изменять его размер и применять другие преобразования предварительной обработки, которые требуются моделям Keras.

Установим ее с помощью pip:

```
pip install keras_image_helper
```

Затем откройте Jupyter и создайте блокнот под названием `chapter-08-model-test`.

Мы начнем с импорта `create_preprocessor` из этой библиотеки:

```
from keras_image_helper import create_preprocessor
```

Функция `create_preprocessor` принимает два аргумента:

- `name` — название модели. Вы можете ознакомиться со списком доступных моделей по адресу <https://keras.io/api/applications/>;
- `target_size` — размер изображения, которое ожидает получить нейронная сеть.

Мы используем модель Xception, которая ожидает изображение размером  $299 \times 299$ . Создадим препроцессор для нашей модели:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Теперь получим изображение брюк (рис. 8.2) и подготовим его:

```
image_url = 'http://bit.ly/mlbookcamp-pants'  
X = preprocessor.from_url(image_url)
```



**Рис. 8.2.** Изображение брюк, которые мы используем для тестирования

Результатом является массив NumPy формы (1, 299, 299, 3):

- это пакет только из одного изображения;
- $299 \times 299$  — это размер изображения;
- в нем три цветовых канала: красный, зеленый и синий.

Мы подготовили изображение и теперь можем с помощью модели классифицировать его. Пришло время узнать, как это делается с помощью TF Lite.

#### **8.1.4. Использование модели TensorFlow Lite**

На предыдущем шаге мы получили массив X и теперь можем с помощью TF Lite классифицировать его.

Сначала импортируем TF Lite:

```
import tensorflow as tf
```

Загрузим ранее преобразованную модель:

```
interpreter = tf.lite.Interpreter(model_path='clothing-model-v4.tflite')  
interpreter.allocate_tensors()
```

Создает интерпретатор TF Lite

Инициализирует интерпретатор с помощью модели

Чтобы использовать модель, нам нужно получить ее входные данные (куда пойдет X) и выходные (откуда мы получим прогнозы):

input_details = interpreter.get_input_details() input_index = input_details[0]['index']	Получает входные данные: часть сети, которая принимает массив X
output_details = interpreter.get_output_details() output_index = output_details[0]['index']	Получает результат: часть сети с окончательными прогнозами

Чтобы применить модель, возьмите подготовленный ранее X и поместите его на вход, вызовите интерпретатор и получите результаты из выходных данных:

```
interpreter.set_tensor(input_index, X)  
interpreter.invoke()  
preds = interpreter.get_tensor(output_index)
```

Помещает X во входные данные

Запускает модель для получения прогнозов

Получает прогнозы из выходных данных

Массив preds содержит прогнозы:

```
array([[-1.8682897, -4.7612453, -2.316984, -1.0625705, 9.887156,  
       -2.8124316, -3.6662838, 3.2003622, -2.6023388, -4.8350453]],  
      dtype=float32)
```

Теперь мы можем проделать с ним то же самое, что и ранее, — присвоить метку каждому элементу массива:

```
labels = [  
    'dress',  
    'hat',  
    'longsleeve',  
    'outwear',  
    'pants',  
    'shirt',  
    'shoes',  
    'shorts',  
    'skirt',  
    't-shirt'  
]  
  
results = dict(zip(labels, preds[0]))
```

Дело сделано! У нас есть прогнозы в переменной `results`:

```
{'dress': -1.8682897,
 'hat': -4.7612453,
 'longsleeve': -2.316984,
 'outwear': -1.0625705,
 'pants': 9.887156,
 'shirt': -2.8124316,
 'shoes': -3.6662838,
 'shorts': 3.2003622,
 'skirt': -2.6023388,
 't-shirt': -4.8350453}
```

Мы видим, что метка `pants` имеет самую высокую оценку, так что это должно быть изображение брюк.

Теперь используем данный код для нашей будущей функции AWS Lambda!

### **8.1.5. Код для лямбда-функции**

В предыдущем подразделе мы уже написали весь код, который нам нужен для лямбда-функции. Сведем все это воедино в одном сценарии — `lambda_function.py`.

Как обычно, начнем с импорта:

```
import tensorflow as tf
from keras_image_helper import create_preprocessor
```

Затем создадим препроцессор:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Далее загрузим модель и получим выходные и входные данные:

```
interpreter = tf.lite.Interpreter(model_path='clothing-model-v4.tflite')
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
input_index = input_details[0]['index']
output_details = interpreter.get_output_details()
output_index = output_details[0]['index']
```

Чтобы сделать код немного чище, мы можем объединить получение прогноза в одну функцию:

```
def predict(X):
    interpreter.set_tensor(input_index, X)
    interpreter.invoke()
    preds = interpreter.get_tensor(output_index)
    return preds[0]
```

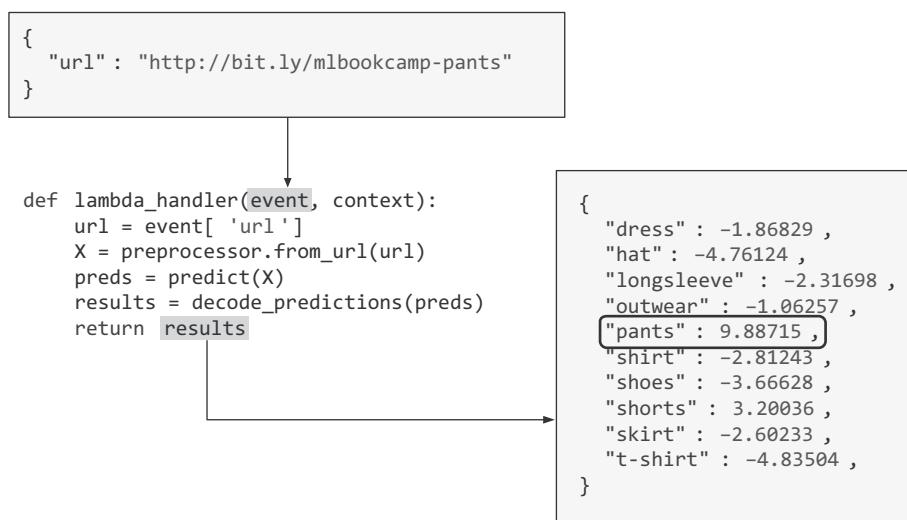
Далее создадим еще одну функцию для подготовки результатов:

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]

def decode_predictions(pred):
    result = {c: float(p) for c, p in zip(labels, pred)}
    return result
```

Наконец соберем все вместе в одной функции — `lambda_handler` — которая является функцией, вызываемой средой AWS Lambda. В ней будет использовано все то, что мы определили ранее:

```
def lambda_handler(event, context):
    url = event['url']
    X = preprocessor.from_url(url)
    preds = predict(X)
    results = decode_predictions(preds)
    return results
```



**Рис. 8.3.** Входные и выходные данные лямбда-функции: входные данные передаются в параметр `event`, а прогнозы возвращаются в качестве выходных данных

В этом случае параметр `event` содержит всю информацию, которую мы передаем лямбда-функции в нашем запросе (рис. 8.3). Параметр `context` обычно не используется.

Мы готовы приступить к тестированию! Для локального теста нам следует поместить код в контейнер Python Docker для AWS Lambda.

## 8.1.6. Подготовка образа Docker

Сначала создадим файл с именем Dockerfile:

```
FROM public.ecr.aws/lambda/python:3.7
RUN pip3 install keras_image_helper --no-cache-dir
RUN pip3 install https://raw.githubusercontent.com/alexeygrigorev/
serverlessdeep-
learning/master/tflite/tflite_runtime-2.2.0-cp37-cp37m-linux_x86_64.whl
--no-cache-dir
COPY clothing-model-v4.tflite clothing-model-v4.tflite
COPY lambda_function.py lambda_function.py
CMD [ "lambda_function.lambda_handler" ]
```

Пройдемся по всем строкам файла. Сначала в ❶ мы используем официальный образ Python 3.7 Docker для Lambda от AWS. Вы можете найти другие доступные образы здесь: <https://gallery.ecr.aws/>. Затем в ❷ мы устанавливаем библиотеку `keras_image_helper`.

Далее в ❸ мы устанавливаем специальную версию TF Lite, которая была скомпилирована для работы с Amazon Linux. Инструкции по установке, которые мы использовали в этой главе ранее, работают только для Ubuntu (и других дистрибутивов на базе Debian), но не для Amazon Linux. Вот почему нам нужна особая версия. Почитать об этом можно здесь: <https://github.com/alexeygrigorev/serverless-deep-learning>.

Затем в ❹ мы копируем модель в образ. После этого она становится его частью. Так модель проще развернуть. Мы могли бы использовать альтернативный подход: модель может быть помещена в S3 и загружена при запуске сценария. Это более сложный метод, но и более гибкий. Для книги мы выбрали более простой подход.

Затем в ❺ мы копируем код лямбда-функции, который подготовили ранее.

Наконец в ❸ мы сообщаем среде `lambda`, что ей необходим файл с именем `lambda_function` и функция `lambda_handler` внутри этой функции. Это и есть подготовленная в предыдущем подразделе функция.

Создадим образ:

```
docker build -t tf-lite-lambda .
```

Далее нам нужно проверить, что лямбда-функция работает. Для этого запустим изображение:

```
docker run --rm -p 8080:8080 tf-lite-lambda
```

Она работает! Можно приступать к тесту.

Мы можем и далее использовать Jupyter Notebook, который создали ранее, или же создать отдельный файл Python с именем `test.py`. Он должен содержать следующее (вы заметите, что содержимое очень похоже на код, который мы написали в главе 5 для тестирования нашего веб-сервиса):

```
import requests
data = { ❶ Подготавливает
         ' запрос
         "url": "http://bit.ly/mlbookcamp-pants"
     }
❷ Указывает URL
url = "http://localhost:8080/2015-03-31/functions/function/invocations" ←

results = requests.post(url, json=data).json() ❸ Отправляет
print(results)                                POST-запрос сервису
```

Сначала мы определяем переменную `data` в ❶ — это наш запрос. Затем мы указываем URL сервиса в ❷ — местоположение функции, развернутой в данный момент. Наконец в ❸ мы используем метод `POST` для отправки запроса и получения обратно прогнозов в переменной `results`.

После запуска мы получаем следующий ответ:

```
{
    "dress": -1.86829,
    "hat": -4.76124,
    "longsleeve": -2.31698,
    "outwear": -1.06257,
    "pants": 9.88715,
    "shirt": -2.81243,
    "shoes": -3.66628,
    "shorts": 3.20036,
    "skirt": -2.60233,
    "t-shirt": -4.83504
}
```

Модель работает!

Мы в шаге от того, чтобы развернуть ее в AWS. Чтобы это сделать, нам нужно опубликовать данный образ в ECR — реестре контейнеров Docker от AWS.

### **8.1.7. Отправка изображения в AWS ECR**

Чтобы опубликовать образ Docker в AWS, нам прежде всего необходимо создать реестр с помощью инструмента AWS CLI:

```
aws ecr create-repository --repository-name lambda-images
```

Команда вернет URL, который выглядит следующим образом:

```
<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/lambda-images
```

Этот URL нам и нужен.

В качестве альтернативы можно создать реестр с помощью консоли AWS.

Как только реестр создан, нужно поместить туда образ. Поскольку реестр принадлежит нашей учетной записи, сначала необходимо аутентифицировать наш клиент Docker. В Linux и macOS это делается следующим образом:

```
$(aws ecr get-login --no-include-email)
```

В Windows запустите `aws ecr get-login --no-include-email`, скопируйте выходные данные, введите их в терминал и выполните вручную.

Теперь воспользуемся URL реестра, чтобы отправить образ в ECR:

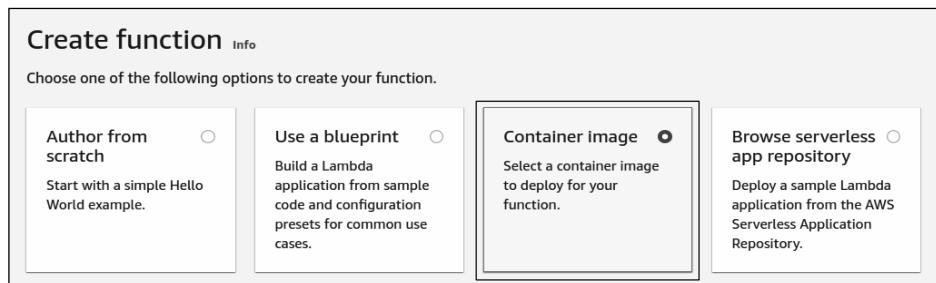
```
REGION=eu-west-1 | Укажите регион и ваш идентификатор
ACCOUNT=XXXXXXXXXXXX | учетной записи AWS
REMOTE_NAME=${ACCOUNT}.dkr.ecr.${REGION}.amazonaws.com/lambda-images:tf-lite-
lambda
docker tag tf-lite-lambda ${REMOTE_NAME}
docker push ${REMOTE_NAME}
```

Образ отправлен, и мы можем использовать его для создания лямбда-функции в AWS.

### **8.1.8. Создание лямбда-функции**

Этот шаг проще выполнить с помощью консоли AWS, поэтому откройте ее, перейдите в раздел сервисов и выберите Lambda.

Далее нажмите кнопку Create Function. Выберите Container Image (рис. 8.4).



**Рис. 8.4.** При создании лямбда-функции выберите Container Image

После этого заполните данные (рис. 8.5).

The screenshot shows the 'Basic information' section of the Lambda function creation wizard. It includes:

- Function name**: Enter a name that describes the purpose of your function. The input field contains "clothes-classification".
- Container image URI**: The location of the container image to use for your function. The input field contains "<ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/lambda-images:tf-lite-lambda".
- Browse images**: A button to browse available container images.

**Рис. 8.5.** Введите имя функции и URI образа контейнера

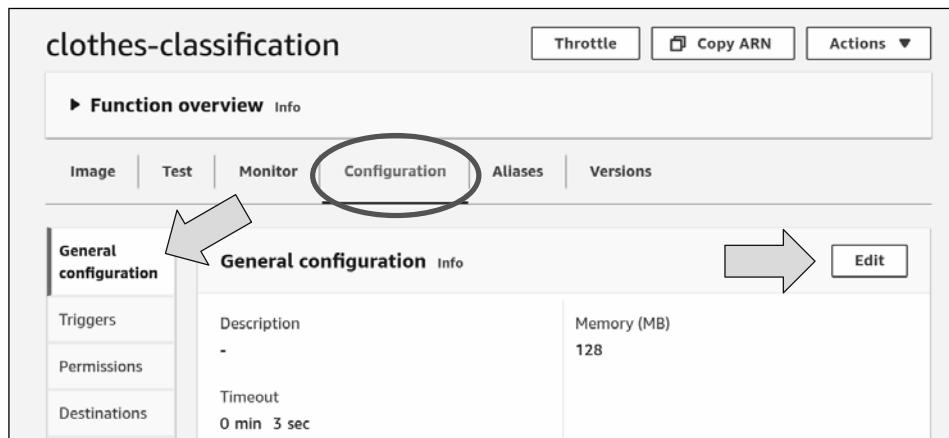
URI образа контейнера должен быть тем образом, который мы создали ранее и отправили в ECR:

<ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/lambda-images:tf-lite-lambda

Вы можете использовать кнопку Browse Images, чтобы его найти (см. рис. 8.5). Оставьте остальное без изменений и нажмите Create Function. Функция готова!

Теперь необходимо предоставить нашей функции больше памяти и дать возможность работать в течение более длительного времени без превышения лимита.

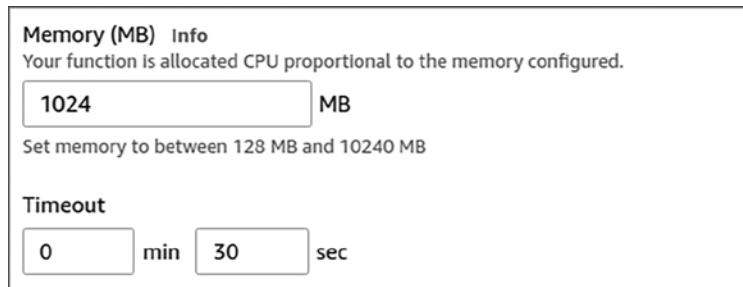
Для этого перейдите на вкладку Configuration, выберите General Configuration, а затем нажмите Edit (рис. 8.6).



**Рис. 8.6.** Настройки лямбда-функции по умолчанию: объема памяти по умолчанию (128 Мбайт) недостаточно, поэтому нужно его увеличить. Для этого нажмите Edit

Настройки по умолчанию не подходят для моделей глубокого обучения. Нам нужно настроить эту функцию и предоставить ей больше оперативной памяти, а также позволить дольше работать.

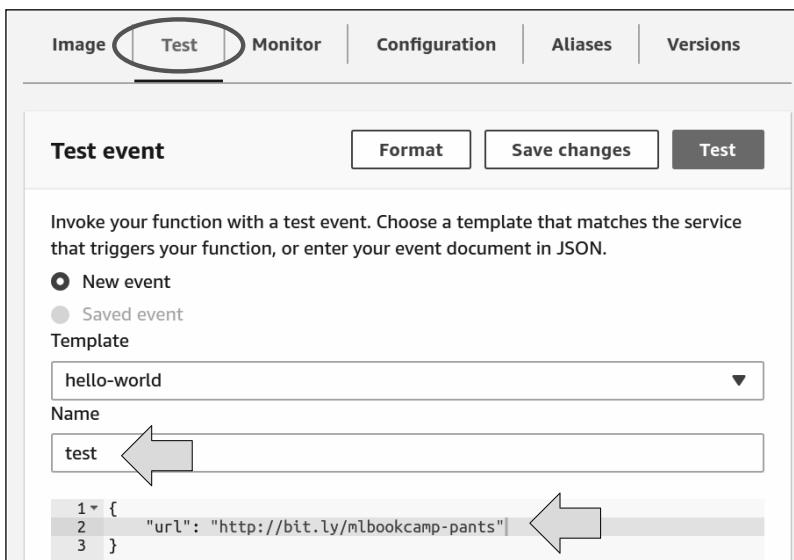
Для этого нажмите кнопку Edit, выделите ей 1024 Мбайт оперативной памяти и установите тайм-аут на 30 секунд (рис. 8.7).



**Рис. 8.7.** Увеличьте объем памяти до 1024 Мбайт и установите лимит времени на 30 секунд

Сохраните.

Все готово! Чтобы протестировать, перейдите на вкладку Test (рис. 8.8).

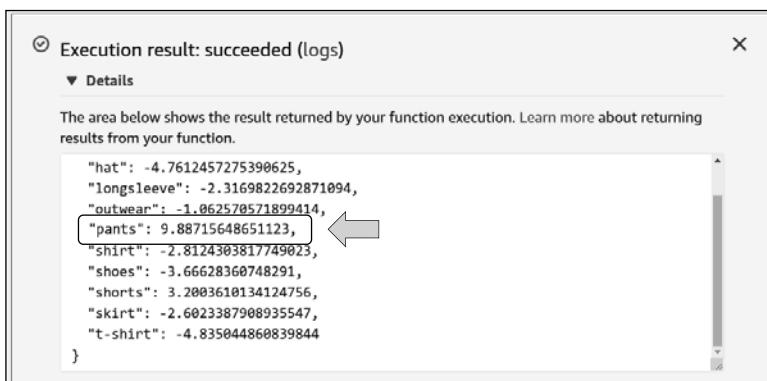


**Рис. 8.8.** Кнопка Test расположена в верхней части экрана. Нажмите на нее, чтобы протестировать функцию

Вам предложат создать тестовое событие. Дайте ему имя (например, `test`) и поместите в тело запроса следующее:

```
{
  "url": "http://bit.ly/mlbookcamp-pants"
}
```

Сохраните ее и снова нажмите кнопку Test. Примерно через 15 секунд вы должны увидеть Execution results: succeeded (рис. 8.9).



**Рис. 8.9.** Прогнозы нашей модели. Прогноз для pants имеет самую высокую оценку

При запуске в первый раз тесту придется извлечь образ из ECR, загрузить в память все библиотеки и выполнить некоторые другие действия для «разминки». Но как только все это будет сделано, последующие вызовы займут уже меньше времени — примерно две секунды для этой модели.

Мы успешно внедрили нашу модель в AWS Lambda, и она заработала!

Кроме того, помните, что вы платите только при вызове функции, поэтому вам не нужно беспокоиться о ее удалении, если она не используется. Да и вообще не нужно беспокоиться об управлении экземплярами EC2 — AWS Lambda все делает за нас.

Эту модель уже можно использовать для многих целей: AWS Lambda хорошо интегрируется со многими другими сервисами AWS. Но если мы захотим использовать ее как веб-сервис и отправлять запросы по протоколу HTTP, то нам потребуется открыть доступ через API Gateway.

Далее мы узнаем, как это сделать.

### 8.1.9. Создание API Gateway

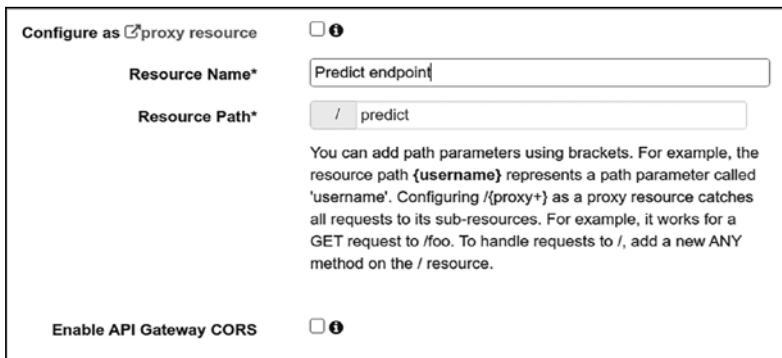
В консоли AWS найдите сервис API Gateway. Создайте новый API: выберите REST API и нажмите Build.

Затем выберите New API и назовите его `clothes-classification` (рис. 8.10). Нажмите Create API.

The screenshot shows the 'Create new API' dialog. At the top, there's a note: 'In Amazon API Gateway, a REST API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.' Below this are three radio buttons: 'New API' (selected), 'Import from Swagger or Open API 3', and 'Example API'. The next section is titled 'Settings' with the sub-instruction: 'Choose a friendly name and description for your API.' It contains three input fields: 'API name\*' with the value 'clothes-classification', 'Description' with the value 'expose lambda as a web service', and 'Endpoint Type' set to 'Regional'. A small info icon is located next to the endpoint type dropdown.

**Рис. 8.10.** Создание нового REST API Gateway в AWS

Далее нажмите кнопку Action и выберите Resource. Затем создайте ресурс predict (рис. 8.11).



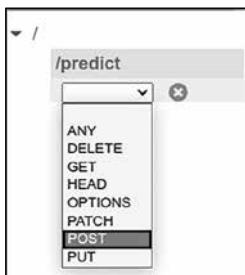
**Рис. 8.11.** Создание ресурса predict

#### ПРИМЕЧАНИЕ

Имя predict не соответствует стандартам именования REST: обычно ресурсы должны быть существительными. Однако обычно конечные точки для прогнозов именуются predict; вот почему мы не следуем здесь соглашению REST.

Когда создание ресурса завершится, создайте для него метод POST (рис. 8.12).

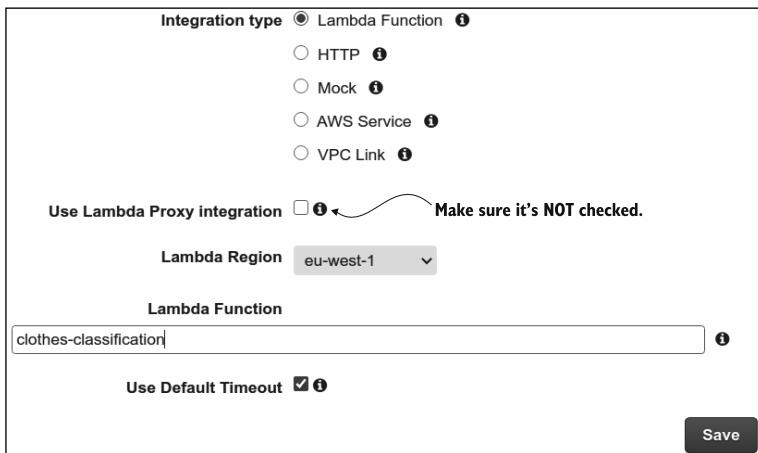
1. Нажмите Predict.
2. Нажмите Actions.
3. Выберите Create Method.
4. Выберите POST из списка.
5. Нажмите кнопку.



**Рис. 8.12.** Создайте метод POST для ресурса predict

Мы почти на месте!

Теперь выберите Lambda Function в качестве типа интеграции и введите имя вашей лямбда-функции (рис. 8.13).

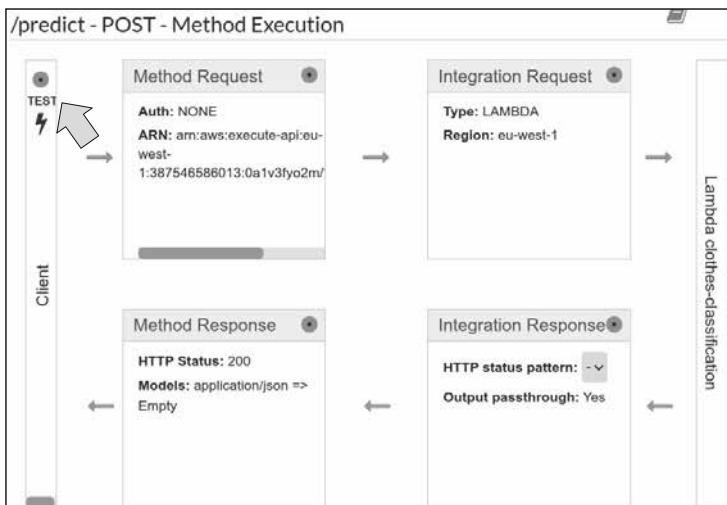


**Рис. 8.13.** Настройка действия POST для ресурса predict. Убедитесь, что флагок прокси-интеграции не установлен

#### ПРИМЕЧАНИЕ

Убедитесь, что вы не используете прокси-интеграцию — этот флагок должен оставаться снятым. Если использовать эту опцию, то API Gateway добавит к запросу дополнительную информацию, и нам придется менять лямбда-функцию.

Проделав все это, мы должны увидеть интеграцию (рис. 8.14).



**Рис. 8.14.** Развёртывание API

Проведем тест. Нажмите TEST и поместите в тело запроса тот же запрос, что и ранее:

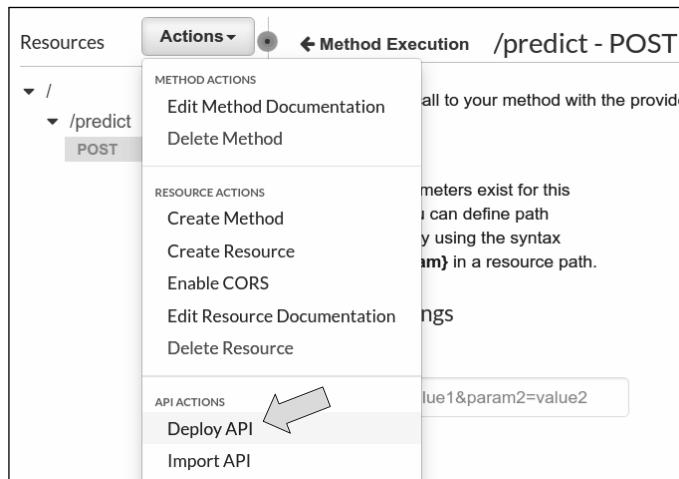
```
{
  "url": "http://bit.ly/mlbookcamp-pants"
}
```

Ответ тот же: прогнозируемый класс — pants (рис. 8.15).

```
Request: /predict
Status: 200
Latency: 5327 ms
Response Body
{
  "dress": -1.8682900667190552,
  "hat": -4.7612457275390625,
  "longsleeve": -2.3169822692871094,
  "outwear": -1.062570571899414,
  "pants": 9.88715648651123, ←
  "shirt": -2.8124303817749023,
  "shoes": -3.66628360748291,
  "shorts": 3.2003610134124756,
  "skirt": -2.6023387988935547,
  "t-shirt": -4.835044860839844
}
```

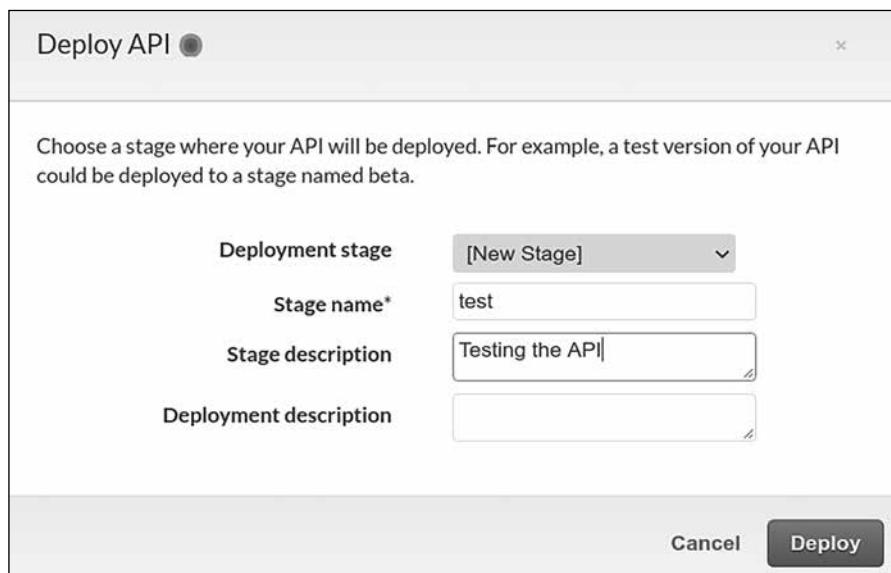
**Рис. 8.15.** Ответ от лямбда-функции. Категория брюк получила самую высокую оценку

Чтобы использовать ее извне, нам потребуется развернуть API. Выберите Deploy API из списка действий (рис. 8.16).



**Рис. 8.16.** Функция clothes-classification теперь подключена к методу POST ресурса predict в API Gateway. Кнопка TEST помогает проверить, работает ли соединение с лямбдой

Далее создайте новый тест New Stage (рис. 8.17).



**Рис. 8.17.** Настройка этапа для API

Нажав кнопку Deploy, мы развертываем API. Теперь найдите поле Invoke URL. Оно должно выглядеть примерно так: <https://0a1v3fy02m.execute-api.eu-west-1.amazonaws.com/test>.

Все, что нам теперь нужно сделать для вызова лямбда-функции, — добавить /predict в конец этого URL.

Возьмем сценарий `test.py`, который мы создали ранее, и заменим URL в нем:

```
import requests

data = {
    "url": "http://bit.ly/mlbookcamp-pants"
}

url = "https://0a1v3fy02m.execute-api.eu-west-1.amazonaws.com/test/predict"

results = requests.post(url, json=data).json()

print(results)
```

Запустите сценарий:

```
python test.py
```

Ответ такой же, как и ранее:

```
{
  "dress": -1.86829,
  "hat": -4.76124,
  "longsleeve": -2.31698,
  "outwear": -1.06257,
  "pants": 9.88715,
  "shirt": -2.81243,
  "shoes": -3.66628,
  "shorts": 3.20036,
  "skirt": -2.60233,
  "t-shirt": -4.83504
}
```

Теперь наша модель доступна с помощью веб-сервиса, который мы можем использовать откуда угодно.

## 8.2. СЛЕДУЮЩИЕ ШАГИ

### 8.2.1. Упражнения

Вы можете более подробно изучить темы развертывания бессерверной модели с помощью компонентов, описанных ниже.

- AWS Lambda — не единственная бессерверная среда. Вы также можете поэкспериментировать с облачными функциями в Google Cloud и функциями Azure в Azure.
- SAM (модель бессерверного приложения) — это инструмент от AWS, упрощающий процесс создания лямбда-функций AWS (<https://aws.amazon.com/serverless/sam/>). Вы можете использовать его для другой реализации проекта из этой главы.
- Serverless (<https://www.serverless.com/>) — это фреймворк, похожий на SAM. Он не привязан к AWS и работает для других облачных провайдеров. Вы можете поэкспериментировать с ним и развернуть проект из данной главы.

### 8.2.2. Другие проекты

Вы можете поработать над многими другими проектами.

- AWS Lambda — удобная платформа для размещения моделей машинного обучения. В этой главе мы развернули модель глубокого обучения. Вы можете самостоятельно поэкспериментировать с ней и использовать модели, которые мы обучали в предыдущих главах, а также модели, разработанные вами в рамках упражнений.

## РЕЗЮМЕ

- TensorFlow Lite — упрощенная альтернатива «полной» TensorFlow. Она содержит только самые важные части, позволяющие применять модели глубокого обучения. Ее использование ускоряет и упрощает процесс развертывания моделей с AWS Lambda.
- Лямбда-функции можно запускать локально с помощью Docker. Таким образом, можно тестировать код, не развертывая его в AWS.
- Чтобы развернуть лямбда-функцию, необходимо поместить ее код в Docker, опубликовать образ Docker в ECR, а затем использовать URI образа при создании лямбда-функции.
- Чтобы предоставить доступ к лямбда-функции, мы используем API Gateway. Таким образом мы делаем лямбда-функцию доступной в качестве веб-сервиса, чтобы ее мог использовать любой желающий.

В данной главе мы использовали AWS Lambda — бессерверный подход для развертывания моделей глубокого обучения. Мы не хотели иметь дело с серверами и позволили среде позаботиться об этом.

В следующей главе все-таки обратим внимание на серверы и используем кластер Kubernetes для развертывания модели.

# *Предоставление моделей с помощью Kubernetes и Kubeflow*

---

## **В этой главе**

- ✓ Понимание различных методов развертывания и предоставления моделей через облако.
- ✓ Предоставление моделей Keras и TensorFlow с помощью TensorFlowServing.
- ✓ Развертывание TensorFlow Serving в Kubernetes.
- ✓ Использование Kubeflow и KServing для упрощения процесса развертывания.

В предыдущей главе мы говорили о развертывании модели с помощью AWS Lambda и TensorFlow Lite.

В этой главе мы обсудим «серверный» подход к развертыванию модели: предоставим доступ к модели классификации одежды с помощью TensorFlow Serving на Kubernetes. Кроме того, поговорим о Kubeflow, расширении для Kubernetes, которое упрощает развертывание модели.

В главе будет много материала, но Kubernetes настолько сложен, что у нас не будет возможности углубляться в детали. Из-за этого мы будем часто обращаться

к внешним ресурсам, которые позволяют более подробно освещать некоторые темы. Однако не волнуйтесь; вы узнаете достаточно, чтобы чувствовать себя комфортно при развертывании с помощью Kubernetes собственных моделей.

## 9.1. KUBERNETES И KUBEFLOW

Kubernetes — платформа контейнерной оркестровки. Звучит сложно, но это не что иное, как место, где мы можем развертывать контейнеры Docker. Kubernetes заботится о предоставлении этих контейнеров в качестве веб-сервисов и масштабирует их по мере изменения количества получаемых запросов.

Kubernetes — не самый простой инструмент для изучения, но очень эффективный. Вполне вероятно, что в какой-то момент вам понадобится его использовать. Именно поэтому мы и решили рассказать о нем в нашей книге.

Kubeflow — еще один популярный инструмент, созданный на основе Kubernetes. Он упрощает использование этой платформы для развертывания моделей машинного обучения. В текущей главе мы рассмотрим как Kubernetes, так и Kubeflow.

В первой части мы поговорим о TensorFlow Serving и чистом Kubernetes. Мы обсудим, как можно использовать эти технологии для развертывания модели. План для первой части таков:

- сначала мы преобразуем модель Keras в специальный формат, используемый TensorFlow Serving;
- затем используем TensorFlow Serving для локального запуска модели;
- после этого создадим сервис для предварительной обработки изображений и взаимодействия с сервисом TensorFlow;
- наконец, мы развернем как модель, так и сервис предварительной обработки с помощью Kubernetes.

### ПРИМЕЧАНИЕ

В этой главе мы не будем пытаться рассказывать о Kubernetes в деталях. Мы покажем только, как использовать его для развертывания моделей, и будем часто ссылаться на более специализированные источники, где все описано более подробно.

Во второй части мы используем Kubeflow, надстройку над Kubernetes, которая упрощает развертывание:

- мы используем модель, которую подготовим для TensorFlow Serving, и развернем ее с помощью KFServing — части Kubeflow, которая заботится о представлении;

- затем создадим трансформатор для предварительной обработки изображений и последующей обработки прогнозов.

Код для этой главы доступен в репозитории книги на GitHub (<https://github.com/alexeygrigorev/mlbookcamp-code/>) в папках chapter-09-kubernetes и chapter-09-kubeflow.

Начнем!

## 9.2. ПРЕДОСТАВЛЕНИЕ МОДЕЛЕЙ С ПОМОЩЬЮ TENSORFLOW SERVING

В главе 7 мы использовали Keras для прогнозирования классов изображений. В главе 8 преобразовали модель в TF Lite и использовали ее для составления прогнозов из AWS Lambda. В этой главе мы сделаем это с помощью TensorFlow Serving.

TensorFlow Serving, обычно сокращаемое как TF Serving, представляет собой систему, предназначенную для предоставления моделей TensorFlow. В отличие от TF Lite, который создан для мобильных устройств, TF Serving ориентирована на серверы. Часто на серверах имеются графические процессоры, и TF Serving умеет их использовать.

AWS Lambda отлично подходит для экспериментов и работы с небольшими объемами изображений — до миллиона в день. Но когда мы увеличиваем этот объем и получаем больше изображений, использование AWS Lambda становится дорогим. Лучшим вариантом будет развертывание моделей с помощью Kubernetes и TF Serving.

Однако простого использования TF Serving для развертывания моделей недостаточно. Нам также понадобится еще один сервис для подготовки изображений. Далее мы обсудим архитектуру создаваемой системы.

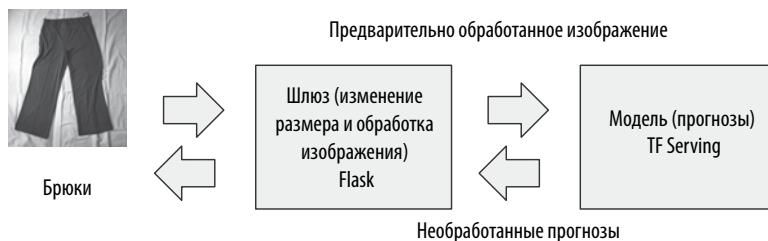
### 9.2.1. Обзор архитектуры предоставления

TF Serving фокусируется только на одном — предоставлении модели. Он ожидает, что получаемые данные уже подготовлены: изображения изменены, предварительно обработаны и отправлены в нужном формате.

Вот почему простого помещения модели в TF Serving недостаточно. Нам нужен дополнительный сервис, который возьмет на себя предварительную обработку данных.

Нам понадобятся два компонента для системы, предоставляющей модель глубокого обучения (рис. 9.1):

- шлюз — часть предварительной обработки. Он получает URL, для которого необходимо сделать прогноз, подготавливает его и переправляет в модель. Мы будем использовать Flask для создания этого сервиса;
- модель — часть с фактической моделью. Для нее мы будем использовать TF Serving.



**Рис. 9.1.** Двухуровневая архитектура нашей системы. Шлюз получает запросы пользователей и подготавливает данные, а TF Serving использует их для составления прогнозов

Создание системы с двумя компонентами вместо одного может показаться ненужным усложнением. В предыдущей главе в этом не было необходимости. У нас была только одна часть, а именно лямбда-функция.

В принципе, мы могли бы взять код данной лямбда-функции, поместить его в Flask и использовать для предоставления модели. Это вполне рабочий способ, но далеко не самый эффективный. Если требуется обработать миллионы изображений, то важно правильно использовать ресурсы.

Наличие двух отдельных компонентов облегчает выбор правильного ресурса для каждой части:

- шлюз тратит много времени на загрузку изображений в дополнение к предварительной обработке. Для этого не нужен мощный компьютер;
- компонент TF Serving требует более мощной машины, часто с графическим процессором. Было бы расточительно использовать ее для загрузки изображений;
- нам может потребоваться много экземпляров шлюза и лишь несколько экземпляров TF Serving. Разделив их, мы сможем масштабировать каждый из них независимо.

Мы начнем со второй составляющей – TF Serving.

В главе 7 мы обучили модель Keras. Чтобы использовать ее с TF Serving, нам придется преобразовать ее в специальный формат TF Serving, который называется `saved_model`. Этим и займемся далее.

### 9.2.2. Формат `saved_model`

Модель Keras, которую мы обучали ранее, была сохранена в формате `h5`. TF Serving не умеет работать с `h5`. Вместо этого она ожидает, что модели будут поступать в формате `saved_model`. В этом подразделе мы преобразуем модель `h5` в файл `saved_model`.

Если у вас нет модели из главы 7, то вы можете скачать ее с помощью `wget`:

```
wget https://github.com/alexeygrigorev/mlbookcamp-code/releases/download/chapter7-model/xception_v4_large_08_0.894.h5
```

Теперь преобразуем ее. Это можно сделать как из блокнота Jupyter, так и из сценария Python.

В любом случае мы начинаем с импорта:

```
import tensorflow as tf
from tensorflow import keras
```

Затем загружаем модель:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

И наконец, сохраняем ее в формате `saved_model`:

```
tf.saved_model.save(model, 'clothing-model')
```

Готово. После выполнения этого кода у нас появляется модель, сохраненная в папке `clothing-model`.

Чтобы использовать ее в дальнейшем, нам нужно знать несколько вещей:

- имя сигнатуры модели. Сигнтура описывает входные и выходные данные модели. Больше информации о сигнтурах моделей вы можете прочитать здесь: [https://www.tensorflow.org/tfx/serving/signature\\_defs](https://www.tensorflow.org/tfx/serving/signature_defs);
- имя входного слоя;
- имя выходного слоя.

При использовании Keras нам не нужно было беспокоиться о подобных вещах, но TF Serving требует наличия этой информации.

TensorFlow поставляется со специальной утилитой анализа моделей в формате `saved_model` — `saved_model_cli`. Нам не придется устанавливать ничего дополнительного. Мы используем команду `show` этой утилиты:

```
saved_model_cli show --dir clothing-model --all
```

Взглянем на результат:

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
```

```
...
signature_def['serving_default']: ← Определение сигнатуры —
The given SavedModel SignatureDef contains the following input(s):
  inputs['input_8'] tensor_info: ← Имя входа —
    dtype: DT_FLOAT
    shape: (-1, 299, 299, 3)
    name: serving_default_input_8:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['dense_7'] tensor_info: ← Имя выхода —
    dtype: DT_FLOAT
    shape: (-1, 10)
    name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
```

В этом выводе нас интересуют три аспекта:

- определение сигнатуры модели (`signature_def`). В данном случае это `serving_default`;
- вход (`input_8`) — имя входа модели;
- выход (`dense_7`) — имя выходного слоя модели.

#### **ПРИМЕЧАНИЕ**

Обратите внимание на эти имена — они понадобятся нам позже при вызове модели.

Модель преобразована, и теперь мы готовы предоставить ее с помощью TF Serving.

### **9.2.3. Локальный запуск TensorFlow Serving**

Один из самых простых способов запустить TF Serving локально — использовать Docker. Больше информации об этом вы можете прочитать в официальной документации: <https://www.tensorflow.org/tfx/serving/docker>. Дополнительная информация о Docker представлена в главе 5.

Все, что нам нужно сделать, — это вызвать команду `docker run`, указав путь к модели и ее имя:

```
docker run -it --rm \
-p 8500:8500 \ ❶ Открывает порт 8500
-v "$(pwd)/clothing-model:/models/clothing-model/1" \ ❷ Монтирует нашу
-e MODEL_NAME=clothing-model \ ❸ Указывает имя модели
tensorflow/serving:2.3.0 \ ❹ Использует образ TensorFlow
                           Serving версии 2.3.0
```

При запуске мы используем три параметра:

- `-p` — чтобы сопоставить порт 8500 на хост-компьютере (на котором мы запускаем Docker) с портом 8500 внутри контейнера ❶;
- `-v` — чтобы поместить файлы модели в образ Docker ❷. Модель помещается в `/models/clothing-model/1`, где `clothing-model` — это имя модели, а `1` — версия;
- `-e` — чтобы установить переменную `MODEL_NAME` в `clothing-model` ❸, которая является именем каталога из ❷.

Больше информации о команде `docker run` можно найти в официальной документации Docker (<https://docs.docker.com/engine/reference/run/>).

После выполнения команды мы должны увидеть в логе терминала следующее:

```
2020-12-26 22:56:37.315629: I tensorflow_serving/core/loader_harness.cc:87] 
Successfully loaded servable version {name: clothing-model version: 1} 
2020-12-26 22:56:37.321376: I tensorflow_serving/model_servers/server.cc:371] 
Running gRPC ModelServer at 0.0.0.0:8500 ... 
[evhttp_server.cc : 238] NET_LOG: Entering the event loop ...
```

Сообщение `Entering the event loop` говорит нам, что TF Serving стартовал успешно и готов к приему запросов.

Но мы пока не можем воспользоваться им. Чтобы подготовить запрос, нам потребуется загрузить изображение, предварительно обработать его и преобразовать в специальный двоичный формат. Далее мы узнаем, как это делается.

## 9.2.4. Вызов модели TF Serving из Jupyter

TF Serving использует gRPC — специальный протокол, разработанный для высокопроизводительной связи. Он основан на protobuf, формате для эффективной передачи данных. В отличие от JSON, он двоичный, что делает запросы значительно более компактными. Чтобы понять, как его использовать, сначала поэкспериментируем с данными технологиями из блокнота Jupyter. Мы подключимся к нашей модели, развернутой через TF Serving, используя gRPC и protobuf. После этого можем поместить данный код в приложение Flask (сделаем это несколько позже).

Начнем. Нам нужно установить пару библиотек:

- grpcio — для поддержки gRPC в Python;
- tensorflow-serving-api — для использования TF Serving из Python.

Установите их с помощью pip:

```
pip install grpcio==1.32.0 tensorflow-serving-api==2.3.0
```

Нам также потребуется библиотека keras\_image\_helper для предварительной обработки изображений. Мы уже использовали ее в главе 8. Если вы ее еще не установили, то используйте для этого pip:

```
pip install keras_image_helper==0.0.1
```

Затем создайте блокнот Jupyter. Мы можем назвать его `chapter-09-image-preparation`. Как обычно, начинаем с импорта:

```
import grpc
import tensorflow as tf

from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc
```

Мы импортировали:

- gRPC для связи с TF Serving;
- TensorFlow для определений protobuf (позже мы увидим, как это используется);
- пару функций из TensorFlow Serving.

Теперь нам нужно определить подключение к нашему сервису:

```
host = 'localhost:8500'
channel = grpc.insecure_channel(host)
stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
```

### **ПРИМЕЧАНИЕ**

Мы используем небезопасный канал, который не требует аутентификации. Все взаимодействия между сервисами, описанными в этой главе, происходят внутри одной и той же сети. Она закрыта для внешнего мира, поэтому использование небезопасного канала не вызывает каких-либо уязвимостей в системе безопасности. Настройка защищенного канала тоже возможна, но эта тема уже выходит за рамки книги.

Для предварительной обработки изображений мы, как и ранее, используем библиотеку keras\_image\_helper:

```
from keras_image_helper import create_preprocessor

preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Используем то же изображение брюк, которое использовали в главе 8 (рис. 9.2).



**Рис. 9.2.** Изображение брюк, которые мы используем для тестирования

Преобразуем его в массив NumPy:

```
url = "http://bit.ly/mlbookcamp-pants"  
X = preprocessor.from_url(url)
```

У нас есть массив NumPy в X, но в таком виде мы его использовать не сможем. Для gRPC нам нужно преобразовать его в protobuf. В TensorFlow для этого имеется специальная функция: `tf.make_tensor_proto`.

Вот как мы ее используем:

```
def np_to_protobuf(data):  
    return tf.make_tensor_proto(data, shape=data.shape)
```

Функция принимает два аргумента:

- массив NumPy: `data`;
- размеры этого массива: `data.shape`.

#### ПРИМЕЧАНИЕ

В данном примере для преобразования массива NumPy в protobuf мы используем TensorFlow. Это объемная библиотека, так что использовать ее из-за одной маленькой функции неразумно. В данной главе мы делаем это лишь для простоты, но в рабочей среде так поступать не следует. Использование Docker с большими изображениями

может быть чревато проблемами: на загрузку изображений уйдет больше времени, а сами они займут больше места. В репозитории <https://github.com/alexeygrigorev/tensorflow-protobuf> представлена информация, которая позволяет узнать, что можно сделать вместо этого.

Теперь мы можем использовать функцию `np_to_protobuf` для подготовки запроса gRPC:

```
pb_request = predict_pb2.PredictRequest() ← ❶ Если есть один элемент,
                                             то использовать один элемент
pb_request.model_spec.name = 'clothing-model' ← ❷ Устанавливает имя модели
pb_request.model_spec.signature_name = 'serving_default' ← в clothing-model
pb_request.inputs['input_8'].CopyFrom(np_to_protobuf(X)) ← ❸ Задает имя сигнатуры:
                                                               serving_default
                                                               ↓
                                                               Преобразует X в protobuf
                                                               и приписывает его входу input_8 ❹
```

Давайте пройдемся по каждой строке. Сначала в ❶ мы создаем объект запроса. TF Serving использует информацию из этого объекта, чтобы определить, как обрабатывать запрос.

В ❷ мы указываем имя модели. Напомним, что при запуске TF Serving в Docker мы указали параметр `MODEL_NAME` и установили его в `clothing-model`. Здесь мы сообщаем, что хотим отправить этой модели запрос.

В ❸ мы указываем запрашиваемую сигнатуру. Когда мы анализировали файл `saved_model`, именем сигнатуры было `serving_default`, так что его мы и используем. Узнать больше информации о подписях вы можете в официальной документации по TF Serving ([https://www.tensorflow.org/tfx/serving/signature\\_defs](https://www.tensorflow.org/tfx/serving/signature_defs)).

В ❹ мы делаем две вещи. Сначала преобразуем `X` в `protobuf`. Затем направляем результат преобразования на вход с именем `input_8`. Это имя также из нашего анализа файла `saved_model`.

Выполним код:

```
pb_result = stub.Predict(pb_request, timeout=20.0)
```

Он отправляет запрос экземпляру TF Serving. Затем TF Serving применяет модель к запросу и направляет обратно результаты. Результаты сохраняются в переменной `pb_result`. Чтобы извлечь из нее прогнозы, нам нужно получить доступ к одному из выходов:

```
pred = pb_result.outputs['dense_7'].float_val
```

Обратите внимание, что необходимо сослаться на конкретный выход по его имени — `dense_7`. При анализе сигнатур файла `saved_model` мы его запомнили и теперь используем для получения прогнозов.

Переменная `pred` представляет собой список значений с плавающей запятой — прогнозов:

```
[ -1.868, -4.761, -2.316, -1.062, 9.887, -2.812, -3.666, 3.200, -2.602, -4.835]
```

Нам нужно превратить этот список во что-то более удобоваримое, то есть связать его с метками. Мы используем тот же подход, что и в предыдущих главах:

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]

result = {c: p for c, p in zip(labels, pred)}
```

Конечный результат выглядит так:

```
{'dress': -1.868,
 'hat': -4.761,
 'longsleeve': -2.316,
 'outwear': -1.062,
 'pants': 9.887,
 'shirt': -2.812,
 'shoes': -3.666,
 'shorts': 3.200,
 'skirt': -2.602,
 't-shirt': -4.835}
```

Мы видим, что метка `pants` получила самую высокую оценку.

Нам успешно удалось подключиться к экземпляру TF Serving из блокнота Jupyter, использовав для этого gRPC и protobuf. Теперь поместим этот код в веб-сервис.

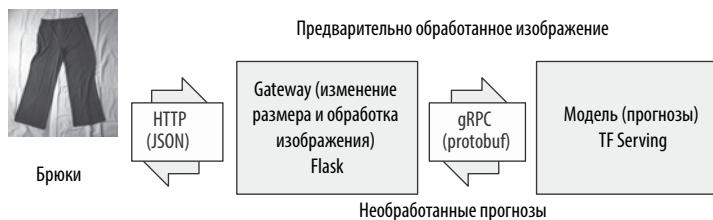
## 9.2.5. Создание службы Gateway

У нас уже написан весь код, позволяющий взаимодействовать с нашей моделью, развернутой с помощью TF Serving.

Тем не менее код не слишком удобен в применении. Пользователям нашей модели не нужно беспокоиться о загрузке изображения, выполнении предварительной

обработки, преобразований в protobuf и всех тех вещах, которые мы выполняли. Им нужна лишь возможность отправлять URL изображения и получать обратно прогнозы.

Чтобы облегчить их работу, мы разместим весь этот код в веб-сервисе. Пользователи будут взаимодействовать с сервисом, а тот — с TF Serving. Таким образом, сервис будет выступать в качестве шлюза для нашей модели. Вот почему мы можем просто назвать его Gateway (рис. 9.3).



**Рис. 9.3.** Сервис Gateway — приложение Flask, которое получает URL изображения и подготавливает его. Затем использует gRPC и protobuf для взаимодействия с TF Serving

Создавать такой сервис мы будем с помощью Flask. Ранее мы уже использовали Flask; более подробную информацию вы можете найти в главе 5.

Сервис Gateway должен выполнять следующие действия:

- получать URL изображения из запроса;
- скачивать изображение, предварительно обрабатывать его и преобразовывать в массив NumPy;
- преобразовывать массив NumPy в protobuf и использовать gRPC для связи с TF Serving;
- выполнять окончательную обработку результатов — преобразовывать необработанный список с числами в понятную человеку форму.

Приступим к ее созданию! Начнем с создания файла `model_server.py` — мы поместим в него всю перечисленную логику.

Для начала выполним тот же импорт, что и в блокноте:

```

import grpc
import tensorflow as tf
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc

from keras_image_helper import create_preprocessor

```

Теперь нам нужно добавить импорт Flask:

```
from flask import Flask, request, jsonify
```

Далее создадим заглушку подключения gRPC:

```
host = os.getenv('TF_SERVING_HOST', 'localhost:8500') ← | Делает URL TF Serving
channel = grpc.insecure_channel(host)           | конфигурируемым
stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
```

Вместо того чтобы просто выполнять жесткое кодирование URL экземпляра TF Serving, мы сделаем его настраиваемым с помощью переменной окружения `TF_SERVING_HOST`. Если переменная не задана, то используется значение по умолчанию `'localhost:8500'`.

Теперь создадим препроцессор:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Кроме того, нам нужно определить имена наших классов:

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]
```

Вместо того чтобы просто копировать и вставлять код из блокнота, мы можем сделать его более организованным и разделить на две функции:

- `make_request` — для создания запроса gRPC из массива NumPy;
- `process_response` — для присвоения меток классов прогнозам.

Начнем с `make_request`:

```
def np_to_protobuf(data):
    return tf.make_tensor_proto(data, shape=data.shape)

def make_request(X):
    pb_request = predict_pb2.PredictRequest()
    pb_request.model_spec.name = 'clothing-model'
    pb_request.model_spec.signature_name = 'serving_default'
    pb_request.inputs['input_8'].CopyFrom(np_to_protobuf(X))
    return pb_request
```

Далее создаем process\_response:

```
def process_response(pb_result):
    pred = pb_result.outputs['dense_7'].float_val
    result = {c: p for c, p in zip(labels, pred)}
    return result
```

И наконец соберем все воедино:

```
def apply_model(url):
    X = preprocessor.from_url(url)           | Предварительно
                                              | обрабатывает изображение
                                              | из предоставленного URL
    pb_request = make_request(X)            | Преобразует массив
                                              | NumPy в запрос gRPC
    pb_result = stub.Predict(pb_request, timeout=20.0) | Выполняет запрос
    return process_response(pb_result)       | Обрабатывает ответ
                                              | и закрепляет метки
                                              | за прогнозами
```

Код готов. Нам осталось сделать последний шаг: создать приложение Flask и функцию predict.

```
app = Flask('clothing-model')

@app.route('/predict', methods=['POST'])
def predict():
    url = request.get_json()
    result = apply_model(url['url'])
    return jsonify(result)

if name == "main":
    app.run(debug=True, host='0.0.0.0', port=9696)
```

Вот теперь все готово к запуску сервиса. Выполните в терминале следующую команду:

```
python model_server.py
```

Подождите, пока все не будет готово. В терминале мы должны увидеть следующее:

```
* Running on http://0.0.0.0:9696/ (Press CTRL+C to quit)
```

Пришло время все протестировать! Как и в главе 5, мы используем библиотеку requests. Вы можете открыть любой блокнот Jupyter. Например, можно продолжить в том же блокноте, где мы экспериментировали с подключением к TF Serving с помощью gRPC.

Нам нужно отправить запрос с URL и вывести ответ. Вот как мы это делаем с помощью requests:

```
import requests
```

```
req = {
```

```

"url": "http://bit.ly/mlbookcamp-pants"
}

url = 'http://localhost:9696/predict'

response = requests.post(url, json=req)
response.json()

```

Здесь мы отправляем POST-запрос в наш сервис и выводим результаты. Ответ такой же, как и ранее:

```
{
'dress': -1.868,
'hat': -4.761,
'longsleeve': -2.316,
'outwear': -1.062,
'pants': 9.887,
'shirt': -2.812,
'shoes': -3.666,
'shorts': 3.200,
'skirt': -2.602,
't-shirt': -4.835}
```

Сервис готов, и он работает локально. Развернем его с помощью Kubernetes!

## 9.3. РАЗВЕРТЫВАНИЕ МОДЕЛИ С ПОМОЩЬЮ KUBERNETES

Kubernetes — система оркестровки, позволяющая автоматизировать развертывание контейнеров. Ее можно использовать для размещения любых контейнеров Docker. В этом разделе мы увидим, как с помощью Kubernetes выполнять развертывание нашего приложения.

Но начнем с изучения основ Kubernetes.

### 9.3.1. Введение в Kubernetes

Основной единицей абстракции в Kubernetes служит *модуль*, или просто *под* (pod). Он содержит один образ Docker, и, когда мы хотим что-то предоставить, фактическую работу выполняют как раз модули.

Модули размещаются на *узле* (node) — это реальная машина. Узел обычно содержит один или несколько модулей.

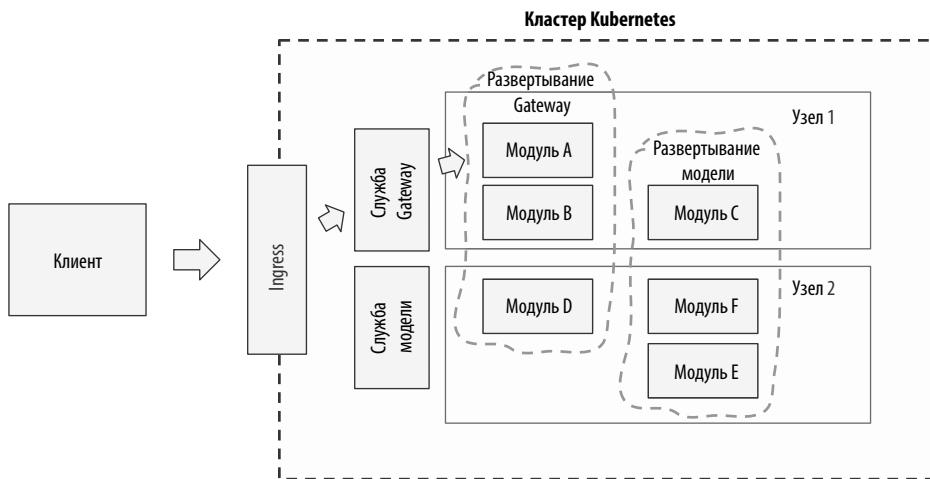
Чтобы развернуть приложение, мы определяем *развертывание* (deployment). Мы указываем, сколько модулей должно быть в приложении и какой образ следует использовать. Когда приложение начинает получать больше запросов, может

потребоваться больше модулей, чтобы справиться с увеличением трафика. Это также может происходить автоматически. Процесс называется *горизонтальным автоматическим масштабированием*.

*Сервис (service)* — точка входа в модули в развертывании. Клиенты взаимодействуют с сервисом, а не с отдельными модулями. Получая запрос, сервис направляет его в один из модулей развертывания.

Клиенты за пределами кластера Kubernetes взаимодействуют с сервисами внутри кластера через *точку входа (ingress)*.

Предположим, у нас уже есть сервис Gateway. Для него у нас имеется развертывание (Gateway Deployment) с тремя модулями — A (pod A), B (pod B) на узле 1 (node 1) и модуль D (pod D) на узле 2 (node 2) (рис. 9.4). Когда клиент хочет отправить запрос в сервис, тот сначала обрабатывается точкой входа (ingress), а затем сервис направляет запрос в один из модулей. В данном примере это модуль A, развернутый на узле 1. Сервис в модуле A обрабатывает запрос, и клиенту направляется ответ.



**Рис. 9.4.** Анатомия кластера Kubernetes. Модули — это экземпляры нашего приложения. Они работают на узлах — реальных машинах. Модули, принадлежащие одному и тому же приложению, группируются в развертывании. Клиент взаимодействует с сервисами, а сервисы направляют запрос в один из модулей в развертывании

Это очень краткое введение в ключевой словарь Kubernetes, но для начала его должно быть достаточно. Более подробную информацию о Kubernetes можно найти в официальной документации (<https://kubernetes.io/>).

В следующем подразделе мы узнаем, как создать собственный кластер Kubernetes на AWS.

### 9.3.2. Создание кластера Kubernetes на AWS

Чтобы развертывать наши сервисы в кластере Kubernetes, нам необходимо его иметь. Для этого есть несколько вариантов:

- можно создать кластер в облаке. Все основные облачные провайдеры позволяют настроить кластер Kubernetes в облаке;
- можно настроить его локально с помощью Minikube или MicroK8S. Больше информации об этом см. здесь: <https://mlbookcamp.com/article/local-k8s.html>.

В этом подразделе мы используем EKS от AWS. EKS (расшифровывается как Elastic Kubernetes Service) — сервис от AWS, позволяющий создавать кластер Kubernetes, прикладывая минимум усилий.

Альтернативами могут быть GKE (Google Kubernetes Engine) от Google Cloud и AKS (Azure Kubernetes Service) от Azure.

Для этого подраздела нам придется использовать три инструмента командной строки:

- AWS CLI управляет ресурсами AWS. В приложении А можно найти дополнительную информацию;
- `eksctl` управляет кластерами EKS (<https://docs.aws.amazon.com/eks/latest/userguide/eksctl.html>);
- `kubectl` управляет ресурсами в кластере Kubernetes (<https://kubernetes.io/docs/tasks/tools/install-kubectl/>). Все это понадобится для любого кластера, а не только для EKS.

Официальной документации достаточно для установки этих инструментов, но вы также можете посетить сайт книги, чтобы получить дополнительную информацию (<https://mlbookcamp.com/article/eks>).

Если вместо AWS вы пользуетесь услугами другого облачного провайдера, то вам необходимо использовать их инструменты для настройки кластера Kubernetes. Поскольку Kubernetes не привязан к какому-либо конкретному поставщику услуг, большинство инструкций в этой главе будут работать независимо от того, где расположен ваш кластер.

Как только вы установите `eksctl` и AWS CLI, мы сможем создать кластер EKS.

Сначала подготовим файл с конфигурацией кластера. Создайте файл в каталоге вашего проекта и назовите его `cluster.yaml`:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
```

```

name: ml-bookcamp-eks
region: eu-west-1
version: "1.18"

nodeGroups:
- name: ng
  desiredCapacity: 2
  instanceType: m5.xlarge

```

После создания конфигурационного файла мы можем использовать `eksctl` для развертывания кластера:

```
eksctl create cluster -f cluster.yaml
```

### ПРИМЕЧАНИЕ

Создание кластера занимает 15–20 минут, поэтому наберитесь терпения.

С помощью этой конфигурации мы создаем кластер с Kubernetes версии 1.18, развернутый в регионе `eu-west-1`. Имя кластера — `ml-bookcamp-eks`. Если вы хотите развернуть кластер в другом регионе, то можете его изменить. Этот кластер будет использовать две машины `m5.xlarge`. Прочитать больше материала о данном типе экземпляра вы можете здесь: <https://aws.amazon.com/ec2/instance-types/m5/>. Этого достаточно, чтобы проводить наши эксперименты из этой главы как для Kubernetes, так и для Kubeflow.

### ПРИМЕЧАНИЕ

На EKS не распространяется бесплатный уровень AWS. Узнать больше о ценах вы можете в официальной документации AWS (<https://aws.amazon.com/eks/pricing/>).

Когда кластер будет создан, нам нужно настроить `kubectl`, чтобы получить доступ. Для AWS мы сделаем это с помощью AWS CLI:

```
aws eks --region eu-west-1 update-kubeconfig --name ml-bookcamp-eks
```

Команда должна сгенерировать конфигурационный файл `kubectl` в месте по умолчанию. В Linux и macOS это `~/.kube/config`.

Пришло время проверить, что все работает и мы можем подключиться к нашему кластеру с помощью команды `kubectl`:

```
kubectl get service
```

Она возвращает список сервисов, запущенных в данный момент. Мы еще ничего не развернули, поэтому ожидаем увидеть только один сервис — сам Kubernetes. Результат, который мы должны увидеть, выглядит так:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.100.0.1	<none>	443/TCP	6m17s

Соединение работает, и можно приступать к развертыванию сервиса. Для этого нам сначала нужно подготовить образ Docker с фактическим сервисом. Сделаем это.

### 9.3.3. Подготовка образов Docker

В предыдущих подразделах мы создали два компонента системы предоставления:

- TF-Serving — компонент с фактической моделью;
- Gateway — компонент для предварительной обработки изображений, который взаимодействует с TF Serving.

Теперь мы их развернем. Начнем с образа TF Serving.

#### Образ TensorFlow Serving

Как и в главе 8, прежде всего следует опубликовать наш образ в ECR — реестре Docker AWS. Создадим реестр с именем model-serving:

```
aws ecr create-repository --repository-name model-serving
```

Он должен возвращать путь, подобный этому:

```
<ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/model-serving
```

#### ВАЖНО

Запомните этот путь, он нам понадобится позже.

При запуске образа Docker TF Serving локально мы использовали эту команду (сейчас ее выполнять не нужно):

```
docker run -it --rm \
-p 8500:8500 \
-v "$(pwd)/clothing-model:/models/clothing-model/1" \
-e MODEL_NAME=clothing-model \
tensorflow/serving:2.3.0
```

Мы использовали параметр `-v` для монтирования модели из `clothing-model` в папку `/models/clothing-model/1` внутри образа.

Это возможно сделать и с помощью Kubernetes, но в данной главе мы ограничимся простым подходом и включим модель в сам образ (как делали это в главе 8).

Создадим для этого Dockerfile. Назвать его можно `tf-serving.dockerfile`:

```
FROM tensorflow/serving:2.3.0
ENV MODEL_NAME clothing-model
COPY clothing-model /models/clothing-model/1
```

Мы основываем наш образ на образе TensorFlow Serving в ❶. Далее в ❷ мы задаем переменной окружения MODEL\_NAME значение clothing-model, что эквивалентно параметру -e. Далее копируем модель в /models/clothing-model/1 в ❸, что эквивалентно использованию параметра -v.

### **ПРИМЕЧАНИЕ**

Если вы хотите использовать компьютер с графическим процессором, то воспользуйтесь образом tensorflow/serving:2.3.0-gpu (закомментировано как ❶ в файле Dockerfile).

Запустим сборку:

```
IMAGE_SERVING_LOCAL="tf-serving-clothing-model"
docker build -t ${IMAGE_SERVING_LOCAL} -f tf-serving.dockerfile .
```

Далее нам нужно опубликовать образ в ECR. Сначала необходимо пройти аутентификацию в ECR с помощью AWS CLI:

```
$(aws ecr get-login --no-include-email)
```

### **ПРИМЕЧАНИЕ**

При вводе команды необходимо включить \$. Команда внутри круглых скобок возвращает другую команду. Используя \$(), мы выполняем эту команду.

Затем пометим образ с помощью удаленного URI:

```
ACCOUNT=XXXXXXXXXXXX
REGION=eu-west-1
REGISTRY=${ACCOUNT}.dkr.ecr.${REGION}.amazonaws.com/model-serving
IMAGE_SERVING_REMOTE=${REGISTRY}: ${IMAGE_SERVING_LOCAL}
docker tag ${IMAGE_SERVING_LOCAL} ${IMAGE_SERVING_REMOTE}
```

Обязательно измените переменные ACCOUNT и REGION.

Теперь мы готовы к отправке образа в ECR:

```
docker push ${IMAGE_SERVING_REMOTE}
```

Готово! Теперь необходимо проделать все то же самое с компонентом Gateway.

### **Образ Gateway**

Сначала подготовим образ для компонента Gateway. Это веб-сервис, основанный на ряде библиотек Python:

- Flask и Gunicorn;
- keras\_image\_helper;
- grpcio;

- TensorFlow;
- TensorFlow-Serving-API.

Как вы помните, в главе 5 с помощью Pipenv мы управляли зависимостями. Используем данную команду и здесь:

```
pipenv install flask gunicorn \
    keras_image_helper==0.0.1 \
    grpcio==1.32.0 \
    tensorflow==2.3.0 \
    tensorflow-serving-api==2.3.0
```

Выполнение команды создаст два файла: `Pipfile` и `Pipfile.lock`.

### **ВНИМАНИЕ**

Мы уже упоминали об этом, однако это достаточно важно, чтобы повторить еще раз. Здесь используем TensorFlow лишь из-за одной функции. В производственной среде лучше не устанавливать TensorFlow. В текущей главе мы делаем это, только чтобы упростить. Вместо того чтобы зависеть от TensorFlow, мы можем взять лишь нужные нам файлы protobuf и значительно уменьшить размер нашего образа Docker. Инструкции можно найти в данном репозитории: <https://github.com/alexeygrigorev/tensorflow-protobuf>.

Теперь создадим образ Docker. Начните с создания Dockerfile с именем `gateway.dockerfile`, содержимое которого выглядит так:

```
FROM python:3.7.5-slim

ENV PYTHONUNBUFFERED=TRUE

RUN pip --no-cache-dir install pipenv

WORKDIR /app

COPY ["Pipfile", "Pipfile.lock", "./"]
RUN pipenv install --deploy --system && \
    rm -rf /root/.cache

COPY "model_server.py" "model_server.py"

EXPOSE 9696

ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "model_server:app"]
```

Этот Dockerfile очень похож на файл, который у нас был ранее. В главе 5 можно найти дополнительную информацию.

Создадим образ:

```
IMAGE_GATEWAY_LOCAL="serving-gateway"
docker build -t ${IMAGE_GATEWAY_LOCAL} -f gateway.dockerfile .
```

И отправим его в ECR:

```
IMAGE_GATEWAY_REMOTE=${REGISTRY}:${IMAGE_GATEWAY_LOCAL}
docker tag ${IMAGE_GATEWAY_LOCAL} ${IMAGE_GATEWAY_REMOTE}

docker push ${IMAGE_GATEWAY_REMOTE}
```

#### **ПРИМЕЧАНИЕ**

Чтобы проверить, работают ли эти образы вместе локально, необходимо использовать Docker Compose (<https://docs.docker.com/compose/>). Это очень полезный инструмент, и мы рекомендуем уделить время для его изучения (здесь мы не будем описывать его).

Мы опубликовали оба образа в ECR и теперь готовы развернуть сервисы в Kubernetes! Этим сейчас и займемся.

### **9.3.4. Развёртывание в Kubernetes**

Прежде чем приступить к развертыванию, вернемся к основам Kubernetes. Внутри кластера у нас есть следующие объекты:

- модуль (pod) — самый маленький блок в Kubernetes. Это единый процесс, и у нас всегда только один контейнер Docker в одном модуле;
- развертывание — группа из нескольких связанных модулей;
- сервис — то, что располагается перед развертыванием и перенаправляет запросы в отдельные модули.

Чтобы развернуть приложение в Kubernetes, нам нужно настроить:

- развертывание — оно задает, как будут выглядеть модули этого развертывания;
- сервис: он указывает, как получить доступ к сервису и как тот подключается к модулям.

Начнем с настройки развертывания для TF Serving.

#### **Развёртывание для TF Serving**

В Kubernetes все обычно настраивается с помощью файлов YAML. Для настройки развертывания мы создадим файл с именем `tf-serving-clothing-model-deployment.yaml` в нашем каталоге проектов, включив в него следующее содержимое:

```
apiVersion: apps/v1
kind: Deployment ← ❶ Настраивает развертывание
metadata:
```

```
name: tf-serving-clothing-model
labels:
  app: tf-serving-clothing-model
spec:
  replicas: 1 ← ❶ Создает только один экземпляр
  selector: ❷ сервис — один модуль
    matchLabels:
      app: tf-serving-clothing-model ←
  template:
    metadata:
      labels:
        app: tf-serving-clothing-model ←
  spec:
    containers: ❸ Задает спецификацию для модулей
      - name: tf-serving-clothing-model ←
        image: <ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/model-serving:tf-
serving-clothing-model
    ports:
      - containerPort: 8500 ← ❹ Открывает порт 8500
  
```

❺ Задает имя развертывания

❻ Использует образ, который мы создали ранее

Конфигурация довольно длинная, так что разберем все ее важные строки.

В ❶ мы указываем тип объекта Kubernetes, который хотим настроить в данном файле YAML, а именно развертывание.

В ❷ мы определяем имя развертывания, а также задаем некоторую информацию о метаданных. Нам придется повторить это несколько раз: один раз для установки имени развертывания (`name`) и еще несколько раз (`labels: app`) для сервиса, которую мы настроим позже.

В **❸** мы устанавливаем количество экземпляров (модулей), которые хотим получить в развертывании.

В ④ мы указываем конфигурацию для модулей — устанавливаем параметры, которые будут иметь все модули.

В **❸** мы задаем URI для образа Docker. Модуль будет использовать этот образ. Не забудьте указать в нем идентификатор вашей учетной записи, а также нужный регион.

Наконец, в **6** мы открываем порт 8500 на модулях этого развертывания. Это порт, который использует TF Serving.

Чтобы узнать больше о настройке развертываний в Kubernetes, ознакомьтесь с официальной документацией (<https://kubernetes.io/docs/concepts/workloads/controllers/deployment>). Конфигурация готова. Теперь нужно использовать ее для создания объекта Kubernetes — в нашем случае развертывания. Мы делаем это с помощью команды `apply` из `kubectl`:

```
kubectl apply -f tf-serving-clothing-model-deployment.yaml
```

Параметр `-f` сообщает `kubectl`, что необходимо прочитать конфигурацию из файла конфигурации.

Чтобы убедиться в том, что все работает, необходимо проверить, появилось ли новое развертывание. Вот как можно получить список всех активных развертываний:

```
kubectl get deployments
```

Результат должен выглядеть примерно так:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
<code>tf-serving-clothing-model</code>	1/1	1	1	41s

Мы видим, что наше развертывание на месте. Кроме того, можем получить список модулей. Это очень похоже на получение списка всех развертываний:

```
kubectl get pods
```

На выходе должно получиться что-то наподобие этого:

NAME	READY	STATUS	RESTARTS	AGE
<code>tf-serving-clothing-model-56bc84678d-b6n4r</code>	1/1	Running	0	108s

Далее необходимо создать сервис поверх этого развертывания.

## Сервис для TF Serving

Мы хотим вызвать TF Serving из Gateway. Для этого необходимо создать сервис перед развертыванием TF Serving.

Как и при развертывании, мы начинаем с создания файла конфигурации для сервиса. Это также файл YAML. Создайте файл с именем `tf-serving-clothing-model-service.yaml`, содержимое которого выглядит так:

```
apiVersion: v1
kind: Service
metadata:
  name: tf-serving-clothing-model
  labels:
    app: tf-serving-clothing-model
spec:
  ports:
    - port: 8500
      targetPort: 8500
      protocol: TCP
      name: http
  selector:
    app: tf-serving-clothing-model
```

The diagram shows annotations and service specification:

- Annotations:** `name: tf-serving-clothing-model` and `labels: app: tf-serving-clothing-model` are annotated with arrows pointing to a callout box labeled "Настраивает имя сервиса" (Sets service name).
- Service Specification:** `ports:` is annotated with a callout box labeled "Спецификация сервиса — порт, который будет использоваться" (Service specification — port to be used).
- Selector:** `selector: app: tf-serving-clothing-model` is annotated with a callout box labeled "Подключает сервис к развертыванию с помощью указания метки развертывания" (Connects the service to the deployment via the deployment label).

Мы применяем его таким же образом — с помощью команды `apply`:

```
kubectl apply -f tf-serving-clothing-model-service.yaml
```

Чтобы проверить, все ли работает, мы можем получить список всех сервисов и посмотреть, есть ли там наш:

```
kubectl get services
```

Мы должны увидеть что-то наподобие этого:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.100.0.1	<none>	443/TCP	84m
tf-serving-clothing-model	ClusterIP	10.100.111.165	<none>	8500/TCP	19s

В дополнение к сервису Kubernetes по умолчанию мы видим сервис tf-serving-clothing-model, который только что создали.

Чтобы получить доступ к сервису, нам нужно получить его URL. Внутренние URL обычно следуют шаблону:

```
<service-name>. <namespace-name>. svc.cluster.local
```

Часть `<service-name>` — это tf-serving-clothing-model.

Мы не использовали какое-либо конкретное пространство имен для этого сервиса, поэтому Kubernetes автоматически поместил сервис в пространство имен по умолчанию. Мы не будем здесь рассматривать пространства имен, но вы можете прочитать о них в официальной документации (<https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>).

Это URL для сервиса, который мы только что создали:

```
tf-serving-clothing-model.default.svc.cluster.local
```

Данный URL понадобится нам позже при настройке Gateway.

Мы создали развертывание для TF Serving, а также сервис. Теперь перейдем к развертыванию для Gateway.

## Развертывание для Gateway

Как и ранее, начнем с создания файла YAML с конфигурацией. Создайте файл с именем `serving-gateway-deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: serving-gateway
  labels:
    app: serving-gateway
spec:
  replicas: 1
  selector:
    matchLabels:
      app: serving-gateway
```

```

template:
  metadata:
    labels:
      app: serving-gateway
  spec:
    containers:
      - name: serving-gateway
        image: <ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/modeserving:
          serving-gateway
    ports:
      - containerPort: 9696
    env:
      - name: TF_SERVING_HOST
        value: "tf-serving-clothing-model.default.svc.cluster.local:8500"


```

Замените <ACCOUNT> и <REGION> в URL образа своими значениями.

Конфигурация этого развертывания очень похожа на развертывание TF Serving, но имеет одно важное различие: мы указываем в переменной **TF\_SERVING\_HOST** URL сервиса с нашей моделью (в листинге выделено жирным шрифтом).

Теперь применим полученную конфигурацию:

```
kubectl apply -f serving-gateway-deployment.yaml
```

В результате должен появиться новый модуль и новое развертывание. Давайте взглянем на список модулей:

```
kubectl get pod
```

И действительно, появился новый модуль:

NAME	READY	STATUS	RESTARTS	AGE
tf-serving-clothing-model-56bc84678d-b6n4r	1/1	Running	0	1h
serving-gateway-5f84d67b59-1x8tq	1/1	Running	0	30s

## ВНИМАНИЕ

Gateway использует gRPC для взаимодействия с TF Serving. При развертывании нескольких экземпляров TF Serving вы можете столкнуться с проблемой распределения нагрузки между ними (<https://kubernetes.io/blog/2018/11/07/grpc-load-balancing-on-kubernetes-without-tears/>). Чтобы решить эту проблему, вам придется установить инструмент для совместной работы сервисов, такой как Linkerd, Istio или что-то подобное. Поговорите с операционной командой, чтобы узнать, как сделать это в рамках вашей компании.

Мы создали развертывание для Gateway, и теперь нужно настроить для него сервис. Это мы и сделаем.

## Сервис для Gateway

Мы подготовили развертывание для Gateway, и теперь нам нужно создать сервис. Он отличается от сервиса, который мы создавали для TF Serving, поскольку должен быть общедоступным. Необходимо, чтобы его можно было использовать из-за пределов нашего кластера Kubernetes. Для этого мы используем специальный тип сервиса — LoadBalancer. Он создает внешний балансировщик нагрузки, который доступен за пределами кластера Kubernetes. В случае AWS он использует ELB, сервис гибкой балансировки нагрузки.

Создадим конфигурационный файл с именем `serving-gateway-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: serving-gateway
  labels:
    app: serving-gateway
spec:
  type: LoadBalancer ① Использует тип LoadBalancer
  ports:
    - port: 80
      targetPort: 9696 ② Сопоставляет порт 9696 в модуле с портом 80 в сервисе
      protocol: TCP
      name: http
  selector:
    app: serving-gateway
```

В ① мы указываем тип сервиса — LoadBalancer.

В ② мы подключаем порт 80 в сервисе к порту 9696 в модулях. Таким образом, нам не нужно указывать порт при подключении к сервису — будет использоваться HTTP-порт по умолчанию, а именно 80.

Применим получившуюся конфигурацию:

```
kubectl apply -f serving-gateway-service.yaml
```

Чтобы увидеть внешний URL сервиса, используйте команду `describe`:

```
kubectl describe service serving-gateway
```

Она выведет определенную информацию о сервисе:

Name:	serving-gateway
Namespace:	default
Labels:	<none>
Annotations:	<none>
Selector:	app=serving-gateway
Type:	LoadBalancer
IP Families:	<none>

```

IP:           10.100.100.24
IPs:          <none>
LoadBalancer Ingress: ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-
                   1.elb.amazonaws.com
Port:          http 80/TCP
TargetPort:    9696/TCP
NodePort:      http 32196/TCP
Endpoints:    <none>
Session Affinity: None
External Traffic Policy: Cluster
Events:
  Type   Reason        Age   From            Message
  ----  -----        ---   ----            -----
Normal  EnsuringLoadBalancer  4s    service-controller  Ensuring load
Normal  EnsuredLoadBalancer   2s    service-controller  Ensured load
                                         balancer
                                         balancer

```

Нас интересует строка с `LoadBalancer Ingress`. Это URL, который нужно использовать для доступа к сервису Gateway. В нашем случае это URL

`ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-1.elb.amazonaws.com`

Сервис Gateway готов к использованию. Попробуем сделать это!

### 9.3.5. Тестирование сервиса

При локальном запуске TF Serving и Gateway мы подготовили простой фрагмент кода на Python для тестирования сервиса. Используем его повторно. Перейдите в тот же блокнот и замените локальный IP-адрес на URL, который мы получили в предыдущем подразделе:

```

import requests

req = {
    "url": "http://bit.ly/mlbookcamp-pants"
}

url = 'http://ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-
       1.elb.amazonaws.com/predict'

response = requests.post(url, json=req)
response.json()

```

Запустите код. В результате мы получаем те же прогнозы, что и ранее:

```

{'dress': -1.86829,
 'hat': -4.76124,
 'longsleeve': -2.31698,
 'outwear': -1.06257,
 'pants': 9.88716,

```

```
'shirt': -2.81243,  
'shoes': -3.66628,  
'shorts': 3.20036,  
'skirt': -2.60233,  
't-shirt': -4.83504}
```

Все работает, и это означает, что мы только что успешно внедрили нашу модель глубокого обучения с помощью TF Serving и Kubernetes!

### **ВАЖНО**

Если вы закончили экспериментировать с EKS, то не забудьте закрыть кластер. Если вы не выключите его, то вам придется платить за него, даже если он простояивает и вы им не пользуетесь. Инструкции по выключению найдете в конце текущей главы.

В этом примере мы рассмотрели Kubernetes только с точки зрения пользователя, но не с операционной. Мы не говорили об автоматическом масштабировании, мониторинге, оповещении и других важных темах, необходимых для создания моделей машинного обучения.

Более подробную информацию по этим темам можно найти в книге по Kubernetes или в официальной документации Kubernetes.

Вы, наверное, заметили, что нам пришлось проделать довольно много всего для развертывания одной модели: создать образ Docker, отправить его в ECR, создать развертывание, создать сервис. Сделать все это для пары моделей не проблема, но если необходимы десяток или сотня моделей, то работа станет проблематичной и монотонной.

Но есть решение, которое упрощает развертывание, — Kubeflow. В следующем разделе мы узнаем, как использовать его для предоставления моделей Keras.

## **9.4. РАЗВЕРТЫВАНИЕ МОДЕЛИ С ПОМОЩЬЮ KUBEFLOW**

Kubeflow — проект, предназначенный для упрощения развертывания сервисов машинного обучения в Kubernetes.

Он состоит из набора инструментов, каждый из которых направлен на решение конкретной задачи. Например:

- Kubeflow Notebooks Server упрощает централизованное размещение блок-нотов Jupyter;
- Kubeflow Pipelines автоматизирует процесс обучения;
- Katib выбирает наилучшие параметры для модели;

- Kubeflow Serving (сокращенно Kfserv) развертывает модели машинного обучения.

В проект входят и многие другие инструменты. Больше информации обо всех компонентах вы можете прочитать здесь: <https://www.kubeflow.org/docs/components/>.

В этой главе мы сосредоточимся на развертывании модели, поэтому нам нужно будет использовать только один компонент Kubeflow — KFServing.

Если вы хотите установить весь проект Kubeflow целиком, то обратитесь к официальной документации. В ней содержатся инструкции по установке для основных облачных провайдеров, таких как Google Cloud Platform, Microsoft Azure и AWS (<https://www.kubeflow.org/docs/aws/aws-e2e/>).

Инструкции по установке только KFServing без остальной части Kubeflow на AWS см. на сайте книги: <https://mlbookcamp.com/article/kfserving-eks-install>. Мы использовали эту статью для настройки среды для остальной части текущей главы, но приведенный здесь код должен работать с любой установкой Kubeflow с незначительными изменениями.

#### **ПРИМЕЧАНИЕ**

Установка может показаться нетривиальной, особенно если вы раньше не делали ничего подобного. Если вы в чем-то не уверены, то попросите о помощи в настройке кого-нибудь из операционной группы.

### **9.4.1. Подготовка модели: загрузка ее в S3**

Чтобы развернуть модель Keras с помощью KFServing, нам сначала нужно преобразовать ее в формат saved\_model. Мы уже делали это, поэтому можем просто использовать преобразованные файлы.

Далее нам нужно создать корзину в S3, куда мы поместим наши модели. Назовем ее `mlbookcamp-models-<Имя>`, где `<Имя>` может быть любым именем — например вашим. Имена корзин должны быть уникальными по всей AWS. Вот почему необходимо добавлять какой-нибудь суффикс к имени корзины. Она должна размещаться в том же регионе, что и наш кластер EKS. В нашем случае это eu-west-1.

Создать ее можно с помощью AWS CLI:

```
aws s3api create-bucket \
--bucket mlbookcamp-models-alexey \
--region eu-west-1 \
--create-bucket-configuration LocationConstraint=eu-west-1
```

Когда корзина будет создана, следует загрузить в нее модель. Используйте для этого AWS CLI:

```
aws s3 cp --recursive clothing-model s3://mlbookcamp-models-alexey/clothing-
model/0001/
```

Обратите внимание на 0001 в конце. Это важно: KFServing, как и TF Serving, нуждается в версии модели. У нас нет никаких предыдущих версий данной модели, поэтому мы добавляем в конце 0001.

Теперь мы готовы развернуть модель.

## 9.4.2. Развёртывание моделей TensorFlow с помощью KFServing

Ранее, при развертывании нашей модели с помощью обычного Kubernetes, нам требовалось настроить развертывание, а затем сервис. Вместо всего этого KFServing определяет особый вид объекта Kubernetes – InferenceService. Нам нужно настроить его одинажды, а он уже автоматически позаботится о создании всех остальных объектов Kubernetes, включая сервис и развертывание.

Сначала создайте еще один файл YAML (`tf-clothes.yaml`) со следующим содержимым:

```
apiVersion: "serving.kubeflow.org/v1beta1"
kind: "InferenceService"
metadata:
  name: "clothing-model"                                ② Указывает местоположение в S3 с моделью
spec:
  default:
    predictor:
      serviceAccountName: sa                         ① Использует serviceAccountName
                                                       для доступа к S3
      tensorflow:                                     ←
      storageUri: "s3:/ /mlbookcamp-models-alexey/clothing-model"
```

При доступе к модели из S3 нам нужно указать имя учетной записи сервиса, чтобы иметь возможность получить модель. Таким образом KFServing узнает о том, как получить доступ к корзине S3, — и мы указываем это в ①. Это также рассматривается в статье об установке KFServing на EKS (<https://mlbookcamp.com/article/kfserving-eks-install>).

Как и в случае с обычным Kubernetes, мы используем `kubectl` для применения конфигурации:

```
kubectl apply -f tf-clothing.yaml
```

Поскольку команда создает объект InferenceService, нам потребуется получить список таких объектов с помощью команды `get` из `kubectl`:

```
kubectl get inferenceservice
```

Мы должны увидеть что-то наподобие этого:

NAME	URL	READY	AGE
clothing-model	http://clothing-model...	True ...	97s

Если `READY` нашего сервиса еще не установлен в `True`, то нам нужно дождаться его готовности. Это может занять одну-две минуты.

Теперь обратите внимание на URL и название модели:

- URL — `https://clothing-model.default.kubeflow.mlbookcamp.com/v1/models/clothing-model`. В вашей конфигурации хост будет другим, поэтому весь URL тоже будет другим;
- название модели — `clothing-model`.

#### ПРИМЕЧАНИЕ

Может потребоваться некоторое время, чтобы URL стал доступен с нашего ноутбука. Распространение изменений в DNS может потребовать некоторого времени.

### 9.4.3. Доступ к модели

Модель развернута. Используем ее! Для этого мы можем запустить блокнот Jupyter или создать файл сценария на Python.

KFServing использует HTTP и JSON, поэтому для связи с ним мы применяем библиотеку `requests`. Прежде всего необходимо ее импортировать:

```
import requests
```

Далее нам нужно использовать препроцессор изображений для их подготовки. Он тот же самый:

```
from keras_image_helper import create_preprocessor

preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Теперь для теста нам нужно само изображение. Мы возьмем уже использовавшееся изображение брюк и задействуем тот же код, чтобы получить его и предварительно обработать:

```
image_url = "http://bit.ly/mlbookcamp-pants"
X = preprocessor.from_url(image_url)
```

Переменная `X` содержит массив NumPy. Прежде чем отправить данные в KFServing, нам нужно преобразовать его в список:

```
data = {
    "instances": X.tolist()
}
```

У нас есть запрос. В качестве следующего шага нужно определить URL, по которому следует отправлять этот запрос. У нас он сохранился из предыдущего подраздела, но потребуется слегка изменить его:

- мы заменим HTTP на HTTPS;
- добавим :predict в конце URL.

С учетом этих изменений URL выглядит следующим образом:

```
url = 'https://clothing-model.default.kubeflow.mlbookcamp.com/v1/models/
clothing-model:predict'
```

Мы готовы отправить запрос:

```
resp = requests.post(url, json=data)
results = resp.json()
```

Взглянем на результаты:

```
{'predictions': [[-1.86828923,
 -4.76124525,
 -2.31698346,
 -1.06257045,
 9.88715553,
 -2.81243205,
 -3.66628242,
 3.20036,
 -2.60233665,
 -4.83504581]]}
```

Нам нужно перевести прогнозы в удобочитаемую форму, как мы уже делали ранее. Для этого каждому элементу результата присваивается метка:

```
pred = results['predictions'][0]
```

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
```

```
'skirt',
't-shirt'
]

result = {c: p for c, p in zip(labels, pred)}
```

Результат выглядит следующим образом:

```
{'dress': -1.86828923,
'hat': -4.76124525,
'longsleeve': -2.31698346,
'outwear': -1.06257045,
'pants': 9.88715553,
'shirt': -2.81243205,
'shoes': -3.66628242,
'shorts': 3.20036,
'skirt': -2.60233665,
't-shirt': -4.83504581}
```

Мы развернули нашу модель, и теперь ее можно использовать.

Но странно было бы ожидать, что пользователей наших моделей обрадует необходимость самостоятельно готовить изображения. В следующем подразделе мы поговорим о преобразователях, которые могут взять на себя всю предварительную обработку изображений.

#### 9.4.4. Преобразователи KFServing

Выше мы представили сервис Gateway. Он располагался между клиентом и моделью и заботился о преобразовании запросов от клиентов в формат, ожидаемый моделью (рис. 9.5).



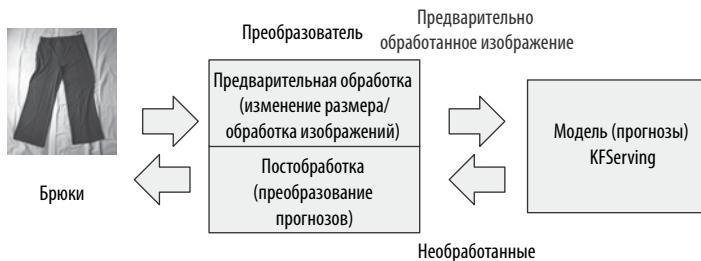
**Рис. 9.5.** Сервис Gateway выполняет предварительную обработку изображения, поэтому клиентам наших приложений этого делать уже не нужно

К счастью для нас, нам не потребуется добавлять еще один сервис Gateway для KFServing. Вместо этого мы можем использовать *преобразователь*.

Преобразователи заботятся:

- о предварительной обработке запроса, поступающего от клиента, и о преобразовании его в формат, ожидаемый моделью;
- о постобработке выходных данных модели — преобразовании их в формат, необходимый клиенту.

Мы можем поместить в такой преобразователь весь код предварительной обработки из предыдущего раздела (рис. 9.6).



**Рис. 9.6.** Преобразователь KFServing может загрузить изображение и подготовить его на этапе предварительной обработки, а также прикрепить метки к выходным данным модели на этапе постобработки

Как и сервис Gateway, которую мы создавали вручную, преобразователи в KFServing развертываются отдельно от модели. Это означает, что они могут независимо масштабироваться в обе стороны. Это хорошо, так как они выполняют другой тип работы:

- преобразователи выполняют работу ввода-вывода (загружают изображение);
- модели выполняют работу с интенсивным использованием процессора (применяя нейронную сеть для получения прогнозов).

Чтобы создать преобразователь, нам нужно установить библиотеку KFServing для Python и создать класс, который расширяет KFModel.

Это выглядит примерно так:

```

class ImageTransformer(kfserving.KFModel):
    def preprocess(self, inputs):
        # implement pre-processing logic

    def postprocess(self, inputs):
        # implement post-processing logic
  
```

Мы не будем вдаваться в подробности создания собственного преобразователя, но если вы захотите узнать, как это делается, то ознакомьтесь с данной статьей: <https://mlbookcamp.com/article/kfserving-transformers>. Мы же, специально для

этой книги, подготовили преобразователь, который использует библиотеку keras\_image\_helper. Вы можете изучить его исходный код здесь: <https://github.com/alexeygrigorev/kfserving-keras-transformer>.

Воспользуемся им. Сначала нам нужно удалить старый сервис вывода:

```
kubectl delete -f tf-clothes.yaml
```

Затем обновить конфигурационный файл (`tf-clothes.yaml`) и включить в него раздел `transformer` (выделенный жирным шрифтом):

```
apiVersion: "serving.kubeflow.org/v1alpha2"
kind: "InferenceService"
metadata:
  name: "clothing-model"
spec:
  default:
    predictor:                                ← Определяет модель в разделе predictor
    serviceAccountName: sa
    tensorflow:
      storageUri: "s3:/mlbookcamp-models-alexey/clothing-model"
  transformer:                                ← Определяет преобразователь в разделе transformer
    custom:
      container:
        image: "agrigorev/kfserving-keras-transformer:0.0.1" ← Задает образ для преобразователя
        name: user-container
      env:
        - name: MODEL_INPUT_SIZE
          value: "299,299"
        - name: KERAS_MODEL_NAME
          value: "xception"
        - name: MODEL_LABELS
          value: "dress,hat,longsleeve,outwear,pants,shirt,shoes,shorts,skirt,t-shirt"
```

Настраивает его — указывает размер ввода, имя модели и метки

В дополнение к разделу `predictor`, который уже был, мы добавляем еще один — `transformer`. Преобразователь, который мы используем, является общедоступным образом на сайте [agrigorev/kfserving-keras-transformer:0.0.1](https://agrigorev/kfserving-keras-transformer:0.0.1).

Чтобы выполнить преобразование, он использует библиотеку `keras_image_helper`. Для этого нам нужно установить три параметра:

- `MODEL_INPUT_SIZE` — размер входных данных, который ожидает модель:  $299 \times 299$ ;
- `KERAS_MODEL_NAME` — название архитектуры из приложений Keras (<https://keras.io/api/applications/>), которую мы использовали для обучения модели;
- `MODEL_LABELS` — классы, которые мы хотим прогнозировать.

Применим эту конфигурацию:

```
kubectl apply -f tf-clothes.yaml
```

Подождите пару минут, прежде чем все будет готово, — используйте `kubectl get inferencesservice` для проверки статуса.

После того как развертывание произойдет (значение `READY` равно `True`), мы сможем его протестировать. Этим сейчас и займемся.

### 9.4.5. Тестирование преобразователя

Имея преобразователь, нам больше не нужно беспокоиться о подготовке изображения: достаточно просто отправить URL этого изображения. Код становится намного проще.

Вот как он выглядит:

```
import requests

data = {
    "instances": [
        {"url": "http://bit.ly/mlbookcamp-pants"},
    ]
}

url = 'https://clothing-model.default.kubeflow.mlbookcamp.com/v1/models/
clothing-model:predict'
result = requests.post(url, json=data).json()
```

URL сервиса остается прежним. Результат содержит прогнозы:

```
{'predictions': [ {'dress': -1.8682, 'hat': -4.7612, 'longsleeve': -2.3169,
  'outwear': -1.0625, 'pants': 9.8871, 'shirt': -2.8124, 'shoes': -3.6662,
  'shorts': 3.2003, 'skirt': -2.6023, 't-shirt': -4.8350} ]}
```

И это все! Теперь мы можем использовать модель.

### 9.4.6. Удаление кластера EKS

После экспериментов с EKS не забудьте отключить кластер. Для этого используйте `eksctl`:

```
eksctl delete cluster --name ml-bookcamp-eks
```

Чтобы убедиться, что кластер удален, можно проверить служебную страницу EKS в консоли AWS.

## 9.5. СЛЕДУЮЩИЕ ШАГИ

Вы изучили основы, которые позволяют подготовить классификационную модель, предназначенную для прогнозирования типа одежды. Мы рассмотрели

много материала, но в реальности его гораздо больше, чем мы смогли бы вместить в эту главу. Вы можете подробнее изучить данную тему, выполнив упражнения.

### 9.5.1. Упражнения

- Docker Compose — инструмент для запуска приложений с несколькими контейнерами. В нашем примере сервис Gateway должен взаимодействовать с моделью TF Serving; именно поэтому нам нужна возможность связать их. В этом и может помочь Docker Compose. Поэкспериментируйте с ним в рамках локального запуска TF Serving и Gateway.
- В этой главе мы использовали EKS от AWS. Изучению Kubernetes помогут эксперименты с Kubernetes локально. Используйте Minikube или Microk8s, чтобы воспроизвести пример с TF Serving и Gateway локально.
- Для всех экспериментов в этой главе мы использовали пространство имен Kubernetes по умолчанию. На практике мы обычно используем разные пространства имен для разных групп приложений. Узнайте больше о пространствах имен в Kubernetes, а затем разверните наши сервисы в другом пространстве имен. Например, вы можете использовать models.
- Преобразователи KFServing — эффективный инструмент для предварительной обработки данных. Мы не обсуждали их самостоятельную реализацию, а вместо этого использовали уже реализованный преобразователь. Чтобы больше узнать о них, реализуйте такой преобразователь самостоятельно.

### 9.5.2. Другие проекты

Есть множество проектов, над которыми можно поработать, чтобы лучше изучить Kubernetes и Kubeflow.

- В этой главе мы рассмотрели модель глубокого обучения. Она довольно сложная, и в итоге у нас получились два сервиса. Другие модели (до главы 7) менее сложны, и их размещение требует лишь простого приложения Flask. Вы можете развернуть модели из глав 2, 3 и 6 с помощью Flask и Kubernetes;
- KFServing можно использовать для развертывания других типов моделей, не только TensorFlow. Попробуйте развернуть модели Scikit-learn из глав 3 и 6.

## РЕЗЮМЕ

- TensorFlow Serving — система для развертывания моделей Keras и TensorFlow. Она использует gRPC и protobuf для взаимодействия и очень хорошо оптимизирована для предоставления сервисов.

- При использовании TensorFlow Serving нам обычно требуется отдельный компонент для подготовки пользовательского запроса в формате, ожидаемом моделью. Этот компонент скрывает сложность взаимодействия с TensorFlow Serving и облегчает использование модели клиентом.
- Чтобы развернуть что-то в Kubernetes, нам нужно создать развертывание и сервис. Развёртывание описывает, что именно должно быть развернуто: образ Docker и его конфигурация. Сервис располагается перед развертыванием и направляет запросы в отдельные контейнеры.
- Kubeflow и KFServing упрощают процесс развертывания: нам необходимо указать только местоположение модели, и они автоматически позаботятся о создании развертывания, сервиса и других важных аспектов.
- Преобразователи KFServing упрощают предварительную обработку данных, поступающих в модель, и последующую обработку результатов. Преобразователи позволяют больше не создавать специальный сервис шлюза для предварительной обработки.

# *Подготовка среды*

## **A.1. УСТАНОВКА PYTHON И ANACONDA**

Для проектов книги мы будем использовать Anaconda, дистрибутив Python, поставляемый с большинством необходимых пакетов машинного обучения, которые вам понадобятся: NumPy, SciPy, Scikit-learn, Pandas и многими другими.

### **A.1.1. Установка Python и Anaconda в Linux**

Инструкции в этом разделе будут работать независимо от того, устанавливаете вы Anaconda на удаленную машину или на свой ноутбук. Хотя мы тестировали его только на Ubuntu 18.04 LTS и 20.04 LTS, процесс должен нормально работать для большинства дистрибутивов Linux.

#### **ПРИМЕЧАНИЕ**

Для примеров в книге рекомендуется использовать Ubuntu Linux. Однако это не строгое требование, и у вас не должно возникнуть проблем с запуском примеров в других операционных системах. Если у вас нет компьютера с Ubuntu, то его можно арендовать онлайн в облаке. Более подробные инструкции можно получить в разделе A.6 «Аренда сервера на AWS».

Почти каждый дистрибутив Linux поставляется с установленным интерпретатором Python, но всегда полезно иметь отдельную установку Python, чтобы избежать проблем с системным Python. Использование Anaconda – отличный вариант: он устанавливается в каталог пользователя и не мешает системному Python.

Чтобы установить Anaconda, вам сначала нужно скачать его. Перейдите на <https://www.anaconda.com> и нажмите Get Starter. Затем выберите Download Anaconda Installer. Это должно привести вас на <https://www.anaconda.com/products/individual>.

Выберите 64-разрядный (x86) установщик и последнюю доступную версию – 3.8 на момент написания (рис. А.1).

Windows	MacOS	Linux
Python 3.8 64-Bit Graphical Installer (477 MB)  32-Bit Graphical Installer (409 MB)	Python 3.8 64-Bit Graphical Installer (440 MB)  64-Bit Command Line Installer (433 MB)	Python 3.8 64-Bit (x86) Installer (544 MB)  64-Bit (Power8 and Power9) Installer (285 MB)  64-Bit (AWS Graviton2 / ARM64) Installer (413 M)  64-bit (Linux on IBM Z & LinuxONE) Installer (292 M)

**Рис. А.1.** Загрузка установщика Linux для Anaconda

Далее скопируйте ссылку на установочный пакет. В нашем случае это был адрес [https://repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86\\_64.sh](https://repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86_64.sh).

#### ПРИМЕЧАНИЕ

Если доступна более новая версия Anaconda, то вам следует установить именно ее. Весь код без проблем будет работать на более новых версиях.

Теперь перейдите к терминалу, чтобы его загрузить:

```
wget https://repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86_64.sh
```

Затем установите его:

```
bash Anaconda3-2021.05-Linux-x86_64.sh
```

Прочитайте соглашение, введите `yes`, если вы принимаете его, а затем выберите место, куда вы хотите установить Anaconda. Вы можете использовать местоположение по умолчанию, но это необязательно.

Во время установки вас спросят, хотите ли вы инициализировать Anaconda. Наберите `yes`, и все будет сделано автоматически:

```
Do you wish the installer to initialize Anaconda3  
by running conda init? [yes|no]  
[no] >>> yes
```

Если вы не хотите, чтобы установщик проводил инициализацию, то можете сделать это вручную, добавив местоположение с двоичными файлами Anaconda в переменную `PATH`. Откройте файл `.bashrc` в домашнем каталоге и добавьте в конец эту строку:

```
export PATH=~/anaconda3/bin:$PATH
```

После завершения установки вы можете удалить установщик:

```
rm Anaconda3-2021.05-Linux-x86_64.sh
```

Затем откройте новую оболочку терминала. Если вы используете удаленный компьютер, то можете просто выйти из текущего сеанса, нажав `Ctrl+D`, а затем снова войти в систему, используя ту же команду `ssh`, что и ранее.

Теперь все должно сработать. Вы можете убедиться, что ваша система выбирает правильный двоичный файл, используя команду `which`:

```
which python
```

Если вы работаете на экземпляре EC2 от AWS, то должны увидеть что-то подобное этому:

```
/home/ubuntu/anaconda3/bin/python
```

Конечно, путь может быть другим, но это должен быть путь к установке Anaconda.

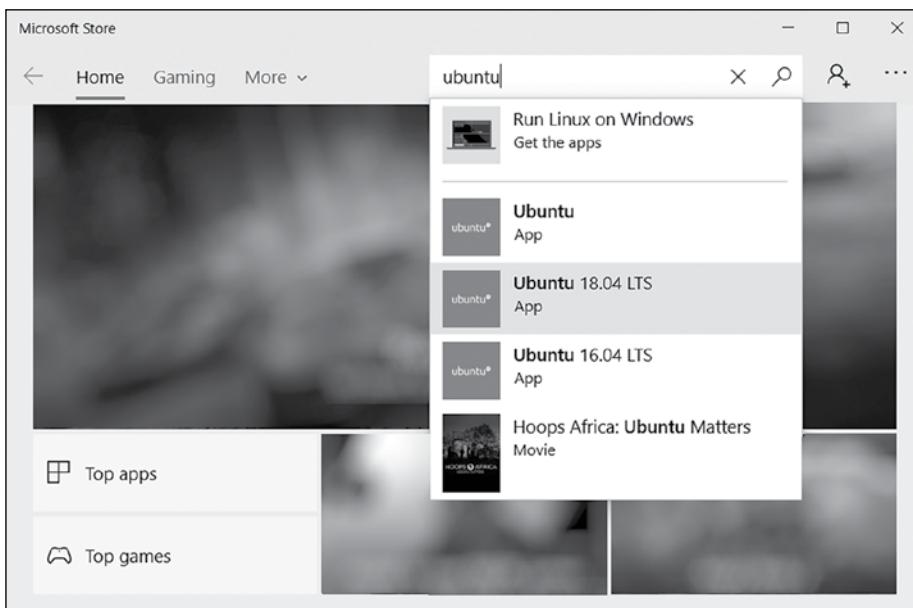
Теперь Python и Anaconda готовы к использованию.

## A.1.2. Установка Python и Anaconda в Windows

### Linux Subsystem для Windows

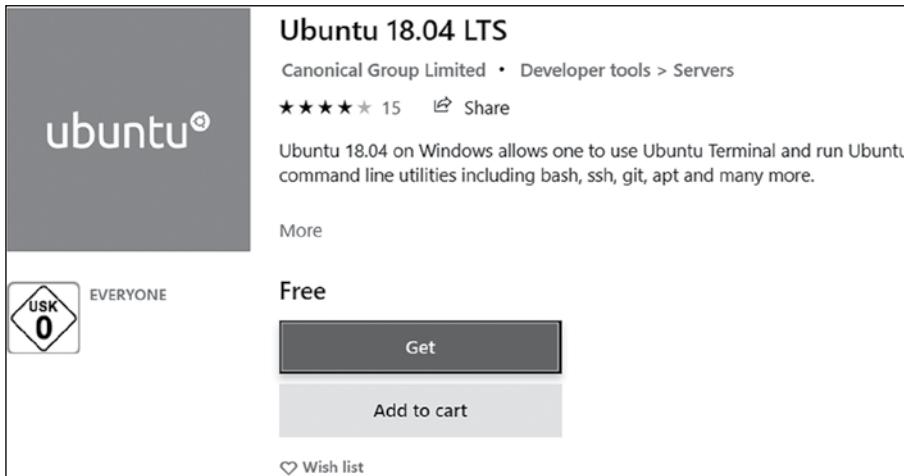
Рекомендуемый способ установки Anaconda в Windows — использовать Linux Subsystem для Windows.

Чтобы установить Ubuntu в Windows, откройте Microsoft Store и найдите `ubuntu` в поле поиска; затем выберите `Ubuntu 18.04 LTS` (рис. A.2).



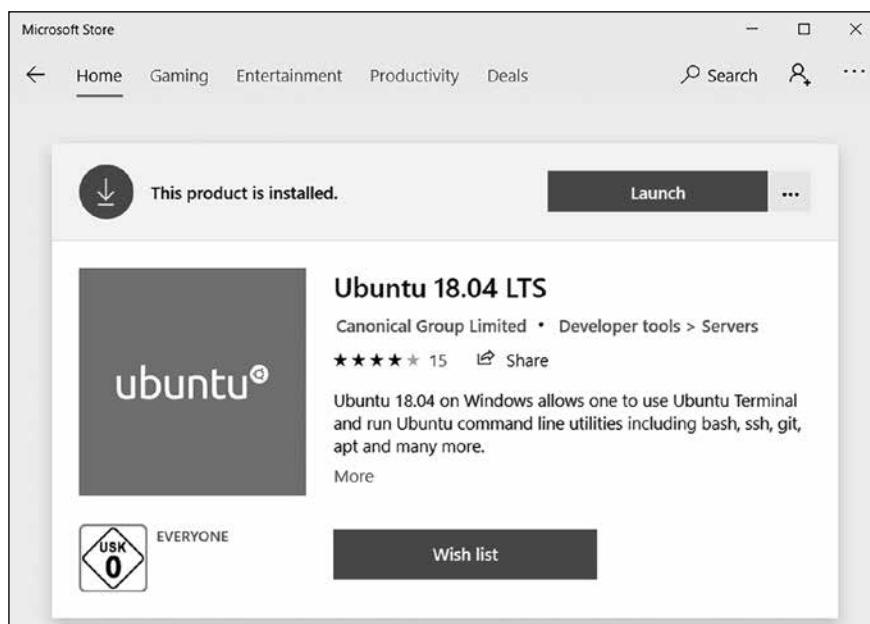
**Рис. А.2.** Используйте Microsoft Store для установки Ubuntu в Windows

Чтобы выполнить установку, просто нажмите Get в следующем окне (рис. А.3).



**Рис. А.3.** Чтобы установить Ubuntu 18.04 для Windows, нажмите Get

Сразу после установки мы сможем использовать ее, нажав кнопку Launch (рис. А.4).



**Рис. А.4.** Нажмите кнопку Launch, чтобы запустить терминал Ubuntu

При первом запуске вас попросят указать имя пользователя и пароль (рис. А.5). После этого терминал готов к использованию.

The screenshot shows a terminal window with a dark background and white text. It starts with a message about installing and creating a default UNIX user account. It asks for a new UNIX username ('Enter new UNIX username: ml') and password ('Enter new UNIX password:'). It then re-prompts for the password ('Retype new UNIX password:'). After the password is entered successfully ('passwd: password updated successfully'), it confirms the installation ('Installation successful!'). It provides instructions for running commands as root ('To run a command as administrator (user "root"), use "sudo <command>".'), and directs the user to see 'man sudo\_root' for details. Finally, it shows the user's name ('ml') and the prompt ('ml@LAPTOP-GV0371AS:~\$ \_') again.

**Рис. А.5.** Терминал Ubuntu, работающий в Windows

Теперь вы можете использовать терминал Ubuntu и следовать инструкциям для Linux, чтобы установить Anaconda.

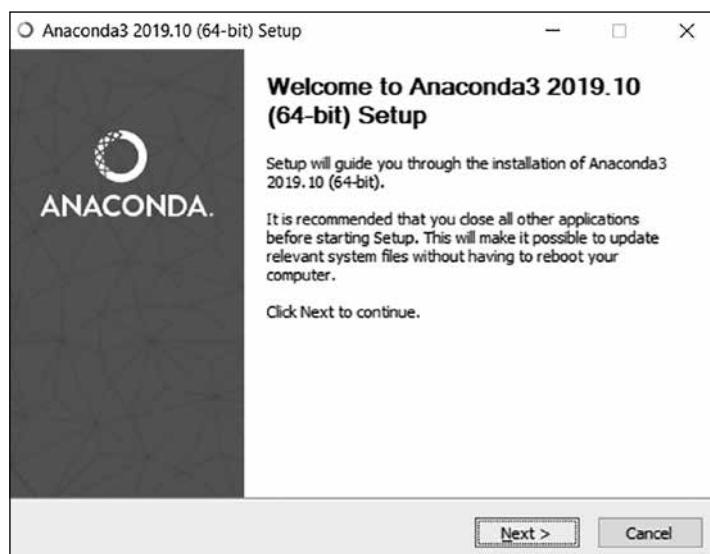
### Anaconda Windows Installer

В качестве альтернативы мы можем использовать Windows Installer для Anaconda. Сначала нам нужно скачать его из <https://anaconda.com/distribution> (рис. А.6). Перейдите в раздел Windows Installer и загрузите 64-разрядный графический установщик (или 32-разрядную версию, если вы используете более старый компьютер).



**Рис. А.6.** Загрузка Windows Installer для Anaconda

После загрузки программы установки просто запустите ее и следуйте руководству по установке (рис. А.7).



**Рис. А.7.** Установщик для Anaconda

Все довольно понятно, и у вас не должно возникнуть проблем с запуском. После успешной установки вы сможете запустить Anaconda, выбрав Anaconda Navigator в меню Пуск.

## A.1.3. Установка Python и Anaconda на macOS

Инструкции для macOS должны быть аналогичны инструкциям для Linux и Windows: выберите установщик с последней версией Python и запустите его.

# A.2. ЗАПУСК JUPYTER

## A.2.1. Запуск Jupyter в Linux

Как только Anaconda установлен, можно запускать Jupyter. Сначала вам нужно создать каталог, который Jupyter будет использовать для всех блокнотов:

```
mkdir notebooks
```

Затем с помощью `cd` перейдите в него, чтобы оттуда запустить Jupyter:

```
cd notebooks
```

Этот каталог будет использоваться для создания блокнотов. Теперь запустим Jupyter:

```
jupyter notebook
```

Этого должно быть достаточно, если нужно запустить Jupyter на локальном компьютере. Если же требуется запустить его на удаленном сервере, например на экземпляре EC2 из AWS, то вам необходимо добавить несколько дополнительных параметров командной строки:

```
jupyter notebook --ip=0.0.0.0 --no-browser
```

В этом случае вы должны указать:

- IP-адрес, который Jupyter будет использовать для приема входящих HTTP-запросов (`--ip=0.0.0.0`). По умолчанию он использует `localhost`, что означает, что доступ к сервису Notebook возможен только изнутри компьютера;
- параметр `--no-browser`, чтобы Jupyter не пытался с помощью браузера по умолчанию открывать URL с блокнотами. Конечно, на удаленном компьютере нет браузера, только терминал.

### ПРИМЕЧАНИЕ

В случае экземпляров EC2 на AWS вам также потребуется настроить правила безопасности, чтобы разрешить экземпляру получать запросы на порт 8888. Более подробную информацию можно найти в разделе А.6 «Аренда сервера на AWS».

## 394 Приложение А. Подготовка среды

Запустив данную команду, вы должны увидеть что-то наподобие этого:

```
[С 04:50:30.099 NotebookApp]
```

Чтобы получить доступ к блокноту, откройте этот файл в браузере:

```
file:///run/user/1000/jupyter/nbserver-3510-open.html
```

Или скопируйте и вставьте один из этих URL:

```
http://(ip-172-31-21-255 or 127.0.0.1):8888/
```

```
?token=670dfec7558c9a84689e4c3cdbb473e158d3328a40bf6bba
```

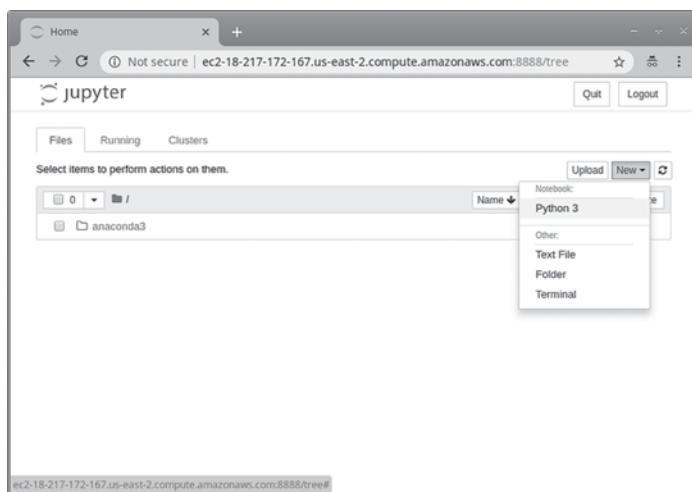
При запуске Jupyter генерирует случайный токен. Он понадобится вам для доступа к веб-странице. Это делается в целях безопасности, чтобы никто, кроме вас, не мог получить доступ к сервису Notebook.

Скопируйте URL из терминала и замените (ip-172-31-21-255 или 127.0.0.1) URL экземпляра. У вас должно получиться что-то наподобие этого: <http://ec2-18-217-172-167.us-east-2.compute.amazonaws.com:8888/?token=f04317713e74e65289fe5a43dac43d5bf164c144d05ce613>.

URL состоит из трех частей:

- DNS-имени экземпляра — если вы используете AWS, вы можете получить его из консоли AWS или с помощью AWS CLI;
- порта (8888, который является портом по умолчанию для сервиса блокнотов Jupyter);
- токена, который вы только что скопировали с терминала.

После этого вы сможете увидеть сервис Jupyter Notebook и создать новый блокнот (рис. А.8).



**Рис. А.8.** Сервис Jupyter Notebook. Теперь вы можете создать новый блокнот

Если вы используете удаленный компьютер, то при выходе из сеанса SSH сервис Jupyter Notebook будет остановлен. Внутренний процесс присоединен к сеансу SSH, и он будет завершен. Чтобы этого избежать, запустите service inside screen, инструмент для управления несколькими виртуальными терминалами:

```
screen -R jupyter
```

Эта команда попытается подключиться к экрану с именем jupyter, а если такого экрана не существует, то она создаст его.

Затем внутри экрана вы можете ввести ту же команду, чтобы запустить Jupyter Notebook:

```
jupyter notebook --ip=0.0.0.0 --no-browser
```

Убедитесь, что все работает, попытавшись получить к нему доступ из вашего браузера. После этого вы можете отсоединить экран, нажав Ctrl+A и затем D: сначала нажмите Ctrl+A, подождите немного, а затем нажмите D (для macOS сначала нажмите Ctrl+A, а затем нажмите Ctrl+D). Все, что выполняется внутри экрана, не привязано к текущему сеансу SSH, поэтому, когда вы отсоедините экран и выйдете из сеанса, процесс Jupyter продолжит выполняться.

Теперь вы можете отключиться от SSH (нажав Ctrl+D) и убедиться, что URL Jupyter все еще работает.

## A.2.2. Запуск Jupyter в Windows

Как и в случае с Python и Anaconda, если вы используете Linux Subsystem для Windows для установки Jupyter, то инструкции для Linux должны сработать и для Windows.

По умолчанию браузер не настроен для запуска в Linux Subsystem. Таким образом, для запуска Jupyter нам придется использовать следующую команду:

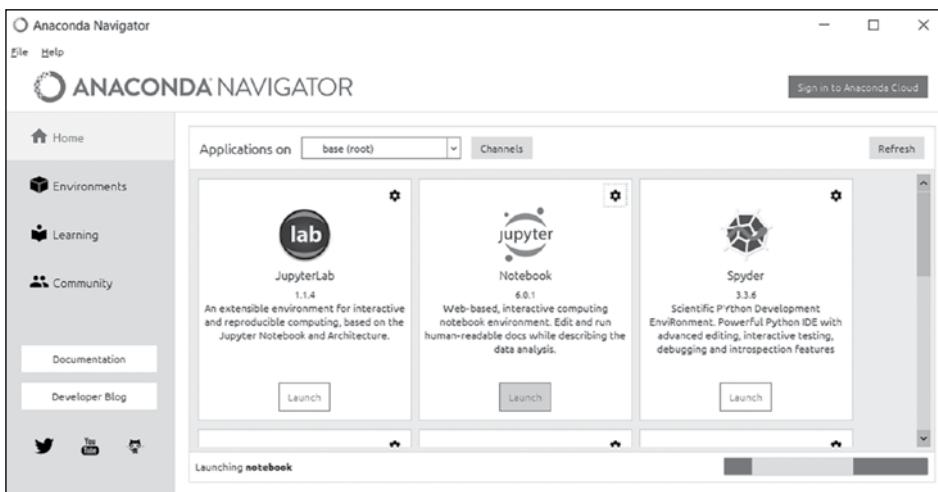
```
jupyter notebook --no-browser
```

В качестве альтернативы мы можем установить переменную BROWSER, чтобы она указывала на браузер из Windows:

```
export BROWSER='/mnt/c/Windows/explorer.exe'
```

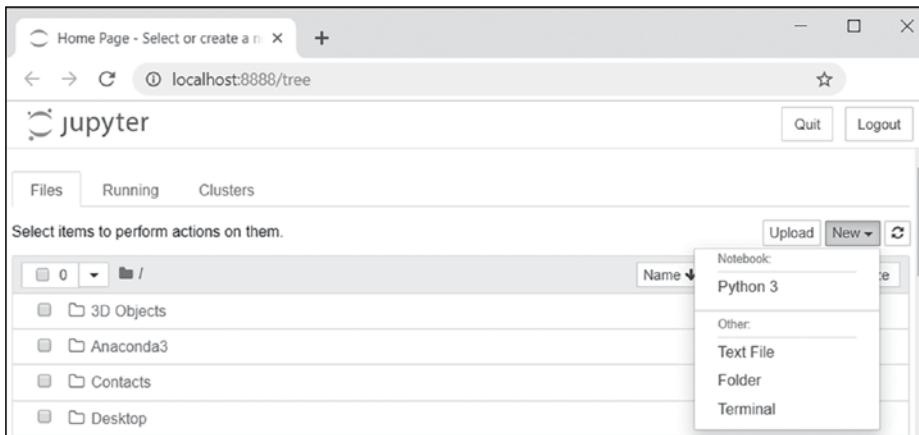
Однако если вы не использовали Linux Subsystem и установили Anaconda с помощью установщика Windows, то запуск сервиса Jupyter Notebook будет иметь различия.

Для начала нам нужно открыть навигатор Anaconda в меню Пуск. Как только он откроется, найдите Jupyter на вкладке Applications и нажмите Launch (рис. A.9).



**Рис. А.9.** Чтобы запустить сервис Jupyter Notebook, найдите Jupyter на вкладке приложений и нажмите Launch

После успешного запуска сервиса браузер с Jupyter должен открыться автоматически (рис. А.10).



**Рис. А.10.** Сервис Jupyter Notebook, запущенный с использованием Anaconda Navigator

### A.2.3. Запуск Jupyter на macOS

Инструкции для Linux также должны работать для macOS без каких-либо дополнительных изменений.

## A.3. УСТАНОВКА KAGGLE CLI

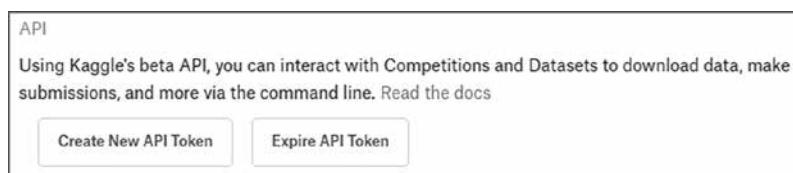
Kaggle CLI – интерфейс командной строки для доступа к платформе Kaggle, которая содержит данные из соревнований Kaggle и наборы данных Kaggle.

Вы можете установить его с помощью pip:

```
pip install kaggle --upgrade
```

Затем его необходимо настроить. Сначала нужно получить учетные данные от Kaggle. Для этого перейдите в свой профиль Kaggle (создайте его, если у вас его еще нет), расположенный по адресу <https://www.kaggle.com/<username>/account>. URL будет выглядеть примерно так: <https://www.kaggle.com/agrigorev/account>.

В разделе API нажмите Create New API Token (рис. A.11).



**Рис. А.11.** Чтобы сгенерировать токен API для использования из командной строки Kaggle, нажмите Create New API Token на странице вашей учетной записи Kaggle

Это позволит загрузить файл с именем `kaggle.json`, который представляет собой файл JSON с двумя полями: `username` и `key`. Если вы настраиваете Kaggle CLI на том же компьютере, который использовали для загрузки файла, то вам следует просто переместить этот файл в папку, где его ожидает увидеть Kaggle CLI:

```
mkdir ~/.kaggle
mv kaggle.json ~/.kaggle/kaggle.json
```

Если же вы настраиваете его на удаленном компьютере, например на экземпляре EC2, то вам необходимо скопировать содержимое этого файла и вставить его в терминал. Откройте файл с помощью nano (таким образом создастся файл, если он не существует):

```
mkdir ~/.kaggle
nano ~/.kaggle/kaggle.json
```

Вставьте туда содержимое файла `kaggle.json`, который вы скачали. Сохраните файл, нажав `Ctrl+O`, и выйдите из nano, нажав `Ctrl+X`.

Теперь проверьте, что все работает, попытавшись перечислить доступные наборы данных:

```
kaggle datasets list
```

Вы также можете проверить, что можно загружать наборы данных, попробовав набор данных из главы 2:

```
kaggle datasets download -d CooperUnion/cardataset
```

Команда должна загрузить файл под названием cardataset.zip.

## A.4. ДОСТУП К ИСХОДНОМУ КОДУ

Мы сохранили исходный код для этой книги на GitHub, платформе для размещения исходного кода. Вы можете найти его здесь: <https://github.com/alexeygrigorev/mlbookcamp-code>.

GitHub использует Git для управления кодом, поэтому вам понадобится клиент Git для доступа к коду для этой книги.

Git предустановлен во всех основных дистрибутивах Linux. Например, AMI, который мы использовали для создания экземпляра с Ubuntu на AWS, уже содержит его.

Если в вашем дистрибутиве нет Git, то его можно легко установить. Например, для дистрибутивов на базе Debian (таких как Ubuntu) вам необходимо выполнить следующую команду:

```
sudo apt-get install git
```

В macOS для использования Git вам необходимо установить средства командной строки или в качестве альтернативы скачать установщик по адресу <https://sourceforge.net/projects/git-osx-installer/>.

Для Windows вы можете скачать Git по адресу <https://git-scm.com/download/win>.

Как только Git установлен, вы сможете его использовать для получения кода книги. Чтобы получить доступ, вам необходимо выполнить следующую команду:

```
git clone https://github.com/alexeygrigorev/mlbookcamp-code.git
```

Теперь вы можете запустить Jupyter Notebook:

```
cd mlbookcamp-code  
jupyter notebook
```

Если у вас нет Git и вы не хотите его устанавливать, то можете получить доступ к коду и без него. Вы можете загрузить последнюю версию кода в zip-архиве и распаковать его. В Linux вы можете выполнить для этого следующие команды:

```
wget -O mlbookcamp-code.zip \  
      https://github.com/alexeygrigorev/mlbookcamp-code/archive/master.zip  
unzip mlbookcamp-code.zip  
rm mlbookcamp-code.zip
```

Вы также можете просто воспользоваться своим браузером: введите URL, скачайте zip-архив и извлеките содержимое.

## A.5. УСТАНОВКА DOCKER

В главе 5 мы используем Docker для упаковки нашего приложения в изолированный контейнер. Его довольно легко установить.

### A.5.1. Установка Docker в Linux

Эти шаги основаны на официальных инструкциях для Ubuntu с сайта Docker (<https://docs.docker.com/engine/install/ubuntu/>).

Сначала нам нужно установить все необходимые компоненты:

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common
```

Далее мы добавляем репозиторий с двоичными файлами Docker:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/
ubuntu $(lsb_release -cs) stable"
```

Теперь мы можем перейти к установке:

```
sudo apt-get update
sudo apt-get install docker-ce
```

Наконец, если мы хотим выполнять команды Docker без sudo, нам нужно добавить нашего пользователя в группу пользователей `docker`:

```
sudo adduser $(whoami) docker
```

Теперь следует перезагрузить вашу систему. В случае EC2 или другой удаленной машины достаточно просто выйти из системы и снова войти.

Чтобы проверить, все ли работает нормально, запустите контейнер `hello-world`:

```
docker run hello-world
```

Вы должны увидеть сообщение о том, что все работает:

```
Hello from Docker!
```

Это сообщение показывает, что ваша установка, по всей видимости, работает правильно.

## A.5.2. Установка Docker в Windows

Чтобы установить Docker в Windows, вам необходимо скачать установщик с официального сайта (<https://hub.docker.com/editions/community/docker-ce-desktop-windows/>) и просто следовать инструкциям.

## A.5.3. Установка Docker на macOS

Как и в случае с Windows, установка Docker на macOS проста: сначала скачайте установщик с официального сайта (<https://hub.docker.com/editions/community/docker-ce-desktop-mac/>), а затем следуйте инструкциям.

## A.6. АРЕНДА СЕРВЕРА НА AWS

Использование облачного сервиса – самый простой способ получить удаленную машину, с помощью которой можно выполнять примеры в книге.

В настоящее время существует довольно много вариантов, включая поставщиков облачных вычислений, таких как Amazon Web Services (AWS), Google Cloud Platform, Microsoft Azure и Digital Ocean. Вместо того чтобы арендовать сервер на длительное время, в облаке вы можете использовать его в течение короткого периода и, как правило, платить за часы, минуты или даже за секунды. Вы можете выбрать машину, наиболее подходящую для ваших нужд с точки зрения вычислительной мощности (количество процессоров или графических процессоров) и оперативной памяти.

К тому же можно арендовать выделенный сервер на более длительный срок и платить ежемесячно. Если вы намерены использовать сервер в течение длительного времени (скажем, шести месяцев или более), то аренда выделенного сервера обойдется дешевле. Hetzner.com может оказаться хорошим вариантом в этом случае. Он также предлагает серверы с графическими процессорами.

Чтобы вам было проще настроить среду со всеми необходимыми библиотеками для книги, мы приведем инструкции по настройке машины EC2 (Elastic Compute Cloud) на AWS. EC2 является частью AWS и позволяет арендовать сервер любой конфигурации на любой срок.

### ПРИМЕЧАНИЕ

Я не связан с Amazon или AWS и решил использовать его в книге, поскольку на момент написания это наиболее часто используемый облачный провайдер.

Если у вас нет учетной записи AWS или вы только недавно ее создали, то у вас есть право на бесплатный уровень: вам доступен 12-месячный пробный

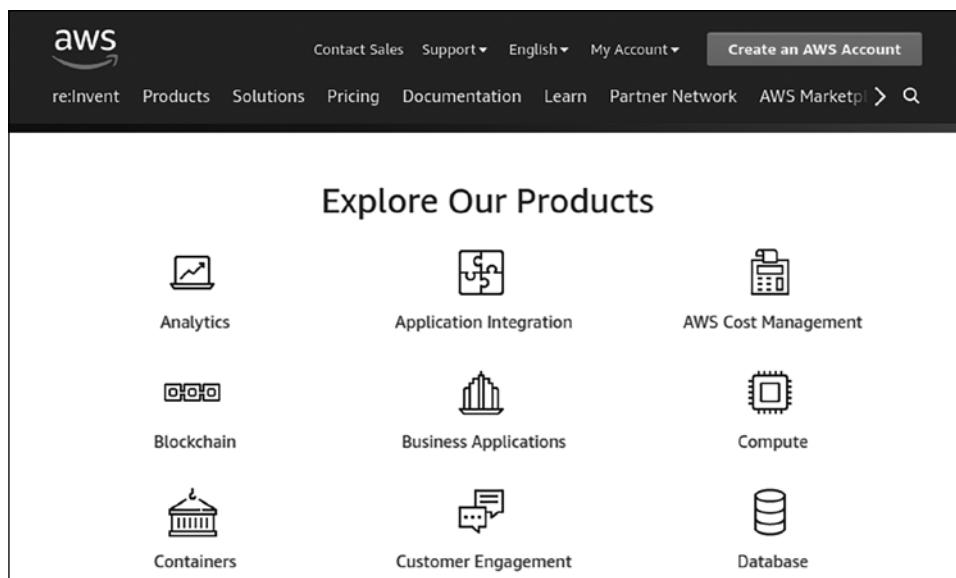
период, в течение которого вы можете бесплатно ознакомиться с большинством продуктов AWS. Мы стараемся использовать бесплатный уровень, когда это возможно, и специально упоминаем, если что-то выходит за его пределы.

Обратите внимание, что инструкции в этом разделе необязательны и вам не обязательно использовать AWS или любое другое облако.

Код должен работать на любой машине с Linux, так что если у вас есть ноутбук с Linux, этого уже достаточно для работы с книгой. Компьютер Mac или Windows также должен подойти, но на этих платформах мы не тестировали код особенно тщательно.

### A.6.1. Регистрация на AWS

Первое, что вам нужно сделать, — создать учетную запись. Чтобы сделать это, перейдите на <https://aws.amazon.com> и нажмите кнопку *Create an AWS Account* (рис. A.12).



**Рис. A.12.** Чтобы создать учетную запись, нажмите *Create an AWS Account* на главной странице AWS

#### ПРИМЕЧАНИЕ

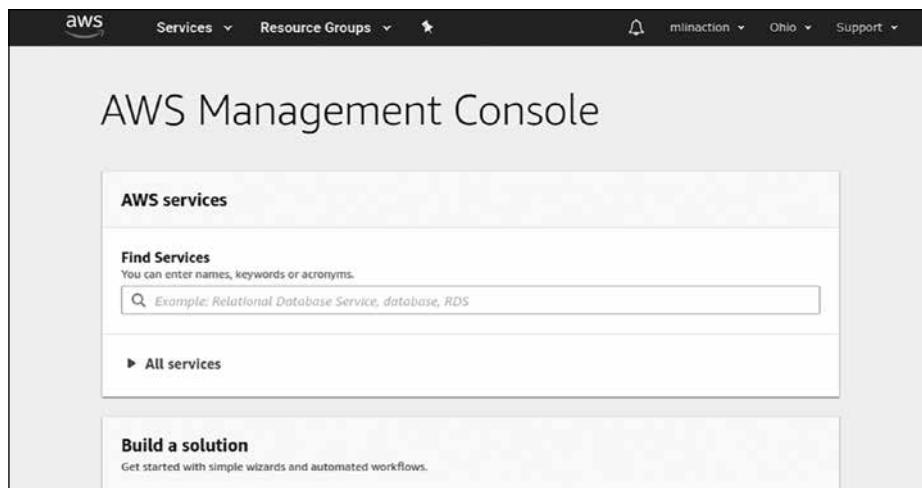
Это приложение было написано в октябре 2019 года, и скриншоты были сделаны в то же время. Пожалуйста, имейте в виду, что содержимое сайта AWS и внешний вид консоли управления могут измениться.

Следуйте инструкциям и заполните необходимые данные. Процесс аналогичен процессу регистрации на любом сайте.

**ПРИМЕЧАНИЕ**

Пожалуйста, имейте в виду, что AWS попросит вас предоставить данные банковской карты в процессе регистрации.

Как только вы завершите регистрацию и подтвердите свою учетную запись, вы должны увидеть главную страницу — консоль управления AWS (рис. A.13).



**Рис. A.13.** Консоль управления AWS — это начальная страница для AWS

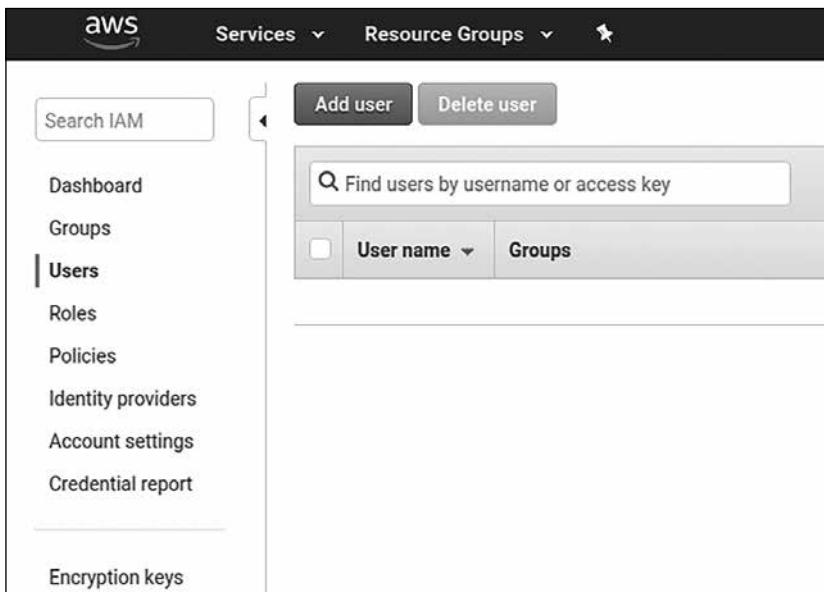
Поздравляю! Вы только что создали учетную запись root. Однако не рекомендуется использовать root для чего-либо: у нее очень широкие полномочия, которые позволяют делать все, что угодно, в вашей учетной записи AWS.

Как правило, вы используете учетную запись root для создания менее привилегированных учетных записей, а затем используете их для своих текущих задач.

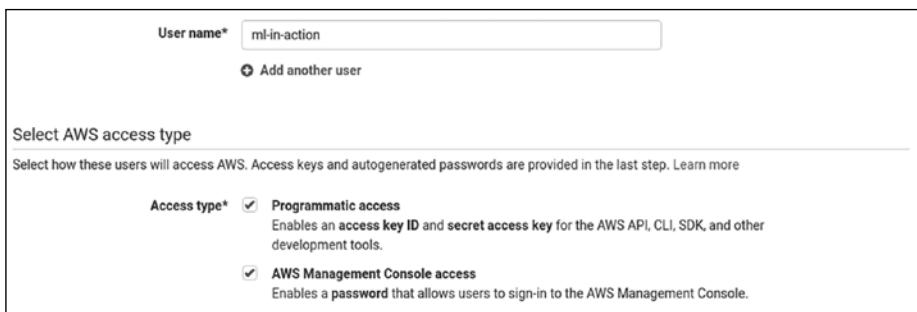
Чтобы создать такую учетную запись, введите IAM в поле Find Services и нажмите на этот элемент в выпадающем списке. Выберите Users в меню слева и нажмите Add User (рис. A.14).

Теперь вам просто нужно следовать инструкциям и отвечать на вопросы. В какой-то момент вас спросят о типе доступа: вам нужно будет выбрать как Programmatic Access, так и AWS Management Console Access (рис. A.15). Для работы

с AWS мы будем использовать как интерфейс командной строки (CLI), так и веб-интерфейс.



**Рис. А.14.** Добавление пользователя в сервис AWS Identity and Access Management (IAM)



**Рис. А.15.** Для работы с AWS мы будем использовать как веб-интерфейс, так и интерфейс командной строки, поэтому необходимо выбрать оба типа доступа

На шаге Set Permissions вы указываете, что сможет делать этот новый пользователь. Вам нужно, чтобы он имел полные привилегии, поэтому выберите Attach Existing Policies Directly вверху и AdministratorAccess в списке политик (рис. А.16).

The screenshot shows the 'Set permissions' interface in the AWS IAM console. At the top, there are three buttons: 'Add user to group', 'Copy permissions from existing user', and 'Attach existing policies directly'. Below these is a 'Create policy' button. A search bar labeled 'Search' is followed by a link 'Showing 442 results'. The main area is a table with columns: Policy name, Type, Used as, and Description. One row is selected, showing 'AdministratorAccess' as a 'Job function' type with 'None' used as. The description states: 'Provides full access to AWS services and resources'. There are also other rows for 'AlexaForBusinessDeviceAccess', 'AlexaForBusinessFullAccess', and 'AlexaForBusinessResourceAccess'. At the bottom right are 'Cancel', 'Previous', 'Next', and 'Tags' buttons.

**Рис. А.16.** Выберите политику AdministratorAccess, чтобы разрешить новому пользователю доступ ко всему в AWS

В качестве следующего шага система спросит вас о тегах — пока это можно спокойно игнорировать. Теги необходимы компаниям, где несколько человек работают в одной учетной записи AWS, в основном для целей управления расходами, поэтому они не требуются для проектов этой книги.

В итоге, когда вы успешно создадите нового пользователя, мастер предложит вам скачать учетные данные (рис. А.17). Скачайте их и храните в безопасности; вы используете их позже при настройке AWS CLI.

The screenshot shows a 'Success' dialog box. It contains a message: 'You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.' Below this is a link: 'Users with AWS Management Console access can sign-in at: https://387546586013.signin.aws.amazon.com/console'. At the bottom left is a 'Download .csv' button. A table below lists the newly created user 'ml-in-action' with details: Access key ID (AKIAVU04TTO04RXEEUP5), Secret access key (\*\*\*\*\* Show), and Email login instructions (Send email). A 'Close' button is at the bottom right.

**Рис. А.17.** Сведения о вновь созданном пользователе. Вы можете просмотреть URL для входа и скачать учетные данные для программного доступа

Чтобы получить доступ к консоли управления, вы можете воспользоваться ссылкой, сгенерированной для вас AWS. Она появляется в окне Success и следует шаблону: <https://<accountid>.signin.aws.amazon.com/console>.

Возможно, стоит добавить эту ссылку в закладки. Как только AWS подтвердит учетную запись (что может занять некоторое время), вы можете использовать ее для входа в систему: просто укажите имя пользователя и пароль, которые указывали при создании пользователя.

Теперь вы можете начать пользоваться услугами AWS. Самое главное, можете создать машину EC2.

## A.6.2. Доступ к платежной информации

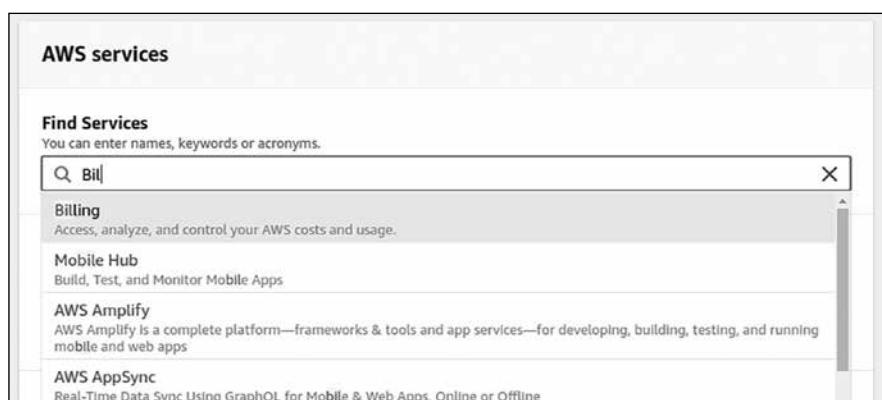
При использовании поставщика облачных услуг с вас обычно взимается посекундная плата: за каждую секунду работы с определенным сервисом AWS вы платите по заранее установленному тарифу. В конце каждого месяца вы получаете счет, который обычно обрабатывается автоматически. Деньги снимаются с банковской карты, которую вы привязали к аккаунту AWS.

### ВАЖНО

Несмотря на то что для выполнения большинства примеров в книге мы действуем бесплатный уровень, вам следует периодически проверять страницу счетов, чтобы убедиться, что вы случайно не воспользовались платными услугами.

Чтобы понять, сколько вам придется заплатить в конце месяца, вы можете зайти на страницу счетов AWS.

Если вы используете учетную запись root (которую создали первой), просто введите **Billing** на главной странице консоли AWS, чтобы перейти на страницу выставления счетов (рис. A.18).



**Рис. A.18.** Чтобы перейти на страницу выставления счетов, введите Billing в поле поиска быстрого доступа

Если вы попытаетесь получить доступ к той же странице из учетной записи пользователя (или пользователя IAM — той, которую мы создали после root), то заметите, что это запрещено. Чтобы исправить это, вам нужно:

- разрешить доступ к странице выставления счетов всем пользователям IAM;
- предоставить пользователю IAM разрешение на доступ к странице выставления счетов.

Разрешить всем пользователям IAM доступ к странице выставления счетов просто: перейдите в My Account (рис. А.19, А), перейдите в раздел IAM User and Role Access to Billing Information и нажмите Edit (рис. А.19, Б), а затем установите флажок Activate IAM Access и нажмите Update (рис. А.19, В).



**А. Чтобы разрешить пользователям IAM доступ к платежной информации, нажмите My Account**

A screenshot of the 'IAM User and Role Access to Billing Information' settings page. The title is 'IAM User and Role Access to Billing Information' with an 'Edit' button. Below the title, there's a descriptive text: 'You can give IAM users and federated users with roles permissions to access billing information. This includes access to Account Settings, Payment Methods, and Report pages. You control which users and roles can see billing information by creating IAM policies. For more information, see Controlling Access to Your Billing Information.' A note below says 'IAM user/role access to billing information is deactivated.' with a small link.

**Б. В настройках My Account найдите раздел IAM User and Role Access to Billing Information и нажмите Edit**

A screenshot of the same 'IAM User and Role Access to Billing Information' settings page as above, but with a checked checkbox next to 'Activate IAM Access'. Below the checkbox are two buttons: 'Update' and 'Cancel'.

**В. Установите флажок опцию Activate IAM Access и нажмите Update**

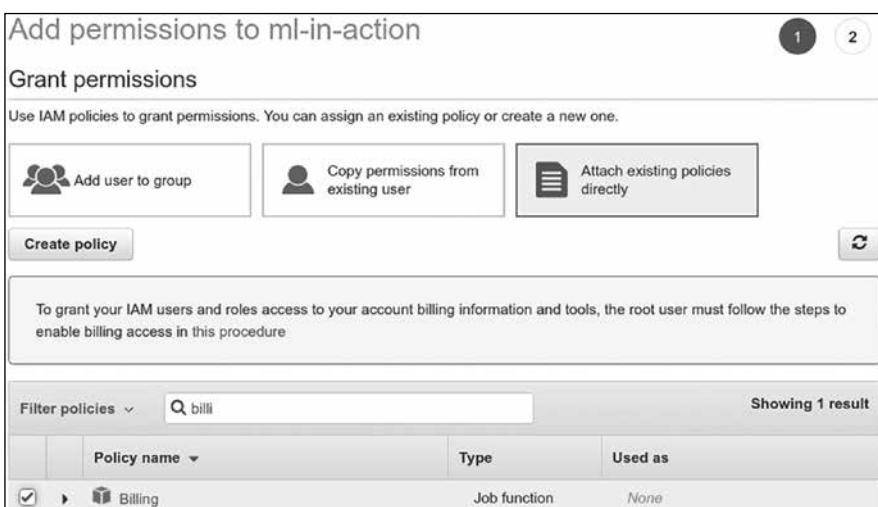
**Рис. А.19.** Предоставление доступа к платежной информации пользователям IAM

После этого перейдите в сервис IAM, найдите пользователя IAM, которого мы ранее создали, и нажмите его. Далее нажмите кнопку Add permissions (рис. А.20).



**Рис. A.20.** Чтобы разрешить пользователю IAM доступ к платежной информации, необходимо добавить для этого специальные разрешения. Для этого нажмите кнопку Add permissions

Затем закрепите за пользователем существующую политику выставления счетов (рис. A.21).



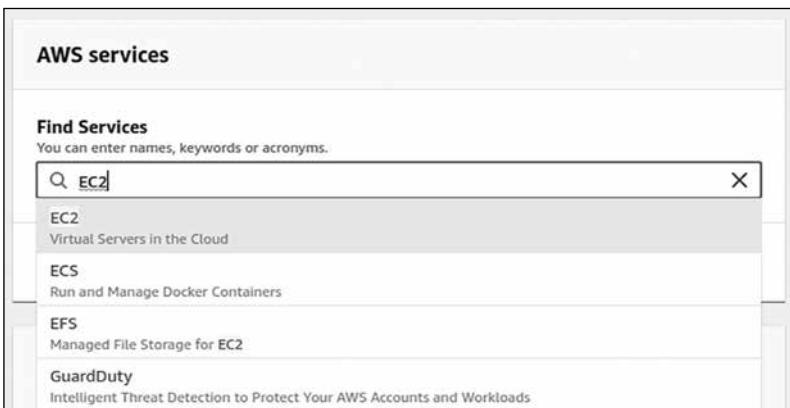
**Рис. A.21.** После нажатия Add permissions выберите опцию Attach existing policies directly и выберите Billing в списке

После этого пользователь IAM должен получить возможность доступа к платежной информации.

### A.6.3. Создание экземпляра EC2

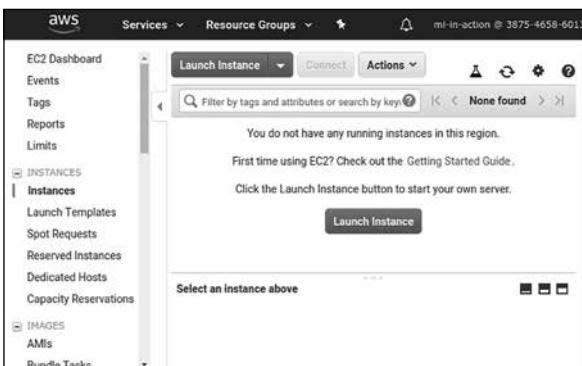
EC2 — это сервис аренды компьютера у AWS. С его помощью можно создать Linux-машину для проектов, описанных в книге. Для этого сначала перейдите на страницу EC2 в AWS. Самый простой способ сделать это — ввести EC2 в поле

Find Services на домашней странице AWS Management Console; затем выберите EC2 из выпадающего списка и нажмите Enter (рис. А.22).



**Рис. А.22.** Чтобы перейти на страницу службы EC2, введите EC2 в поле Find Services на главной странице AWS Management Console и нажмите Enter

На странице EC2 выберите Instances в меню слева, а затем нажмите Launch Instance (рис. А.23).



**Рис. А.23.** Чтобы создать экземпляр EC2, выберите Instances в меню слева и нажмите Launch Instance

Это приведет вас к форме из шести шагов. Первый шаг — указать AMI (образ машины Amazon), который вы будете использовать для экземпляра. Мы рекомендуем Ubuntu: это один из самых популярных дистрибутивов Linux, и мы использовали его для всех примеров в данной книге. Другие образы также должны работать нормально, но мы их не тестировали.

На момент написания статьи доступен Ubuntu Server 20.04 LTS (рис. A.24), поэтому используйте его. Отыщите его в списке и затем нажмите Select.



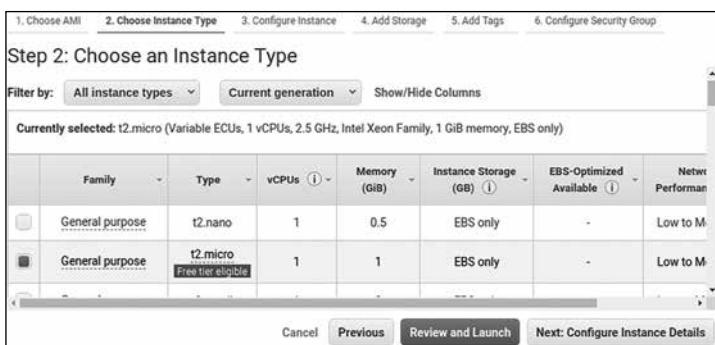
**Рис. A.24.** Ваш экземпляр будет основан на Ubuntu Server 18.04 LTS

Вам следует обратить внимание на идентификатор AMI: в этом примере это ami-0a8e758f5e873d1c1, но у вас он может быть другим в зависимости от региона AWS и версии Ubuntu.

#### ПРИМЕЧАНИЕ

Данный AMI доступен в бесплатном уровне; это значит, что если вы используете бесплатный уровень для тестирования AWS, то с вас не будет взиматься плата за использование этого AMI.

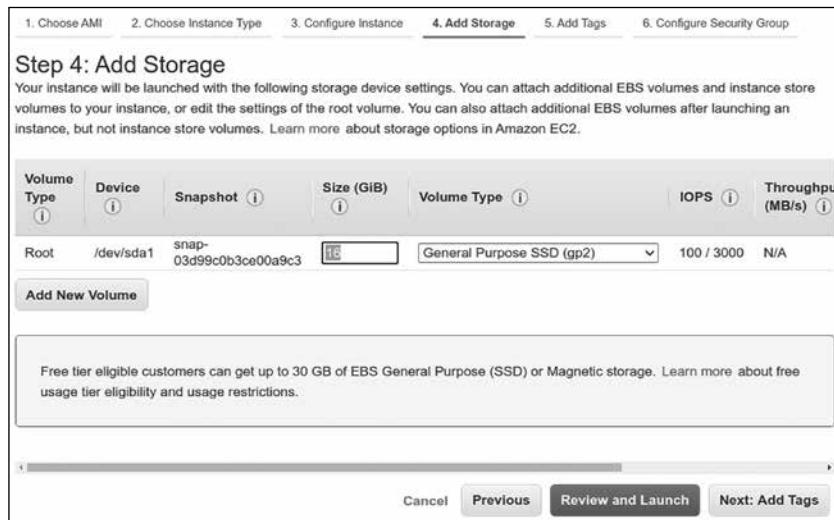
После этого вам нужно выбрать тип экземпляра. Существует множество вариантов с разным количеством ядер процессора и разным объемом оперативной памяти. Если вы хотите оставаться на бесплатном уровне, то выберите t2.micro (рис. A.25). Это довольно маленькая машина: у нее всего один процессор и 1 Гбайт оперативной памяти. Конечно, это не лучший вариант с точки зрения вычислительной мощности, но этого должно хватить для многих проектов в данной книге.



**Рис. A.25.** t2.micro — довольно маленький экземпляр, имеющий всего один процессор и 1 Гбайт оперативной памяти, но им можно пользоваться бесплатно

## 410 Приложение А. Подготовка среды

Следующий шаг — настройка сведений об экземпляре. Здесь вам не нужно ничего менять, и вы можете просто перейти к следующему шагу: добавлению хранилища (рис. А.26).



**Рис. А.26.** Четвертый шаг создания экземпляра EC2 в AWS: добавление хранилища. Измените размер на 16 Гбайт

Здесь вы указываете, сколько места вам требуется на экземпляре. Предложенного по умолчанию значения 8 Гбайт недостаточно, поэтому выберите 16 Гбайт. Этого должно хватить для большинства проектов, которые мы будем выполнять в данной книге. Когда изменение будет внесено, нажмите Next: Add Tags.

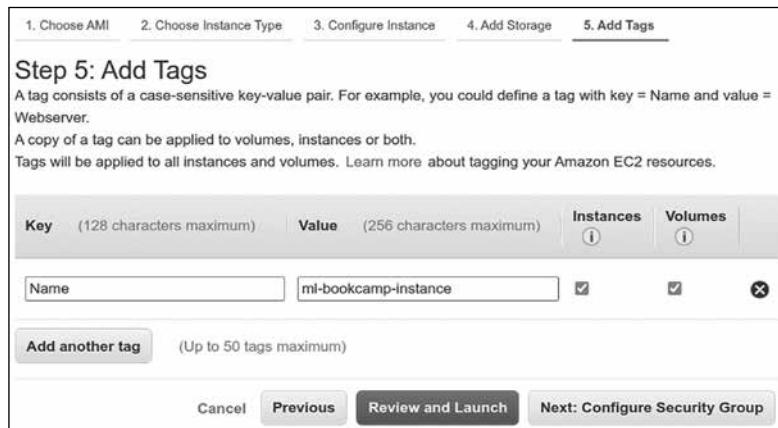
На следующем шаге вы добавите теги к своему новому экземпляру. Единственный тег, который здесь следует добавить, — это Name, позволяющий присвоить экземпляру удобочитаемое имя.

Добавьте имя ключа и значение `ml-bookcamp-instance` (или любое другое имя, которое вам нравится), как показано на рис. А.27.

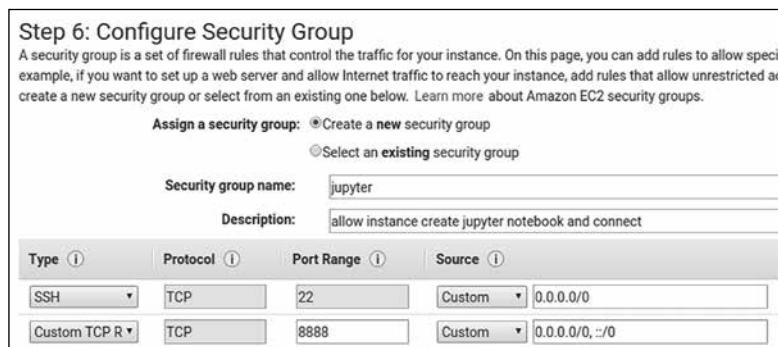
Следующий шаг довольно важен: выбор группы безопасности. Он позволяет настроить сетевой брандмауэр и указать, как можно получить доступ к экземпляру и какие порты будут при этом открыты. Вам понадобится размещать на экземпляре Jupyter Notebook, поэтому необходимо убедиться, что его порт открыт и вы можете войти на удаленный компьютер.

Поскольку в вашей учетной записи AWS еще нет групп безопасности, вам следует создать новую прямо сейчас: выберите `Create a New Security Group` и присвойте ей имя `jupyter` (рис. А.28). Для подключения к экземпляру со своих компьютеров

вы будете использовать SSH, поэтому следует убедиться, что SSH-соединения разрешены. Чтобы их разрешить, выберите SSH в раскрывающемся списке Type в первой строке.



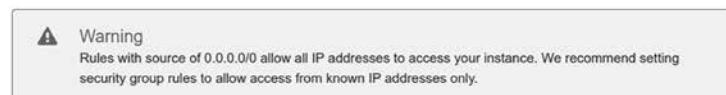
**Рис. A.27.** Единственный тег, который вы, возможно, захотите указать на шаге 5, — это Name: он позволяет присвоить экземпляру удобочитаемое имя



**Рис. A.28.** Создание группы безопасности для запуска Jupyter Notebook на экземплярах EC2

Обычно сервис Jupyter Notebook работает на порте 8888, поэтому вам необходимо добавить пользовательское правило TCP, чтобы к порту 8888 можно было получить доступ из любой точки Интернета.

Может появиться предупреждение о том, что это небезопасно (рис. A.29). Для нас это не проблема, поскольку мы не запускаем на экземплярах ничего критичного. Внедрение надлежащей безопасности не является тривиальным и выходит за рамки книги.



**Рис. А.29.** AWS предупреждает нас о том, что добавленные нами правила не являются достаточно строгими. В нашем случае это не проблема, и можно спокойно игнорировать предупреждение

В следующий раз при создании экземпляра вы сможете повторно использовать эту группу безопасности вместо того, чтобы создавать новую. Выберите **Select an Existing Security Group** и найдите ее в списке (рис. А.30).



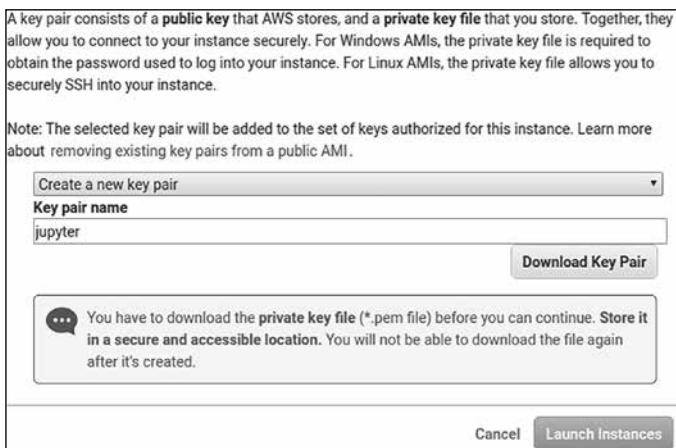
**Рис. А.30.** При создании экземпляра также возможно назначить экземпляру существующую группу безопасности

Настройка группы безопасности — это последний шаг. Убедитесь, что все в порядке, и нажмите **Review and Launch**.

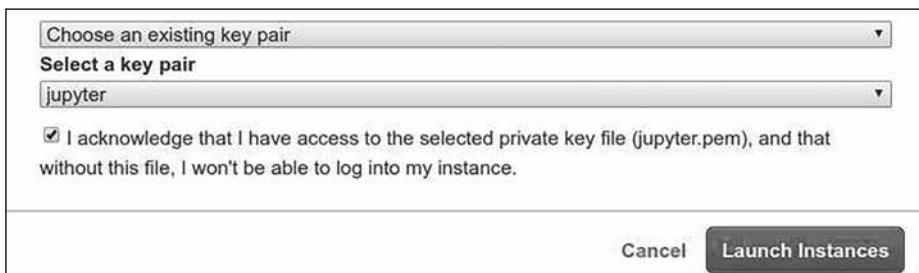
AWS пока не позволит вам запустить экземпляр: вам все еще требуется настроить SSH-ключи для входа в экземпляр. Поскольку ваша учетная запись AWS совсем новая и у нее еще нет ключей, вам необходимо создать новую пару. Выберите **Create a New Key Pair** из выпадающего списка и дайте ей имя **jupyter** (рис. А.31).

Нажмите **Download Key Pair** и сохраните файл где-нибудь на своем компьютере. Удостоверьтесь, что сможете получить доступ к этому файлу позже; это важно для возможности подключения к экземпляру.

В следующий раз при создании экземпляра вы сможете повторно использовать этот ключ. Выберите **Choose an Existing Key Pair** в первом выпадающем списке, выберите ключ, который хотите использовать. После этого установите флажок, чтобы подтвердить, что ключ все еще у вас (рис. А.32).

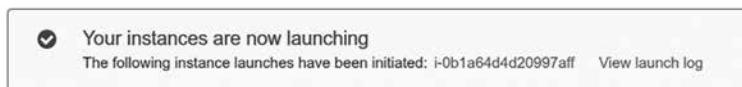


**Рис. А.31.** Чтобы иметь возможность использовать SSH для входа в экземпляр, вам необходимо создать пару ключей



**Рис. А.32.** Вы также можете использовать существующий ключ при создании экземпляра

Вот теперь можно запустить экземпляр, нажав кнопку **Launch Instances**. Вы должны увидеть подтверждение того, что все в порядке и экземпляр запускается (рис. А.33).



**Рис. А.33.** AWS сообщает нам, что все прошло хорошо и теперь экземпляр запускается

В этом сообщении содержится идентификатор экземпляра. В нашем случае это `i-0b1a64d4d20997aff`. Вы можете сразу нажать на него и увидеть подробную

информацию об экземпляре (рис. А.34). Поскольку вы хотите использовать SSH для подключения к своему экземпляру, вам необходимо получить общедоступное DNS-имя. Его можно найти на вкладке Description.

Instance ID	Public DNS (IPv4)
i-0b1a64d4d20997aff	ec2-18-224-137-4.us-east-2.compute.amazonaws.com

**Рис. А.34.** Сведения о вновь созданном экземпляре. Чтобы подключиться к нему по SSH, вам нужно общедоступное DNS-имя

## A.6.4. Подключение к экземпляру

В предыдущем подразделе вы создали экземпляр на EC2. Теперь вам нужно войти в этот экземпляр, чтобы установить все необходимое программное обеспечение. Для этого вы используете SSH.

### Подключение к экземпляру из Linux

У вас уже имеется общедоступное DNS-имя вашего экземпляра. В нашем примере это ec2-18-191-156-172.us-east-2.compute.amazonaws.com. В вашем случае имя будет другим: первая часть имени (ec2-18-191-156-172) зависит от IP-адреса, который получает экземпляр, а вторая (us-east-2) — от региона запуска. Это имя вам понадобится, чтобы использовать SSH для входа в экземпляр.

При первом использовании ключа, который вы скачали из AWS, необходимо удостовериться, что права доступа к файлу установлены правильно. Выполните следующую команду:

```
chmod 400 jupyter.pem
```

Теперь вы можете использовать ключ для входа в экземпляр:

```
ssh -i "jupyter.pem" \
ubuntu@ec2-18-191-156-172.us-east-2.compute.amazonaws.com
```

Конечно, вам следует заменить указанное здесь DNS-имя на то, которое вы скопировали из описания экземпляра.

Прежде чем разрешить вам войти в компьютер, SSH-клиент запросит подтверждение того, что вы доверяете удаленному экземпляру:

```
The authenticity of host 'ec2-18-191-156-172.us-east-2.compute.amazonaws.com
(18.191.156.172)' can't be established.
ECDSA key fingerprint is SHA256:S5doTJ0GwXVF3i1IFjB10RuHufaVSe+EDqKbGpIN0WI.
Are you sure you want to continue connecting (yes/no)?
```

Ведите yes, чтобы подтвердить.

Теперь вы должны иметь возможность войти в экземпляр и увидеть приветственное сообщение (рис. A.35).

```
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-1032-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Wed Jun  5 06:01:51 UTC 2019

System load:      0.02          Processes:        86
Usage of /:       13.6% of 7.69GB  Users logged in:  0
Memory usage:     14%           IP address for eth0: 172.31.46.216
Swap usage:       0%

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-46-216:~$
```

**Рис. A.35.** После успешного входа в экземпляр EC2 вы должны увидеть приветственное сообщение

С этого момента с машиной можно делать все что угодно.

### Подключение к экземпляру в Windows

Использовать подсистему Linux в Windows — самый простой способ подключиться к экземпляру EC2: вы можете применить SSH и следовать тем же инструкциям, что и в случае Linux.

В качестве альтернативы вы можете использовать Putty (<https://www.putty.org>), чтобы подключиться к экземплярам EC2 из Windows.

### Подключение к экземпляру в macOS

SSH встроен в macOS, поэтому шаги для Linux должны работать и на Mac.

## A.6.5. Завершение работы экземпляра

Окончив работу с экземпляром, вы должны отключить его.

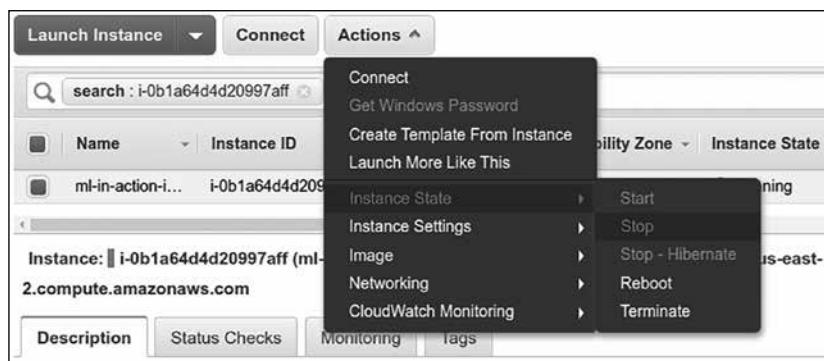
### ВАЖНО

Очень важно отключить экземпляр после завершения работы. Вам выставляется счет за каждую секунду его использования, даже если машина вам больше не нужна и простоявает. Это не распространяется на первые 12 месяцев использования AWS, если запрошенный экземпляр соответствует требованиям бесплатного уровня, но тем не менее полезно выработать привычку периодически проверять статус своей учетной записи и отключать ненужные сервисы.

Это можно сделать из терминала:

```
sudo shutdown now
```

Отключение также возможно из веб-интерфейса: выберите экземпляр, который хотите отключить, перейдите в раздел Actions и выберите Instance State ▶ Stop (рис. A.36).



**Рис. А.36.** Остановка экземпляра из консоли AWS

Как только экземпляр остановлен, вы можете запустить его снова, выбрав Start в том же подменю. Кроме того, можно полностью удалить экземпляр: для этого вам нужно использовать опцию Terminate.

## A.6.6. Настройка интерфейса AWS CLI

AWS CLI — интерфейс командной строки для AWS. Большинство нужных нам действий можно сделать с помощью консоли AWS, но в ряде случаев нам потребуется инструмент командной строки. Например, в главе 5 мы развертываем модель в Elastic Beanstalk, и нам необходимо настраивать интерфейс командной строки.

Чтобы использовать CLI, нужен Python. Если вы используете Linux или macOS, то у вас уже должен быть встроенный дистрибутив Python. В качестве альтернативы вы можете установить Anaconda, следуя приведенным выше инструкциям.

Просто иметь Python недостаточно; вам также необходимо установить сам AWS CLI. Вы можете сделать это, выполнив следующую команду в терминале:

```
pip install awscli
```

Если он у вас уже есть, то полезно будет его обновить:

```
pip install -U awscli
```

После завершения установки необходимо настроить инструмент, указав маркер доступа и секрет, которые вы загрузили ранее при создании пользователя.

Один из способов сделать это — использовать команду `configure`:

```
aws configure
```

Она запросит у вас ключи, которые мы скачали при создании пользователя:

```
$ aws configure
AWS Access Key ID [None]: <ENTER_ACCESS_KEY>
AWS Secret Access Key [None]: <ENTER_SECRET_KEY>
Default region name [None]: us-east-2
Default output format [None]:
```

Здесь используется название региона us-east-2, который расположен в штате Огайо.

Когда вы закончите настройку инструмента, убедитесь, что он работает. Вы можете попросить CLI вернуть ваши идентификационные данные, которые должны соответствовать данным вашего пользователя:

```
$ aws sts get-caller-identity
{
    "UserId": "AIDAVU04TT0055WN6WHZ4",
    "Account": "XXXXXXXXXXXX",
    "Arn": "arn:aws:iam::XXXXXXXXXXXX:user/ml-bookcamp"
}
```

# *Введение в Python*

В настоящее время Python — самый популярный язык для создания проектов машинного обучения, и именно поэтому мы используем его для проектов этой книги.

На случай, если вы не знакомы с Python, в данном приложении рассматриваются его основы: синтаксис и языковые возможности, которые мы используем в книге. Это не самое подробное руководство, но оно призвано дать вам достаточно информации, чтобы вы могли начать использовать Python сразу после завершения чтения. Обратите внимание, что оно довольно краткое и предназначено для людей, которые уже умеют программировать на каком-либо другом языке программирования.

Чтобы выжать максимум из данного приложения, создайте блокнот jupyter, дайте ему имя типа appendix-b-python и используйте для выполнения кода из приложения. Начнем.

## **Б.1. ПЕРЕМЕННЫЕ**

Python — динамический язык, поэтому вам не требуется объявлять типы, как в Java или C++. Например, чтобы создать переменную с целым числом или строкой, нужно всего лишь выполнить простое присвоение:

```
a = 10      ← a — это целое число
b = 'string_b' | b и c — это строки
c = "string_c"
d = 0.999   ← d — это число с плавающей запятой
```

Чтобы напечатать что-либо в стандартный вывод, мы можем использовать функцию `print`:

```
print(a, b, c, d)
```

Она выводит следующее:

```
10 string_b string_c 0.999
```

Чтобы выполнить код, вы можете поместить каждый фрагмент кода в отдельную ячейку jupyter notebook и затем выполнить. Для выполнения кода в ячейке вы можете нажать кнопку Run или же использовать сочетание клавиш Shift+Enter (рис. Б.1).

```
In [1]: a = 10
        b = 'string_b'
        c = "string_c"
        d = 0.999
```

```
In [2]: print(a, b, c, d)
```

```
10 string_b string_c 0.999
```

**Рис. Б.1.** Код, выполняемый в ячейках блокнота Jupyter. Результат виден сразу после выполнения кода

При передаче в `print` нескольких аргументов, как в предыдущем примере, между ними при печати добавляется пробел.

Мы можем объединить несколько переменных с помощью специальной конструкции, называемой *кортежем*:

```
t = (a, b)
```

При выводе `t` на печать мы получим следующее:

```
(10, 'string_b')
```

Чтобы разложить кортеж на несколько переменных, мы используем *присваивание кортежей*:

```
(c, d) = t
```

Теперь `c` и `d` содержат первое и второе значения кортежа:

```
print(c, d)
```

Команда выводит следующее:

```
10 string_b
```

## 420 Приложение Б. Введение в Python

Мы можем опустить круглые скобки при использовании присваивания кортежа:

```
c, d = t
```

Результат будет тем же.

Благодаря присваиванию кортежей можно сократить код. Например, мы можем использовать его для обмена содержимым между двумя переменными:

```
a = 10
b = 20
a, b = b, a           Заменяем a на b и b на a
print("a =", a)
print("b =", b)
```

На выходе видим:

```
a = 20
b = 10
```

При печати мы можем получить красиво отформатированные строки, используя оператор %:

```
print("a = %s" % a)           Заменяем %s на содержимое a
print("b = %s" % b)           Заменяем %s на содержимое b
```

Это приведет к тому же результату:

```
a = 20
b = 10
```

Здесь %s является заполнителем: в данном случае он означает, что мы хотим отформатировать переданный аргумент в виде строки. Другими часто используемыми вариантами являются:

- %d, чтобы отформатировать его как число;
- %f, чтобы отформатировать его как число с плавающей запятой.

Мы можем передать несколько аргументов оператору форматирования в кортеже:

```
print("a = %s, b = %s" % (a, b))
```

Первое вхождение заполнителя %s будет заменено на a, а второе — на b, так что результат будет следующим:

```
a = 20, b = 10
```

Наконец, если у нас есть число с плавающей запятой, то можем использовать для него специальное форматирование:

```
n = 0.0099999999
print("n = %.2f" % n)
```

Это округлит значение с плавающей запятой до второй десятичной запятой при форматировании строки, поэтому при выполнении кода мы получим значение 0,01.

Существует множество вариантов форматирования строк, а также других способов форматирования. Например, существует и так называемый новый способ форматирования с помощью метода `string.format`, который мы не будем рассматривать в этом приложении. Больше информации о параметрах форматирования вы можете узнать по адресу <https://pyformat.info> или в официальной документации.

## Б.1.1. Поток управления

В Python есть три инструкции потока управления: `if`, `for` и `while`. Рассмотрим каждую из них.

### Условия

Простейшим способом управления потоком выполнения программы служит оператор `if`. В Python синтаксис для `if` следующий:

```
a = 10

if a >= 5:
    print('the statement is true')
else:
    print('the statement is false')
```

Код выведет первое утверждение:

```
the statement is true
```

Обратите внимание, что в Python мы используем отступ для группировки кода после оператора `if`. Мы можем связать несколько операторов `if`, используя `elif`, что является сокращением для `else-if`:

```
a = 3

if a >= 5:
    print('the first statement is true')
elif a >= 0:
    print('the second statement is true')
else:
    print('both statements are false')
```

Этот код выведет второе утверждение:

```
the second statement is true
```

## Цикл `for`

Когда требуется повторить один и тот же фрагмент кода несколько раз, мы используем циклы. Традиционный цикл `for` в Python выглядит следующим образом:

```
for i in range(10):
    print(i)
```

Этот код напечатает цифры от 0 до 9, а 10 в эту последовательность уже не войдет:

```
0
1
2
3
4
5
6
7
8
9
```

При указании диапазона мы можем задать начальный и конечный номера, а также шаг приращения:

```
for i in range(10, 100, 5):
    print(i)
```

Этот код выведет числа от 10 до 100 (исключительно) с шагом 5: 10, 15, 20... 95.

Чтобы выйти из цикла раньше времени, мы можем использовать оператор `break`:

```
for i in range(10):
    print(i)
    if i > 5:
        break
```

Этот код выведет числа от 0 до 6. Когда `i` будет равно 6, цикл прервется, поэтому никаких чисел после 6 мы уже не увидим:

```
0
1
2
3
4
5
6
```

Чтобы пропустить итерацию цикла, мы используем оператор `continue`:

```
for i in range(10):
    if i <= 5:
        continue
    print(i)
```

Этот код пропустит итерации при `i` равном 5 или меньше, поэтому мы увидим только числа, начиная с 6:

```
6  
7  
8  
9
```

### Цикл while

Цикл `while` в Python также доступен. Он выполняется, пока определенное условие равно `True`. Например:

```
cnt = 0  
  
while cnt <= 5:  
    print(cnt)  
    cnt = cnt + 1
```

В этом коде мы повторяем цикл, пока условие `cnt <= 5` равно `True`. Как только данное условие перестает быть `True`, выполнение прекращается. Этот код выведет числа от 0 до 5, включая 5:

```
0  
1  
2  
3  
4  
5
```

Мы также можем использовать операторы `break` и `continue` в циклах `while`.

## Б.1.2. Коллекции

Коллекции — это специальные контейнеры, которые позволяют хранить несколько элементов. Мы рассмотрим четыре типа коллекций: списки, кортежи, наборы и словари.

### Списки

*Список* — упорядоченная коллекция с возможностью доступа к элементу по индексу. Чтобы создать список, мы можем просто поместить элементы в квадратные скобки:

```
numbers = [1, 2, 3, 5, 7, 11, 13]
```

Чтобы получить элемент по его индексу, мы можем использовать обозначение в скобках:

```
el = numbers[1]  
print(el)
```

## 424 Приложение Б. Введение в Python

Индексация в Python начинается с 0, поэтому, запрашивая элемент с индексом 1, мы получаем 2-й элемент.

Мы также можем изменить значения в списке:

```
numbers[1] = -2
```

Чтобы получить доступ к элементам с конца, мы можем использовать отрицательные индексы. Например, -1 даст последний элемент, -2 — предпоследний и т. д.:

```
print(numbers[-1], numbers[-2])
```

Как мы и ожидали, команда выводит 13 11.

Чтобы добавить элементы в список, используйте функцию `append`. Она добавит элемент в конец списка:

```
numbers.append(17)
```

Для перебора элементов списка мы используем цикл `for`:

```
for n in numbers:  
    print(n)
```

Выполнив код, увидим печать всех элементов:

```
1  
-2  
3  
5  
7  
11  
13  
17
```

В других языках это также известно как цикл `for-each`: мы выполняем тело цикла для каждого элемента коллекции. Он не включает индексы, только сами элементы. Если нам также нужен доступ к индексу каждого элемента, то мы можем использовать `range`, как делали ранее:

```
for i in range(len(numbers)):  
    n = numbers[i]  
    print("numbers[%d] = %d" % (i, n))
```

Функция `len` возвращает длину списка, поэтому этот код примерно эквивалент традиционному способу обхода массива в C или Java и доступа к каждому элементу по его индексу. После выполнения мы видим следующее:

```
numbers[0] = 1  
numbers[1] = -2  
numbers[2] = 3  
numbers[3] = 5
```

```
numbers[4] = 7
numbers[5] = 11
numbers[6] = 13
numbers[7] = 17
```

Более «питоничным» (более распространенным и идиоматичным в мире Python) способом достижения того же результата будет использование функции `enumerate`:

```
for i, n in enumerate(numbers):
    print("numbers[%d] = %d" % (i, n))
```

В этом коде переменная `i` получит индекс, а переменная `n` — соответствующий элемент из списка. Данный код выдаст точно такой же результат, как и в предыдущем цикле. Чтобы объединить несколько списков в один, мы можем использовать оператор `+`. Например, рассмотрим два списка:

```
list1 = [1, 2, 3, 5]
list2 = [7, 11, 13, 17]
```

Мы можем создать третий список, содержащий все элементы из `list1`, за которыми следуют элементы из `list2`, объединив два списка:

```
new_list = list1 + list2
```

Это приведет к созданию следующего списка:

```
[1, 2, 3, 5, 7, 11, 13, 17]
```

Наконец, также возможно создать список списков: список, элементы которого также являются списками. Чтобы продемонстрировать это, сначала создадим три списка с числами:

```
list1 = [1, 2, 3, 5]
list2 = [7, 11, 13, 17]
list3 = [19, 23, 27, 29]
```

Теперь объединим их в новый список:

```
lists = [list1, list2, list3]
```

Теперь `lists` — это список списков. Перебирая его с помощью цикла `for`, на каждой итерации мы получаем список:

```
for l in lists:
    print(l)
```

Это приведет к следующему результату:

```
[1, 2, 3, 5]
[7, 11, 13, 17]
[19, 23, 27, 29]
```

## Срезы

Еще одной полезной концепцией в Python является *срез*, используемый для получения части списка. Например, еще раз рассмотрим список чисел:

```
numbers = [1, 2, 3, 5, 7]
```

Если мы хотим выбрать подсписок с первыми тремя элементами, то можем использовать оператор двоеточия (:), чтобы указать диапазон для выбора:

```
top3 = numbers[0:3]
```

В этом случае **0:3** означает «выбрать элементы, начиная с индекса 0 до индекса 3 (исключительно)». Результат содержит первые три элемента: [1, 2, 3]. Обратите внимание, что он выбирает элементы с индексами 0, 1 и 2, поэтому 3 не включается.

Если мы хотим включить начало списка, то нам не нужно указывать первое число:

```
top3 = numbers[:3]
```

Если мы не укажем второе число в диапазоне, то получим все до конца списка:

```
last3 = numbers[2:]
```

Список **last3** будет содержать последние три элемента: [3, 5, 7] (рис. Б.2).

	Индекс	0	1	2	3	4	
(A)	numbers[1:3]		1	2	3	5	7
(B)	numbers[:3]		1	2	3	5	7
(C)	numbers[2:]		1	2	3	5	7

**Рис. Б.2.** Использование оператора двоеточия для выбора подсписка

## Кортежи

Ранее мы уже встречались с кортежами в разделе Б.1, посвященном переменным. Кортежи тоже являются коллекциями и очень похожи на списки. Единственное различие заключается в том, что они неизменяемы: как только вы создаете кортеж, вы уже не можете изменить его содержимое.

Для создания кортежа мы используем круглые скобки:

```
numbers = (1, 2, 3, 5, 7, 11, 13)
```

Как и в случае со списками, мы можем получить значение по индексу:

```
e1 = numbers[1]
print(e1)
```

Однако мы не можем обновлять значения в кортеже. Попытавшись это сделать, получим сообщение об ошибке:

```
numbers[1] = -2
```

Если мы попытаемся выполнить этот код, то получим

---

```
TypeError                                     Traceback (most recent call last)
<ipython-input-15-9166360b9018> in <module>
----> 1 numbers[1] = -2

TypeError: 'tuple' object does not support item assignment
```

Аналогично мы не можем добавить к кортежу новый элемент. Однако мы можем использовать конкатенацию для достижения того же результата:

```
numbers = numbers + (17,)
```

Здесь мы создаем новый кортеж, содержащий старые числа, и объединяем его с другим кортежем, содержащим только одно число: 17. Обратите внимание, что нам нужно добавить запятую, чтобы создать кортеж; в противном случае Python будет рассматривать его как простое число.

По сути, выражение на предыдущей странице совпадает с написанием:

```
numbers = (1, 2, 3, 5, 7, 11, 13) + (17,)
```

Сделав это, мы получаем новый кортеж, содержащий новый элемент, поэтому при его печати мы увидим следующее:

```
(1, 2, 3, 5, 7, 11, 13, 17)
```

## **Набор**

Еще одна полезная коллекция — это *набор*: неупорядоченная коллекция, в которой хранятся только уникальные элементы. В отличие от списков, она не может содержать дубликаты; кроме того, невозможно получить доступ к отдельному элементу набора по индексу.

Чтобы создать набор, мы используем фигурные скобки:

- `numbers = {1, 2, 3, 5, 7, 11, 13}`

### **ПРИМЕЧАНИЕ**

Чтобы создать пустой набор, нам нужно использовать `set`:

```
empty_set = set()
```

Использование лишь пустых фигурных скобок создаст словарь — коллекцию, которую мы рассмотрим позже в этом приложении:

```
empty_dict = {}
```

## 428 Приложение Б. Введение в Python

Наборы быстрее списков проверяют, содержит ли коллекция элемент. Для этого в операторе проверки мы используем `in`:

```
print(1 in numbers)
```

Поскольку в наборе `numbers` есть 1, данная строка кода выведет `True`.

Чтобы добавить элемент в набор, мы используем метод `add`:

```
numbers.add(17)
```

Чтобы выполнить проход по всем элементам набора, мы снова используем цикл `for`:

```
for n in numbers:  
    print(n)
```

После выполнения мы увидим

```
1  
2  
3  
5  
7  
11  
13  
17
```

### Словари

Словарь — еще одна крайне полезная коллекция в Python: мы используем ее для построения карты «ключ-значение». Для создания словаря используются фигурные скобки, а для разделения ключей и значений — двоеточия (:):

```
words_to_numbers = {  
    'one': 1,  
    'two': 2,  
    'three': 3,  
}
```

Чтобы получить значение по ключу, мы используем квадратные скобки:

```
print(words_to_numbers['one'])
```

Если чего-то нет в словаре, то Python вызывает исключение:

```
print(words_to_numbers['five'])
```

Пытаясь выполнить команду, мы получим следующую ошибку:

```
-----  
KeyError                                                 Traceback (most recent call last)  
<ipython-input-38-66a309b8feb5> in <module>  
----> 1 print(words_to_numbers['five'])  
  
KeyError: 'five'
```

Чтобы избежать этого, мы можем сначала проверить, содержится ли ключ в словаре. Для этого можно использовать оператор `in`:

```
if 'five' in words_to_numbers:
    print(words_to_numbers['five'])
else:
    print('not in the dictionary')
```

При запуске этого кода мы увидим сообщение `not in the dictionary`.

Другой вариант — использовать метод `get`. Он не вызывает исключений, но возвращает `None`, если ключ отсутствует в словаре:

```
value = words_to_numbers.get('five')
print(value)
```

Он напечатает `None`. При использовании `get` мы можем указать значение по умолчанию в случае, если ключ отсутствует:

```
value = words_to_numbers.get('five', -1)
print(value)
```

В этой ситуации мы получим `-1`.

Чтобы выполнить итерацию по всем ключам словаря, мы используем цикл `for` по результатам метода `keys`:

```
for k in words_to_numbers.keys():
    v = words_to_numbers[k]
    print("%s: %d" % (k, v))
```

На выходе видим

```
one: 1
two: 2
three: 3
```

В качестве альтернативы мы можем напрямую перебирать пары «ключ-значение» в словаре, используя метод `items`:

```
for k, v in words_to_numbers.items():
    print("%s: %d" % (k, v))
```

Он выдает точно такой же результат, как и предыдущий код.

## Генератор списков

Генерация списков — специальный синтаксис для создания и фильтрации списков в Python. Вернемся к списку с числами:

```
numbers = [1, 2, 3, 5, 7]
```

## 430 Приложение Б. Введение в Python

Предположим, мы хотим создать еще один список, в котором все элементы исходного списка возведены в квадрат. Мы можем использовать цикл `for`:

```
squared = []
for n in numbers:
    s = n * n
    squared.append(s)
```

Используя генерацию списков, можно компактно переписать этот код, превратив его в одну строку:

```
squared = [n * n for n in numbers]
```

Кроме того, можно добавить внутри условие `if`, что позволит обрабатывать только те элементы, которые соответствуют условию:

```
squared = [n * n for n in numbers if n > 3]
```

Это можно также представить в виде следующего кода:

```
squared = []
for n in numbers:
    if n > 3:
        s = n * n
        squared.append(s)
```

Если требуется лишь применить фильтр и оставить элементы как есть, то мы можем сделать и это:

```
filtered = [n for n in numbers if n > 3]
```

Эта строка разворачивается так:

```
filtered = []
for n in numbers:
    if n > 3:
        filtered.append(n)
```

Кроме того, можно использовать генератор списка для создания других коллекций с немного другим синтаксисом. Например, для словарей мы заключаем выражение в фигурные скобки и используем двоеточие, чтобы разделить ключи и значения:

```
result = {k: v * 10 for (k, v) in words_to_numbers.items() if v % 2 == 0}
```

В расширенном варианте это будет выглядеть так:

```
result = {}
for (k, v) in words_to_numbers.items():
    if v % 2 == 0:
        result[k] = v * 10
```

**ВНИМАНИЕ**

При изучении генерации списков может возникнуть соблазн начать использовать их повсеместно. Обычно данный синтаксис подходит для простых случаев, но для более сложных ситуаций следует предпочесть циклы `for`, так как они улучшают читабельность кода. В случае сомнений используйте циклы `for`.

### **Б.1.3. Повторное использование кода**

В какой-то момент при написании большого объема кода нам приходится думать о том, как его лучше организовать. Для этого можно разместить небольшие повторно используемые фрагменты кода внутри функций или классов. Узнаем, как это сделать.

#### **Функции**

Чтобы создать функцию, мы используем ключевое слово `def`:

```
def function_name(arg1, arg2):
    # body of the function
    return 0
```

Когда мы хотим выйти из функции и вернуть какое-либо значение, используем оператор `return`. Если просто указать `return` без какого-либо значения или не включать `return` в тело функции вообще, то она вернет `None`.

Например, мы можем написать функцию, которая выводит значения от 0 до указанного числа:

```
def print_numbers(max):
    for i in range(max + 1):
        print(i)
```

Создайте функцию с одним аргументом: `max`  
Используйте аргумент `max` внутри функции

Чтобы вызвать эту функцию, просто добавьте аргументы в круглых скобках после имени:

```
print_numbers(10)
```

Имена аргументов также можно указать при вызове функции:

```
print_numbers(max=10)
```

#### **Классы**

Классы обеспечивают абстракцию более высокого уровня, чем функции: они могут иметь внутреннее состояние и методы, которые работают с этим состоянием. Рассмотрим класс `NumberPrinter`, который выполняет то же самое, что и функция из предыдущего раздела, — печатает числа.

```

class NumberPrinter:

    def __init__(self, max): ← Инициализатор класса
        self.max = max ← Назначьте аргумент
                        max полю max

    def print_numbers(self): ← Метод класса
        for i in range(self.max + 1):←
            print(i)                Используйте внутреннее
                                состояние при вызове метода

```

В этом коде `init` служит инициализатором. Он запускается всякий раз, когда мы хотим создать экземпляр класса:

```
num_printer = NumberPrinter(max=10)
```

Обратите внимание, что внутри класса метод `init` имеет два аргумента: `self` и `max`. Первым аргументом всех методов всегда должен быть `self`: таким образом мы можем с помощью `self` внутри метода получать доступ к состоянию объекта.

Однако, вызывая метод позже, мы ничего не передаем в аргумент `self`: он скрыт от нас. Следовательно, вызывая метод `print_number` экземпляра объекта `NumberPrinter`, мы просто оставляем пустые круглые скобки без параметров:

```
num_printer.print_numbers()
```

Этот код выдает тот же результат, что и функция из предыдущего пункта.

## Импорт кода

Теперь предположим, что мы хотим поместить некий код в отдельный файл. Создадим файл `useful_code.py` и поместим его в ту же папку, где находится и блокнот.

Откройте этот файл с помощью редактора. Внутри файла мы можем поместить функцию и класс, которые создали только что. Таким образом, создаем модуль с именем `useful_code`. Чтобы получить доступ к функции и классу внутри модуля, импортируем их с помощью инструкции `import`:

```
import useful_code
```

Как только он импортирован, его можно использовать:

```
num_printer = useful_code.NumberPrinter(max=10)
num_printer.print_numbers()
```

Можно также импортировать модуль и присвоить ему краткое имя. Например, если вместо написания `useful_code` мы хотим написать `uc`, то можно проделать следующее:

```
import useful_code as uc
```

```
num_printer = uc.NumberPrinter(max=10)
num_printer.print_numbers()
```

Это очень распространенная идиома в научном Python. Пакеты, такие как NumPy и Pandas, обычно импортируются с более короткими псевдонимами:

```
import numpy as np
import pandas as pd
```

Наконец, если мы не хотим импортировать модуль целиком, то можем выбрать, что именно нам нужно, используя синтаксис `from ... import`:

```
from useful_code import NumberPrinter

num_printer = NumberPrinter(max=10)
num_printer.print_numbers()
```

## Б.1.4. Установка библиотек

Наш код можно поместить в пакеты, доступные каждому. Например, подобными пакетами являются NumPy или Pandas. Они уже доступны в дистрибутиве Anaconda, но обычно не поставляются вместе с Python.

Для установки таких внешних пакетов используется встроенный установщик пакетов под названием `pip`. Чтобы воспользоваться им, откройте терминал и выполните команду `pip install`:

```
pip install numpy scipy pandas
```

После команды `install` мы перечисляем пакеты, которые хотим установить. При установке можно указать и версию каждого пакета:

```
pip install numpy==1.16.5 scipy==1.3.1 pandas==0.25.1
```

Когда у нас уже есть установленный пакет, но он устарел и мы хотим его обновить, нужно запустить `pip install` с флагом `-U`:

```
pip install -U numpy
```

Наконец, если мы хотим удалить пакет, то используем `pip uninstall`:

```
pip uninstall numpy
```

## Б.1.5. Программы на Python

Чтобы выполнить код на Python, мы можем просто вызвать интерпретатор Python и указать файл, который необходимо выполнить. Например, чтобы запустить код внутри нашего сценария `useful_code.py`, выполните следующую команду в командной строке:

```
python useful_code.py
```

## 434 Приложение Б. Введение в Python

Когда мы его выполняем, ничего особенного не происходит: в нем мы лишь объявляем функцию и класс, но фактически их не используем. Чтобы увидеть какой-то результат, нам нужно добавить в файл несколько строк кода. Например, можем добавить следующее:

```
num_printer = NumberPrinter(max=10)
num_printer.print_numbers()
```

Теперь, выполнив этот файл, увидим числа, которые печатает `NumberPrinter`.

Однако, когда мы импортируем модуль, внутри Python выполняет все содержимое модуля. Это означает, что в следующий раз, выполнив `import useful_code` в блокноте, мы увидим напечатанные числа.

Чтобы избежать этого, можно сообщить интерпретатору Python, что какой-то код должен выполняться только при выполнении в виде сценария, а не при импорте. Для этого мы помещаем наш код в следующую конструкцию:

```
if __name__ == "__main__":
    num_printer = NumberPrinter(max=10)
    num_printer.print_numbers()
```

Наконец, мы также можем передавать аргументы при запуске сценариев `python`:

```
import sys

# объявления print_numbers и NumberPrinter

if __name__ == "__main__":
    max_number = int(sys.argv[1]) ← Проанализируйте параметр как целое
    num_printer = NumberPrinter(max=max_number) ← число: по умолчанию это строка
    num_printer.print_numbers() ← Передайте проанализированный
                                | аргумент экземпляру NumberPrinter
```

Теперь мы можем запустить сценарий с пользовательскими параметрами:

```
python useful_code.py 5
```

В результате увидим числа от 0 до 5:

```
0
1
2
3
4
5
```

# *Введение в NumPy*

---

Мы не ожидаем, что читатели будут иметь какие-либо знания NumPy, и стараемся размещать всю необходимую информацию в главах по мере продвижения. Однако цель книги — научить машинному обучению, а не NumPy, поэтому мы не смогли подробно описать в этих главах все. Для этого и было создано данное приложение: дать в одном месте обзор наиболее важных концепций NumPy.

В дополнение к введению в NumPy приложение также охватывает немного линейной алгебры, полезной для машинного обучения, включая умножение матриц и векторов, обратную матрицу и нормальное уравнение.

NumPy — библиотека Python, так что если вы еще не знакомы с Python, то прочтайте приложение Б.

## **B.1. NUMPY**

NumPy — сокращение от *Numerical Python* — библиотека Python для числовых манипуляций. NumPy играет центральную роль в экосистеме машинного обучения Python: почти все библиотеки в Python зависят от нее. Например, Pandas, Scikit-learn и TensorFlow полагаются на NumPy в плане числовых операций.

NumPy поставляется предустановленной в дистрибутиве NumPy Anaconda, поэтому, если вы используете ее, вам не придется делать что-то еще. Но если вы не используете Anaconda, то установить NumPy с помощью `pip` довольно просто:

```
pip install numpy
```

Чтобы поэкспериментировать с NumPy, создадим новый блокнот Jupyter и назовем его `appendix-c-numpy`.

## 436 Приложение B. Введение в NumPy

Чтобы использовать библиотеку NumPy, нам нужно ее импортировать. Вот почему в первой ячейке мы пишем:

```
import numpy as np
```

В научном сообществе Python принято использовать псевдоним при импорте NumPy. Вот почему мы добавляем `as np` в установочный код. Это позволяет нам писать в коде `np` вместо `numpy`.

Мы начнем изучение NumPy с его основной структуры данных: массива NumPy.

### B.1.1. Массивы NumPy

Массивы NumPy похожи на списки Python, но они лучше оптимизированы для задач обработки чисел, таких как машинное обучение.

Чтобы создать массив предопределенного размера, заполненный нулями, мы используем функцию `np.zeros`:

```
zeros = np.zeros(10)
```

Это создает массив с десятью нулевыми элементами (рис. B.1).

```
zeros = np.zeros(10)
zeros
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

**Рис. B.1.** Создание числового массива длиной 10, заполненного нулями

Аналогично мы можем создать массив с единицами, используя функцию `np.ones`:

```
ones = np.ones(10)
```

Это сработает точно так же, как с нулями, за исключением того, что элементами будут единицы.

Обе функции являются сокращением для более общей функции: `np.full`. Она создает массив определенного размера, заполненный указанным элементом. Например, чтобы создать массив размером 10, заполненный нулями, мы делаем следующее:

```
array = np.full(10, 0.0)
```

Мы можем добиться того же результата, используя функцию `np.repeat`:

```
array = np.repeat(0.0, 10)
```

Этот код выдает тот же результат, что и предыдущий (рис. B.2).

```
array = np.full(10, 0.0)
array
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

array = np.repeat(0.0, 10)
array
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

**Рис. B.2.** Чтобы создать массив, заполненный определенным числом, используйте `np.full` или `np.repeat`

Хотя в этом примере обе функции выдают один и тот же код, `np.repeat` на самом деле более действенная команда. Например, мы можем использовать ее для создания массива, в котором несколько элементов повторяются один за другим:

```
array = np.repeat([0.0, 1.0], 5)
```

Он создает массив размером 10, в котором число 0 повторяется пять раз, а затем число 1 повторяется еще пять раз (рис. B.3):

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

```
array = np.repeat([0.0, 1.0], 5)
array
array([0., 0., 0., 0., 0., 1., 1., 1., 1., 1.])

array = np.repeat([0.0, 1.0], [2, 3])
array
array([0., 0., 1., 1., 1.])
```

**Рис. B.3.** Функция `np.repeat` более гибкая, чем `np.full`: она может создавать массивы, повторяя несколько элементов

Мы можем добиться еще большей гибкости и указать, сколько раз должен повторяться каждый элемент:

```
array = np.repeat([0.0, 1.0], [2, 3])
```

В этом случае 0,0 повторяется два раза, а 1,0 — три:

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

Как и в случае со списками, мы можем получить доступ к элементу массива с помощью квадратных скобок:

```
e1 = array[1]
print(e1)
```

Этот код выводит значение 0,0.

## 438 Приложение B. Введение в NumPy

В отличие от обычных списков Python, мы можем получить доступ к нескольким элементам массива одновременно, используя список с индексами в квадратных скобках:

```
print(array[[4, 2, 0]])
```

Результатом является другой массив размера 3, состоящий из элементов исходного массива, проиндексированных на 4, 2 и 0 соответственно:

```
[1., 1., 0.]
```

Мы также можем обновить элементы массива, используя квадратные скобки:

```
array[1] = 1  
print(array)
```

Поскольку мы изменили элемент с индексом 1 с 0 на 1, он выводит следующее:

```
[0. 1. 1. 1. 1.]
```

Если у нас уже есть список с числами, можем преобразовать его в массив NumPy, используя `np.array`:

```
elements = [1, 2, 3, 4]  
array = np.array(elements)
```

Теперь `array` — это массив NumPy размером 4 с теми же элементами, что и исходный список:

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

Другой полезной функцией для создания массивов NumPy является `np.arange`. Это числовой эквивалент `range` в Python:

```
np.arange(10)
```

Он создает массив длиной 10 с числами от 0 до 9, и, как и при использовании стандартной `range` из Python, 10 не включается в массив:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Часто нам нужно создать массив определенного размера, заполненный числами между неким числом  $x$  и неким числом  $y$ . Например, представьте, что нужно создать массив с числами от 0 до 1:

```
0,0,0,1,0,2,...,0,9,1,0
```

Мы можем использовать `np.linspace`:

```
thresholds = np.linspace(0, 1, 11)
```

Эта функция принимает три параметра:

- начальное число — в нашем случае мы хотим начать с 0;
- последнее число — мы хотим закончить единицей;

- длина итогового массива — в нашем случае мы хотим, чтобы в массиве было 11 чисел.

Этот код выдает 11 чисел от 0 до 1 (рис. B.4).

```
thresholds = np.linspace(0, 1, 11)
thresholds
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

**Рис. B.4.** Функция `linspace` из NumPy создает последовательность заданной длины (11), которая начинается с 0 и заканчивается числом 1

Списки Python обычно могут содержать элементы любого типа. Это не относится к массивам NumPy: все элементы массива должны иметь один и тот же тип. Эти типы называются *dtypes*.

Существуют четыре широкие категории *dtypes*:

- беззнаковые целые (`uint`) — целые числа, которые всегда положительны (или равны нулю);
- целые числа со знаком (`int`) — целые числа, которые могут быть положительными и отрицательными;
- числа с плавающей запятой (`float`) — действительные числа;
- логические значения (`bool`) — только значения `True` и `False`.

Существует множество вариантов каждого *dtype*, в зависимости от количества битов, используемых для представления значения в памяти.

Для `uint` у нас есть четыре типа: `uint8`, `uint16`, `uint32` и `uint64` размером 8, 16, 32 и 64 бита соответственно. Аналогично у нас есть четыре типа `int`: `int8`, `int16`, `int32` и `int64`. Чем больше битов мы используем, тем большие числа можем хранить (табл. B.1).

**Таблица B.1.** Три распространенных типа данных NumPy: `uint`, `int` и `float`. Каждый *dtype* имеет несколько вариантов размера в диапазоне от 8 до 64 бит

Размер (биты)	<code>uint</code>	<code>int</code>	<code>float</code>
<b>8</b>	$0 \dots 2^8 - 1$	$-2^7 \dots 2^7 - 1$	—
<b>16</b>	$0 \dots 2^{16} - 1$	$-2^{15} \dots 2^{15} - 1$	Половинная точность
<b>32</b>	$0 \dots 2^{32} - 1$	$-2^{31} \dots 2^{31} - 1$	Одинарная точность
<b>64</b>	$0 \dots 2^{64} - 1$	$-2^{63} \dots 2^{63} - 1$	Двойная точность

В случае `float` у нас есть три типа: `float16`, `float32` и `float64`. Чем больше битов мы используем, тем точнее значение `float`.

С полным списком различных *dtypes* вы можете ознакомиться в официальной документации (<https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html>).

**ПРИМЕЧАНИЕ**

В NumPy в качестве float по умолчанию используется float64, который использует 64 бита (8 байт) для каждого числа. Для большинства приложений машинного обучения нам не нужна такая точность, и мы можем уменьшить объем памяти в два раза, используя float32 вместо float64.

При создании массива мы можем указать dtype. Например, при использовании `np.zeros` и `np.ones` dtype по умолчанию равен `float64`. Мы можем указать dtype при создании массива (рис. B.5):

```
zeros = np.zeros(10, dtype=np.uint8)
```

```
zeros = np.zeros(10, dtype=np.uint8)
zeros
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=uint8)
```

**Рис. B.5.** Мы можем указать dtype при создании массива

Когда у нас есть массив с целыми числами и мы присваиваем число за пределами диапазона, число сокращается — сохраняются только младшие значащие биты.

Например, предположим, что мы используем только что созданный массив `zeros` с типом `uint8`. Поскольку его dtype — это `uint8`, наибольшее число, которое он может хранить, равно 255. Попробуем присвоить 300 первому элементу массива:

```
zeros[0] = 300
print(zeros[0])
```

Поскольку 300 больше 255, сохраняются только младшие значащие биты, поэтому код выводит 44.

**ВНИМАНИЕ**

Будьте осторожны при выборе dtype для массива. Если вы случайно выберете слишком узкий dtype, то NumPy не предупредит вас, когда вы введете большое число. Он просто его усечет.

Перебор всех элементов массива аналогичен перебору списка. Мы просто можем использовать цикл `for`:

```
for i in np.arange(5):
    print(i)
```

Этот код выводит числа от 0 до 4:

```
0
1
2
3
4
```

## **В.1.2. Двумерные массивы NumPy**

До сих пор мы рассматривали одномерные массивы NumPy. Мы можем думать об этих массивах как о векторах. Однако для приложений машинного обучения недостаточно иметь только векторы: нам также часто нужны матрицы.

В обычном Python мы бы использовали для этого список списков. В NumPy эквивалентом служит двумерный массив.

Чтобы создать двумерный массив с нулями, мы просто используем кортеж вместо числа при вызове `np.zeros`:

```
zeros = np.zeros((5, 2), dtype=np.float32)
```

Мы используем кортеж (5, 2), поэтому создается массив нулей с пятью строками и двумя столбцами (рис. В.6).

```
zeros = np.zeros((5, 2), dtype=np.float32)
zeros

array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]], dtype=float32)
```

**Рис. В.6.** Чтобы создать двумерный массив, используйте кортеж из двух элементов. Первый задает количество строк, а второй — количество столбцов

Таким же образом мы можем использовать `np.ones` или `np.fill` — вместо одного числа вводим кортеж.

Размерность массива называется *формой*. Это первый параметр, который мы передаем функции `pr.zeros`: он определяет, сколько строк и столбцов будет в массиве. Чтобы получить форму массива, используйте свойство `shape`:

```
print(zeros.shape)
```

После выполнения мы видим  $(5, 2)$ .

Можно преобразовать список списков в массив NumPy. Как и в случае с обычными списками чисел, просто используйте `np.array`:

```
numbers = [ [1, 2, 3],  
            [4, 5, 6],  
            [7, 8, 9] ]  
numbers = np.array(numbers) ← Создает список списков  
                           | Преобразует список  
                           в двумерный массив
```

## 442     Приложение B. Введение в NumPy

После выполнения этого кода `numbers` становится массивом NumPy с формой  $(3, 3)$ . Когда мы распечатываем его, то видим следующее:

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

Чтобы получить доступ к элементу двумерного массива, нам нужно использовать два числа внутри скобок:

```
print(numbers[0, 1])
```

Этот код получит доступ к строке с индексом 0 и столбцу с индексом 1. Таким образом, он напечатает 2.

Как и в случае с одномерными массивами, мы используем оператор присваивания (`=`) для изменения отдельного значения двумерного массива:

```
numbers[0, 1] = 10
```

После выполнения содержимое массива изменяется:

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

Если вместо двух чисел мы поместим только одно, то получим всю строку, которая представляет собой одномерный массив NumPy:

```
numbers[0]
```

Этот код возвращает всю строку с индексом 0:

```
array([1 2 3])
```

Чтобы получить доступ к столбцу двумерного массива, мы используем двоеточие `(:)` вместо первого элемента. Как и в случае со строками, результатом также является одномерный массив NumPy:

```
numbers[:, 1]
```

Выполнив команду, мы увидим весь столбец целиком:

```
array([2 5 8])
```

Кроме того, можно перезаписать содержимое всей строки или столбца с помощью оператора присваивания. Например, предположим, что мы хотим заменить строку в матрице:

```
numbers[1] = [1, 1, 1]
```

Это приводит к следующему изменению:

```
array([[ 1, 10, 3],
       [ 1,  1, 1],
       [ 7,  8, 9]])
```

Аналогично можем заменить содержимое всего столбца:

```
numbers[:, 2] = [9, 9, 9]
```

В результате последний столбец изменится:

```
array([[ 1, 10, 9],
       [ 1,  1, 9],
       [ 7,  8, 9]])
```

### B.1.3. Случайно сгенерированные массивы

Часто бывают полезны массивы, заполненные случайными числами. Чтобы создать такой массив в NumPy, мы используем модуль `np.random`.

Например, чтобы сгенерировать массив случайных чисел размером  $5 \times 2$ , равномерно распределенных между 0 и 1, используйте `np.random.rand`:

```
arr = np.random.rand(5, 2)
```

При запуске будет получен массив, который выглядит следующим образом:

```
array([[0.64814431, 0.51283823],
       [0.40306102, 0.59236807],
       [0.94772704, 0.05777113],
       [0.32034757, 0.15150334],
       [0.10377917, 0.68786012]])
```

Каждый раз, когда мы запускаем код, он генерирует другой результат. Иногда нам нужно, чтобы результаты были воспроизводимыми, а это значит, что при выполнении этого кода позже мы должны получить те же результаты. Чтобы достичь этого, можем установить начальное значение генератора случайных чисел. Как только начальное значение задано, генератор случайных чисел выдает одну и ту же последовательность каждый раз, когда мы запускаем код:

```
np.random.seed(2)
arr = np.random.rand(5, 2)
```

В Ubuntu Linux версии 18.04 с NumPy версии 1.17.2 он генерирует следующий массив:

```
array([[0.4359949 , 0.02592623],
       [0.54966248, 0.43532239],
       [0.4203678 , 0.33033482],
       [0.20464863, 0.61927097],
       [0.29965467, 0.26682728]])
```

Независимо от того, сколько раз мы повторно выполняем эту ячейку, результаты остаются неизменными.

### ВНИМАНИЕ

Исправление начального значения генератора случайных чисел гарантирует, что генератор выдаст те же результаты при выполнении на той же ОС с той же версией NumPy. Однако нет никакой гарантии, что обновление версии NumPy не повлияет на воспроизводимость: изменение версии может вызвать изменения в алгоритме генератора случайных чисел, и это, в свою очередь, может привести к разным результатам в разных версиях.

Если вместо равномерного распределения мы хотим сделать выборку из стандартного нормального распределения, то используем `np.random.randn`:

```
arr = np.random.randn(5, 2)
```

### ПРИМЕЧАНИЕ

Каждый раз, генерируя случайный массив в этом приложении, мы обязательно фиксируем начальное число перед его генерацией, даже если явно не указываем его в коде. Это делается для обеспечения согласованности. Мы используем 2 в качестве начального значения. Для выбора именно этого числа нет никаких особых причин.

Чтобы генерировать равномерно распределенные случайные целые числа от 0 до 100 (исключительно), мы можем использовать `np.random.randint`:

```
randint = np.random.randint(low=0, high=100, size=(5, 2))
```

При выполнении кода мы получаем числовой массив целых чисел размером  $5 \times 2$ :

```
array([[40, 15],
       [72, 22],
       [43, 82],
       [75, 7],
       [34, 49]])
```

Еще одной весьма полезной функцией является перетасовка массива — перестановка элементов массива в случайном порядке. Например, создадим массив с диапазоном, а затем перетасуем его:

```
idx = np.arange(5)
print('before shuffle', idx)

np.random.shuffle(idx)
print('after shuffle', idx)
```

После запуска кода мы видим следующее:

```
before shuffle [0 1 2 3 4]
after shuffle [2 3 0 4 1]
```

## B.2. ОПЕРАЦИИ NUMPY

NumPy поставляется с широким спектром операций, которые работают с массивами NumPy. В этом разделе мы рассмотрим операции, которые понадобятся нам на протяжении всей книги.

### B.2.1. Операции по элементам

Массивы NumPy поддерживают все арифметические операции: сложение (+), вычитание (-), умножение (\*), деление (/) и др.

Чтобы проиллюстрировать эти операции, сначала создадим массив с помощью `arange`:

```
rng = np.arange(5)
```

Данный массив содержит пять элементов от 0 до 4:

```
array([0, 1, 2, 3, 4])
```

Чтобы умножить каждый элемент массива на два, мы просто используем оператор умножения (\*):

```
rng * 2
```

В результате мы получаем новый массив, где каждый элемент из исходного массива умножен на два:

```
array([0, 2, 4, 6, 8])
```

Обратите внимание, что нам не нужно явно использовать какие-либо циклы, чтобы применить операцию умножения индивидуально к каждому элементу: NumPy делает это за нас. Мы можем сказать, что операция умножения применяется *поэлементно* — ко всем элементам сразу. Операции сложения (+), вычитания (-) и деления (/) также являются поэлементными и не требуют явных циклов.

Такие поэлементные операции часто называют *векторизованными*: цикл `for` выполняется внутри собственного кода (написанного на C и Fortran), поэтому операции выполняются очень быстро!

#### ПРИМЕЧАНИЕ

Всякий раз, когда это возможно, используйте векторизованные операции из NumPy вместо циклов: они всегда выполняются на порядок быстрее.

В предыдущем коде мы использовали только одну операцию. Но в одном выражении можно применить сразу несколько:

```
(rng - 1) * 3 / 2 + 1
```

## 446 Приложение B. Введение в NumPy

Этот код создает новый массив с результатом:

```
array([-0.5, 1. , 2.5, 4. , 5.5])
```

Обратите внимание, что исходный массив содержит целые числа, но, поскольку мы использовали операцию деления, результатом будет массив с числами с плавающей запятой.

Ранее наш код включал массив и простые числа Python. Можно выполнять и поэлементные операции с двумя массивами, если они имеют одинаковую форму.

Например, предположим, что у нас два массива, один из которых содержит числа от 0 до 4, а другой — некий случайный шум:

```
noise = 0.01 * np.random.rand(5)
numbers = np.arange(5)
```

Иногда нам приходится делать это в целях моделирования неидеальных реальных данных: на самом деле при сборе данных всегда есть неоднородности, и мы можем смоделировать их, добавив шум.

Мы строим массив `noise`, сначала генерируя числа от 0 до 1, а затем умножая их на 0,01. По сути, команда генерирует случайные числа от 0 до 0,01:

```
array([0.00435995, 0.00025926, 0.00549662, 0.00435322, 0.00420368])
```

Затем мы можем сложить эти два массива и получить третий с суммой:

```
result = numbers + noise
```

В этом массиве каждый элемент результата является суммой соответствующих элементов двух других массивов:

```
array([0.00435995, 1.00025926, 2.00549662, 3.00435322, 4.00420368])
```

Мы можем округлить числа с любой точностью, используя метод `round`:

```
result.round(4)
```

Это также поэлементная операция, поэтому она применяется ко всем элементам сразу, а числа округляются до четвертой цифры:

```
array([0.0044, 1.0003, 2.0055, 3.0044, 4.0042])
```

Иногда нам нужно возвести в квадрат все элементы массива. Для этого мы можем просто умножить массив сам на себя. Сначала сгенерируем массив:

```
pred = np.random.rand(3).round(2)
```

Он содержит три случайных числа:

```
array([0.44, 0.03, 0.55])
```

Теперь мы можем умножить его на самого себя:

```
square = pred * pred
```

В результате мы получим новый массив, где каждый элемент исходного массива возведен в квадрат:

```
array([0.1936, 0.0009, 0.3025])
```

В качестве альтернативы мы можем использовать оператор степени (\*\*):

```
square = pred ** 2
```

Оба подхода приводят к одинаковым результатам (рис. B.7).

<pre>np.random.seed(2) pred = np.random.rand(3).round(2) pred</pre>	<pre>array([0.44, 0.03, 0.55])</pre>
<pre>square = pred * pred square</pre>	<pre>array([0.1936, 0.0009, 0.3025])</pre>
<pre>square = pred ** 2 square</pre>	<pre>array([0.1936, 0.0009, 0.3025])</pre>

**Рис. B.7.** Существуют два способа возведения элементов массива в квадрат: умножить массив на сам себя или использовать операцию степени (\*\*)

Другими полезными поэлементными операциями, которые могут пригодиться для приложений машинного обучения, являются экспонента, логарифм и квадратный корень:

```
pred_exp = np.exp(pred)      ← Вычисляет экспоненту
pred_log = np.log(pred)      ← Вычисляет логарифм
pred_sqrt = np.sqrt(pred)    ← Вычисляет квадратный корень
```

Логические операции также можно применять к массивам NumPy поэлементно. Чтобы проиллюстрировать это, снова сгенерируем массив с некоторыми случайными числами:

```
pred = np.random.rand(3).round(2)
```

Он содержит следующие числа:

```
array([0.44, 0.03, 0.55])
```

Мы можем выявить элементы, которые больше 0,5:

```
result = pred >= 0.5
```

## 448 Приложение B. Введение в NumPy

В результате мы получаем массив с тремя логическими значениями:

```
array([False, False, True])
```

Мы знаем, что только последний элемент исходного массива больше 0,5, поэтому в этой позиции значение будет `True`, а все остальное — `False`.

Как и в случае с арифметическими операциями, мы можем применить логические операции к двум массивам NumPy одинаковой формы. Сгенерируем два случайных массива:

```
pred1 = np.random.rand(3).round(2)
pred2 = np.random.rand(3).round(2)
```

Массивы имеют следующие значения:

```
array([0.44, 0.03, 0.55])
array([0.44, 0.42, 0.33])
```

Теперь мы можем с помощью оператора «больше или равно» (`>=`) сравнить их значения:

```
pred1 >= pred2
```

В итоге мы получаем массив с логическими значениями (рис. B.8):

```
array([ True, False, True])
```

```
pred1 = np.random.rand(3).round(2)
pred1
```

```
array([0.44, 0.03, 0.55])
```

```
pred2 = np.random.rand(3).round(2)
pred2
```

```
array([0.44, 0.42, 0.33])
```

```
pred1 >= pred2
```

```
array([ True, False, True])
```

**Рис. B.8.** Логические операции в NumPy являются поэлементными и могут быть применены к двум массивам одинаковой формы для сравнения значений

Наконец, мы можем применять логические операции — такие как логическое И (`&`) и ИЛИ (`|`) — к логическим массивам NumPy. Снова сгенерируем два случайных массива:

```
pred1 = np.random.rand(5) >= 0.3
pred2 = np.random.rand(5) >= 0.4
```

Сгенерированные массивы имеют следующие значения:

```
array([ True, False, True])
array([ True, True, False])
```

Подобно арифметическим операциям, логические операторы также являются поэлементными. Например, для поэлементного вычисления логического И мы просто используем оператор `&` с массивами (рис. B.9):

```
res_and = pred1 & pred2
```

```
pred1 = np.random.rand(3) >= 0.3
pred1
```

```
array([ True, False,  True])
```

```
pred2 = np.random.rand(3) >= 0.4
pred2
```

```
array([ True,  True, False])
```

```
pred1 & pred2
```

```
array([ True, False, False])
```

```
pred1 | pred2
```

```
array([ True,  True,  True])
```

**Рис. B.9.** Логические операции, такие как логическое И и логическое ИЛИ, также могут применяться поэлементно

В результате мы получаем

```
array([ True, False, False])
```

Логическое ИЛИ работает таким же образом (см. рис. B.9):

```
res_or = pred1 | pred2
```

В итоге мы получим следующий массив:

```
array([ True,  True,  True])
```

## B.2.2. Суммирующие операции

В то время как операции по элементам принимают массив и создают массив той же формы, операции суммирования принимают массив и производят одно число.

Например, мы можем сгенерировать массив, а затем вычислить сумму всех элементов:

```
pred = np.random.rand(3).round(2)
pred_sum = pred.sum()
```

В этом примере `pred` — это

```
array([0.44, 0.03, 0.55])
```

Тогда `pred_sum` — сумма всех трех элементов, которая равна 1,02:

$$0,44 + 0,03 + 0,55 = 1,02$$

Другие операции суммирования включают `min`, `mean`, `max` и `std`:

```
print('min = %.2f' % pred.min())
print('mean = %.2f' % pred.mean())
```

## 450 Приложение B. Введение в NumPy

```
print('max = %.2f' % pred.max())
print('std = %.2f' % pred.std())
```

После запуска этого кода мы видим

```
min = 0.03
mean = 0.34
max = 0.55
std = 0.22
```

Когда у нас есть двумерный массив, операции суммирования также приводят к получению одного числа. Однако можно применить эти операции и к строкам или столбцам отдельно.

Например, сгенерируем массив  $4 \times 3$ :

```
matrix = np.random.rand(4, 3).round(2)
```

В итоге мы получим массив:

```
array([[0.44, 0.03, 0.55],
       [0.44, 0.42, 0.33],
       [0.2 , 0.62, 0.3 ],
       [0.27, 0.62, 0.53]])
```

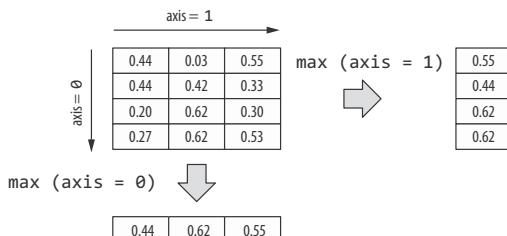
Если вызвать метод `max`, то он возвратит одно число:

```
matrix.max()
```

Результат равен 0,62, что является максимальным числом из всех элементов матрицы.

Если мы теперь захотим найти наибольшее число в каждой строке, то можем использовать метод `max`, указывающий ось, вдоль которой нужно применять эту операцию. Когда мы хотим сделать это для строк, используем `axis=1` (рис. B.10):

```
matrix.max(axis=1)
```



**Рис. B.10.** Мы можем указать ось, вдоль которой применяем операцию: `axis=1` означает применение к строкам, а `axis=0` означает применение к столбцам

В результате получаем массив с четырьмя числами — по наибольшему числу в каждой строке:

```
array([0.55, 0.44, 0.62, 0.62])
```

Аналогично мы можем найти наибольшее число в каждом столбце. Для этого используем `axis=0`:

```
matrix.max(axis=0)
```

На этот раз результатом будут три числа — самые большие числа в каждом столбце:

```
array([0.44, 0.62, 0.55])
```

Другие операции — `sum`, `min`, `mean`, `std` и многие другие — также могут принимать `axis` в качестве аргумента. Например, можно легко вычислить сумму элементов каждой строки:

```
matrix.sum(axis=1)
```

При его выполнении мы получаем четыре числа:

```
array([1.02, 1.19, 1.12, 1.42])
```

### B.2.3. Сортировка

Часто нам приходится сортировать элементы массива. Посмотрим, как это сделать в NumPy. Сначала генерируем одномерный массив с четырьмя элементами:

```
pred = np.random.rand(4).round(2)
```

Готовый массив содержит следующие элементы:

```
array([0.44, 0.03, 0.55, 0.44])
```

Чтобы создать отсортированную копию массива, используйте `np.sort`:

```
np.sort(pred)
```

Команда возвращает массив со всеми отсортированными элементами:

```
array([0.03, 0.44, 0.44, 0.55])
```

Поскольку он создает копию и сортирует именно ее, исходный массив `pred` остается неизменным.

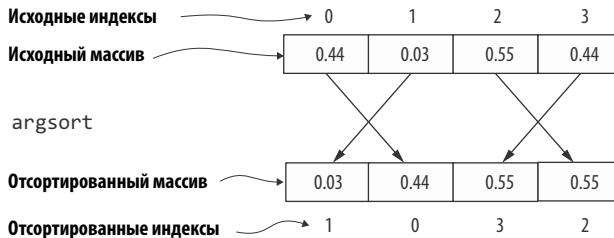
Если мы хотим отсортировать элементы массива без создания нового, то вызываем метод `sort` для самого массива:

```
pred.sort()
```

Теперь массив `pred` отсортирован.

Что касается сортировки, то в нашем распоряжении есть еще одна полезная команда: `argsort`. Вместо сортировки массива она возвращает индексы массива в отсортированном порядке (рис. B.11):

```
idx = pred.argsort()
```



**Рис. B.11.** Функция `sort` сортирует массив, в то время как `argsort` создает массив индексов отсортированного массива

Теперь массив `idx` содержит индексы в порядке сортировки:

```
array([1, 0, 3, 2])
```

Теперь мы можем использовать массив `idx`, чтобы получить исходный массив в отсортированном порядке:

```
pred[idx]
```

Как мы видим, порядок действительно тот, который нам нужен:

```
array([0.03, 0.44, 0.44, 0.55])
```

## B.2.4. Изменение формы и объединение

Каждый массив NumPy имеет форму, которая определяет его размер. Для одномерного массива это его длина, а для двумерного — количество строк и столбцов. Мы уже знаем, что можем получить доступ к форме массива, используя свойство `shape`:

```
rng = np.arange(12)
rng.shape
```

Форма `rng` равна (12); это значит, что это одномерный массив длиной 12. Поскольку мы использовали `np.arange` для создания массива, он содержит числа от 0 до 11 (включительно):

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Можно изменить форму массива с одномерной на двумерную. Для этого используем метод `reshape`:

```
rng.reshape(4, 3)
```

В результате мы получаем матрицу с четырьмя строками и тремя столбцами:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Изменение формы сработало, поскольку вполне возможно превратить 12 исходных элементов в четыре строки с тремя столбцами. Другими словами, общее количество элементов не изменилось. Однако если мы попытаемся изменить форму на  $(4, 4)$ , то ничего не получится:

```
rng.reshape(4, 4)
```

Когда мы это сделаем, NumPy вызовет `ValueError`:

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-176-880fb98fa9c8> in <module>
----> 1 rng.reshape(4, 4)
```

```
ValueError: cannot reshape array of size 12 into shape (4,4)
```

Иногда нам нужно создать новый массив NumPy, объединив несколько массивов. Посмотрим, как это сделать.

Сначала создадим два массива, которые используем для иллюстрации:

```
vec = np.arange(3)
mat = np.arange(6).reshape(3, 2)
```

Первый, `vec`, представляет собой одномерный массив с тремя элементами:

```
array([0, 1, 2])
```

Второй, `mat`, двумерный, с тремя строками и двумя столбцами:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Самый простой способ объединить два массива NumPy — использовать функцию `np.concatenate`:

```
np.concatenate([vec, vec])
```

Она принимает список одномерных массивов и объединяет их в один большой. В нашем случае мы передаем `vec` два раза, поэтому в результате имеем массив длиной в шесть элементов:

```
array([0, 1, 2, 0, 1, 2])
```

## 454 Приложение B. Введение в NumPy

Мы можем достичь того же результата, используя `np.hstack`, что является сокращением от horizontal stack (горизонтальный стек):

```
np.hstack([vec, vec])
```

Эта функция тоже берет список массивов и складывает их горизонтально, создавая больший массив:

```
array([0, 1, 2, 0, 1, 2])
```

Мы также можем применить `np.hstack` к двумерным массивам:

```
np.hstack([mat, mat])
```

Результатом будет новая матрица, в которой исходные матрицы расположены горизонтально по столбцам:

```
array([[0, 1, 0, 1],
       [2, 3, 2, 3],
       [4, 5, 4, 5]])
```

Однако в случае двумерных массивов `np.concatenate` работает иначе, чем `np.hstack`:

```
np.concatenate([mat, mat])
```

Когда мы применяем `np.concatenate` к матрицам, команда складывает их вертикально, а не горизонтально, как одномерные массивы, создавая новую матрицу с шестью строками:

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [0, 1],
       [2, 3],
       [4, 5]])
```

Другим полезным методом объединения массивов NumPy служит `np.column_stack`: он позволяет складывать векторы и матрицы вместе. Например, предположим, что мы хотим добавить дополнительный столбец в нашу матрицу. Для этого мы просто передаем список, содержащий вектор, а затем матрицу:

```
np.column_stack([vec, mat])
```

В результате мы имеем новую матрицу, где `vec` становится первым столбцом, а следом идет остальная часть `mat`:

```
array([[0, 0, 1],
       [1, 2, 3],
       [2, 4, 5]])
```

Мы можем применить `np.column_stack` к двум векторам:

```
np.column_stack([vec, vec])
```

В результате получается матрица из двух столбцов:

```
array([[0, 0],
       [1, 1],
       [2, 2]])
```

Помимо команды `np.hstack`, которая складывает массивы горизонтально, существует и `np.vstack`, которая складывает их вертикально:

```
np.vstack([vec, vec])
```

Когда мы вертикально складываем два вектора, получаем матрицу с двумя строками:

```
array([[0, 1, 2],
       [0, 1, 2]])
```

Мы также можем сложить вертикально две матрицы:

```
np.vstack([mat, mat])
```

Результат тот же, что и у `np.concatenate([mat, mat])`, — новая матрица с шестью строками:

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [0, 1],
       [2, 3],
       [4, 5]])
```

Функция `np.vstack` также может складывать вместе векторы и матрицы, фактически создавая матрицу с новыми строками:

```
np.vstack([vec, mat.T])
```

Когда мы это делаем, `vec` становится первой строкой в новой матрице:

```
array([[0, 1, 2],
       [0, 2, 4],
       [1, 3, 5]])
```

Обратите внимание, что в этом коде мы использовали свойство `T` матрицы `mat`. Это операция транспонирования матрицы, которая преобразует строки матрицы в столбцы:

```
mat.T
```

Первоначально `mat` имеет следующие данные:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

## 456 Приложение B. Введение в NumPy

После транспонирования то, что было столбцом, становится строкой:

```
array([[0, 2, 4],  
       [1, 3, 5]])
```

### B.2.5. Срезы и фильтрация

Как и в случае со списками Python, мы также можем использовать срез для доступа к части массива NumPy. Предположим, что у нас есть матрица  $5 \times 3$ :

```
mat = np.arange(15).reshape(5, 3)
```

Она состоит из пяти строк и трех столбцов:

```
array([[ 0,  1,  2],  
       [ 3,  4,  5],  
       [ 6,  7,  8],  
       [ 9, 10, 11],  
       [12, 13, 14]])
```

Мы можем получить доступ к частям этой матрицы с помощью среза. Например, можно получить первые свободные строки, используя оператор диапазона (`:`):

```
mat[:3]
```

Он возвращает строки с индексами 0, 1 и 2 (3 не включен):

```
array([[0, 1, 2],  
       [3, 4, 5],  
       [6, 7, 8]])
```

Если нам нужны только строки 1 и 2, то мы указываем как начало, так и конец диапазона:

```
mat[1:3]
```

Это даст нам нужные строки:

```
array([[3, 4, 5],  
       [6, 7, 8]])
```

Как и в случае со строками, мы можем выбрать только некоторые столбцы; например, первые два:

```
mat[:, :2]
```

Здесь у нас два диапазона:

- первый — это просто двоеточие `(:)` без начала и конца, что означает «включить все строки»;
- второй — это диапазон, который включает столбцы 0 и 1 (2 не включен).

Следовательно, в результате получаем такую матрицу:

```
array([[ 0,  1],
       [ 3,  4],
       [ 6,  7],
       [ 9, 10],
       [12, 13]])
```

Конечно, мы можем объединить и то и другое и выбрать любую нужную нам часть матрицы:

```
mat[1:3, :2]
```

Код даст нам строки 1 и 2 и столбцы 0 и 1:

```
array([[3, 4],
       [6, 7]])
```

Если нам нужен не диапазон, а какие-то конкретные строки или столбцы, то мы можем просто предоставить список индексов:

```
mat[[3, 0, 1]]
```

Результатом будут три строки с индексами 3, 0 и 1:

```
array([[ 9, 10, 11],
       [ 0,  1,  2],
       [ 3,  4,  5]])
```

Вместо отдельных индексов можно использовать двоичную маску, чтобы указать, какие строки следует выбирать. Например, предположим, что мы хотим выбрать строки, где первым элементом строки является нечетное число.

Чтобы проверить, является ли первый элемент нечетным, нам нужно выполнить следующие действия:

1. Выбрать первый столбец матрицы.
2. Применить операцию `mod 2 (%)` ко всем элементам, чтобы вычислить остаток от деления на 2.
3. Если остаток равен 1, то число нечетное, а если 0, то четное.

Это переводится в следующее выражение NumPy:

```
mat[:, 0] % 2 == 1
```

В итоге получим массив с логическими значениями:

```
array([False, True, False, True, False])
```

Мы видим, что выражение `True` задано для строк 1 и 3 и `False` для строк 0, 2 и 5. Теперь можем использовать это выражение для выбора только тех строк, где есть выражение `True`:

```
mat[mat[:, 0] % 2 == 1]
```

Так получим матрицу только с двумя строками — 1 и 3:

```
array([[ 3,  4,  5],  
       [ 9, 10, 11]])
```

## B.3. ЛИНЕЙНАЯ АЛГЕБРА

Одной из причин, по которой NumPy так популярен, является его поддержка операций линейной алгебры. NumPy делегирует все внутренние вычисления BLAS и LAPACK — проверенным временем библиотекам для эффективных низкоуровневых вычислений — и именно поэтому он невероятно быстр.

В этом разделе мы совершим краткий обзор операций линейной алгебры, которые нам понадобятся на протяжении всей книги. Мы начнем с наиболее распространенных из них: матричных и векторных умножений.

### B.3.1. Умножение

В линейной алгебре у нас есть несколько типов умножения:

- умножение вектора на другой вектор;
- умножение матрицы на вектор;
- умножение матрицы на другую матрицу.

Подробнее рассмотрим каждый из них и посмотрим, как это сделать в NumPy.

#### Умножение вектора на вектор

Умножение вектора на вектор требует двух векторов. Обычно это называется *скалярным произведением*; оно принимает два вектора и выдает *скаляр* — единственное число.

Предположим, у нас есть два вектора,  $u$  и  $v$ , каждый длиной  $n$ ; тогда скалярное произведение  $u$  и  $v$  равно

$$u^T v = \sum_{i=1}^n u_i v_i = u_1 v_1 + u_2 v_2 + \dots + u_n v_n.$$

#### ПРИМЕЧАНИЕ

В этом приложении элементы вектора длины  $n$  индексируются от 0 до  $n - 1$ : таким образом легче отобразить понятия из математической записи в NumPy.

Это напрямую переводится на Python. Если у нас есть два числовых массива  $u$  и  $v$ , то их скалярное произведение равно:

```
dot = 0

for i in range(n):
    dot = u[i] * v[i]
```

Конечно, мы можем воспользоваться преимуществами векторизованных операций в NumPy и вычислить его с помощью одностороннего выражения:

```
(u * v).sum()
```

Но это довольно распространенная операция, поэтому она реализована внутри NumPy в методе `dot`. Таким образом, чтобы вычислить точечное произведение, мы просто вызываем `dot`:

```
u.dot(v)
```

### Умножение матрицы на вектор

Другим типом умножения является умножение матрицы на вектор.

Предположим, у нас есть матрица  $X$  размера  $m$  на  $n$  и вектор  $u$  размера  $n$ . Умножив  $X$  на  $u$ , мы получим другой вектор размера  $m$  (рис. B.12):

$$Xu = v.$$

$$\begin{array}{|c|c|c|} \hline x_{00} & x_{01} & x_{02} \\ \hline x_{10} & x_{11} & x_{12} \\ \hline x_{20} & x_{21} & x_{22} \\ \hline x_{30} & x_{31} & x_{32} \\ \hline \end{array} \times \begin{array}{|c|} \hline u_0 \\ \hline u_1 \\ \hline u_2 \\ \hline \end{array} = \begin{array}{|c|} \hline v_0 \\ \hline v_1 \\ \hline v_2 \\ \hline v_3 \\ \hline \end{array}$$

**Рис. B.12.** Умножая матрицу  $4 \times 3$  на вектор длины 3, мы получаем вектор длины 4

Мы можем представить матрицу  $X$  как набор из  $n$  строк-векторов  $x_i$ , каждый из которых имеет размер  $m$  (рис. B.13).

$$\begin{array}{|c|c|c|} \hline x_{00} & x_{01} & x_{02} \\ \hline x_{10} & x_{11} & x_{12} \\ \hline x_{20} & x_{21} & x_{22} \\ \hline x_{30} & x_{31} & x_{32} \\ \hline \end{array} \rightarrow \begin{array}{|c|} \hline x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline \end{array}$$

**Рис. B.13.** Мы можем представить матрицу  $X$  как состоящую из четырех векторов строк  $x_i$ , каждый из которых имеет размер 3

Тогда мы можем представить матрично-векторное умножение  $Xu$  как  $m$  умножений вектора на вектор между каждой строкой  $x_i$  и вектором  $u$ . В результате получается другой вектор —  $v$  (рис. B.14).

$$\begin{array}{|c|} \hline x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline \end{array} \times \begin{array}{|c|} \hline u \\ \hline \end{array} = \begin{array}{|c|} \hline x_0^T u \\ \hline x_1^T u \\ \hline x_2^T u \\ \hline x_3^T u \\ \hline \end{array} = \begin{array}{|c|} \hline v_0 \\ \hline v_1 \\ \hline v_2 \\ \hline v_3 \\ \hline \end{array}$$

**Рис. B.14.** Умножение матрицы на вектор представляет собой набор умножений вектора на вектор: мы умножаем каждую строку  $x_i$  матрицы  $X$  на вектор  $u$  и в результате получаем вектор  $v$

Перевести эту идею на Python несложно:

```
v = np.zeros(m) ← Создает пустой вектор v
for i in range(m): ← Для каждой строки  $x_i$  из X
    v[i] = X[i].dot(u) ← Вычисляет  $i$ -й элемент из v как
                           скалярное произведение  $x_i^T u$ 
```

Как и при умножении вектора на вектор, мы можем использовать метод `dot` матрицы  $X$  (двумерный массив), чтобы умножить ее на вектор  $u$  (одномерный массив):

```
v = X.dot(u)
```

Результатом станет вектор  $v$  — одномерный массив NumPy.

### Умножение матрицы на матрицу

Наконец, у нас есть умножение матрицы на матрицу. Предположим, у нас две матрицы:  $X$  размера  $m$  на  $n$  и  $U$  размера  $n$  на  $k$ . Тогда результатом будет новая матрица  $V$  размером  $m$  на  $k$  (рис. B.15):

$$XU = V.$$

$$\begin{array}{|c|c|c|} \hline x_{00} & x_{01} & x_{02} \\ \hline x_{10} & x_{11} & x_{12} \\ \hline x_{20} & x_{21} & x_{22} \\ \hline x_{30} & x_{31} & x_{32} \\ \hline \end{array} \times \begin{array}{|c|c|} \hline u_{00} & u_{01} \\ \hline u_{10} & u_{11} \\ \hline u_{20} & u_{21} \\ \hline \end{array} = \begin{array}{|c|c|} \hline v_{00} & v_{01} \\ \hline v_{10} & v_{11} \\ \hline v_{20} & v_{21} \\ \hline v_{30} & v_{31} \\ \hline \end{array}$$

**Рис. B.15.** Умножая матрицу  $4 \times 3 X$  на матрицу  $3 \times 2 U$ , мы получаем матрицу  $4 \times 2 V$

Самый простой способ понять умножение матрицы на матрицу — рассматривать  $U$  как набор столбцов:  $u_0, u_1, \dots, u_{k-1}$  (рис. B.16).

$$\begin{array}{|c|c|} \hline u_{00} & u_{01} \\ \hline u_{10} & u_{11} \\ \hline u_{20} & u_{21} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|} \hline | & | \\ \hline u_0 & u_1 \\ \hline | & | \\ \hline \end{array}$$

**Рис. B.16.** Мы можем представить  $U$  как набор векторов-столбцов. В этом случае у нас есть два столбца:  $u_0$  и  $u_1$

Тогда умножение матрицы на матрицу  $XU$  представляет собой набор умножений матрицы на вектор  $Xu_i$ . Результатом каждого умножения будет вектор  $v_i$ , который является  $i$ -м столбцом итоговой матрицы  $V$  (рис. B.17):

$$v_i = Xu_i.$$

$$\boxed{X} \times \begin{vmatrix} | & | \\ u_0 & u_1 \\ | & | \end{vmatrix} = \begin{vmatrix} | & | \\ Xu_0 & Xu_1 \\ | & | \end{vmatrix} = \begin{vmatrix} | & | \\ v_0 & v_1 \\ | & | \end{vmatrix}$$

**Рис. 17.** Мы можем представить себе умножение матрицы на матрицу  $XU$  как набор умножений матрицы на вектор  $v_i = Xu_i$ , где  $u_i$  — столбцы  $U$ . Результатом будет матрица  $V$  со всеми  $v_i$ , сложенными вместе

Чтобы реализовать это в NumPy, можно просто написать следующее:

```
V = np.zeros((m, k))           | Создает пустую матрицу V
for i in range(k):            | Для каждого столбца  $u_i$  из  $U$ 
    vi = X.dot(U[:, i])        | Вычисляет  $v_i$  как
    V[:, i] = vi              | Передает  $v_i$  как  $i$ -й столбец  $V$  умножение матрицы
                                | на вектор  $X^* u_i$ 
```

Напомним, что  $U[:, i]$  означает получение  $i$ -го столбца. Затем мы умножаем  $X$  на этот столбец и получаем  $vi$ . С помощью  $V[:, i]$  и поскольку у нас есть присваивание ( $=$ ), мы перезаписываем  $i$ -й столбец  $V$  с помощью  $vi$ .

Конечно, в NumPy для этого есть короткая версия — это снова метод `dot`:

```
v = X.dot(u)
```

### B.3.2. Обратная матрица

Обратная квадратной матрице  $X$  — это матрица  $X^{-1}$ , такая, что  $X^{-1} X = I$ , где  $I$  — единичная матрица. Единичная матрица  $I$  не изменяет вектор, когда мы выполняем матрично-векторное умножение:

$$Iv = v.$$

Зачем она вообще нужна? Предположим, у нас есть система:

$$Ax = b.$$

Мы знаем матрицу  $A$  и итоговый вектор  $b$ , но не знаем вектор  $x$  — и хотим его найти. Другими словами, мы хотим *решить* эту систему.

## 462 Приложение B. Введение в NumPy

Один из возможных способов сделать это — выполнить такие действия:

- вычислить  $A^{-1}$ , которая является обратной  $A$ , а затем
- умножить обе части уравнения на обратную  $A^{-1}$ .

При этом мы получаем:

$$A^{-1}Ax = A^{-1}b.$$

Поскольку  $A^{-1}A = I$ , мы имеем:

$$Ix = A^{-1}b$$

или

$$x = A^{-1}b.$$

В NumPy для вычисления обратной матрицы мы используем `np.linalg.inv`:

```
A = np.array([
    [0, 1, 2],
    [1, 2, 3],
    [2, 3, 3]
])
Ainv = np.linalg.inv(A)
```

Для этой конкретной квадратной матрицы `A` можно вычислить ее обратную матрицу, поэтому `Ainv` имеет следующие значения:

```
array([[-3.,  3., -1.],
       [ 3., -4.,  2.],
       [-1.,  2., -1.]])
```

Мы можем убедиться, умножив матрицу на ее обратную матрицу, что получим единичную матрицу:

```
A.dot(Ainv)
```

Результатом действительно является единичная матрица:

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

### ПРИМЕЧАНИЕ

Если все, чего вы хотите, — это решить уравнение  $Ax = b$ , то на самом деле вам не нужно вычислять обратную матрицу. С вычислительной точки зрения вычисление обратной матрицы является дорогостоящей операцией. Вместо этого следует использовать `np.linalg.solve`, что на порядок быстрее:

```
b = np.array([1, 2, 3])
x = np.linalg.solve(A, b)
```

В этой книге при вычислении весов для линейной регрессии мы используем обратную матрицу для простоты: это облегчает понимание кода.

Существуют матрицы, которые невозможно инвертировать. Прежде всего, это относится к неквадратным матрицам. Кроме того, не все квадратные матрицы могут быть инвертированы: например, *сингулярные* матрицы.

Пытаясь инвертировать сингулярную матрицу в NumPy, мы получаем ошибку:

```
B = np.array([
    [0, 1, 1],
    [1, 2, 3],
    [2, 3, 5]
])

np.linalg.inv(B)
```

Этот код вызывает `LinAlgError`:

```
-----
LinAlgError                                Traceback (most recent call last)
<ipython-input-286-14528a9f848e> in <module>
      5 ]
      6
----> 7 np.linalg.inv(B)
<__array_function__ internals> in inv(*args, **kwargs)

<...>

LinAlgError: Singular matrix
```

### **B.3.3. Нормальное уравнение**

В главе 2 с помощью нормального уравнения мы вычисляли вектор весов для линейной регрессии. В этом подразделе кратко обрисуем, как вывести эту формулу, но не будем вдаваться в подробности. Получить дополнительную информацию можно в любом учебнике линейной алгебры.

Этот подраздел может показаться перегруженным математическими аспектами, и вы можете пропустить его: он не повлияет на ваше понимание книги. Если вы изучали нормальное уравнение и линейную регрессию в институте, но уже забыли большую их часть, то этот подраздел поможет освежить вашу память.

Предположим, у нас есть матрица  $X$  с наблюдениями и вектор  $y$  с результатами. Мы хотим найти такой вектор  $w$ , которого

$$Xw = y.$$

## 464 Приложение B. Введение в NumPy

Однако  $X$  не является квадратной матрицей, поэтому мы не можем просто инвертировать ее, и точного решения для этой системы не существует. Мы можем попытаться найти неточное решение и проделать следующий трюк. Мы умножим обе стороны на транспозицию  $X$ :

$$X^T X w = X^T y.$$

Теперь  $X^T X$  – это квадратная матрица, которую возможно инвертировать. Назовем эту матрицу  $C$ :

$$C = X^T X.$$

Уравнение превращается в

$$C w = X^T y.$$

В этом уравнении  $X^T y$  также является вектором: умножая матрицу на вектор, мы получаем вектор. Назовем его  $z$ . Итак, теперь у нас есть

$$C w = z.$$

Теперь у этой системы есть точное решение, являющееся наилучшим приближенным решением к системе, которую мы изначально хотели решить. Доказательство этого выходит за рамки книги. Более подробную информацию можно получить в учебнике.

Чтобы решить систему, мы можем инвертировать  $C$  и умножить на нее обе стороны:

$$C^{-1} C w = C^{-1} z$$

или

$$w = C^{-1} z.$$

Теперь у нас есть решение для  $w$ . Перепишем это в терминах исходных  $X$  и  $y$ :

$$w = (X^T X)^{-1} X^T y.$$

Это нормальное уравнение, которое находит наилучшее приближенное решение  $w$  для исходной системы  $Xw = y$ .

Это довольно просто перевести на NumPy:

```
C = X.T.dot(X)
Cinv = np.linalg.inv(C)
w = Cinv.dot(X.T).dot(y)
```

Теперь массив  $w$  содержит наилучшее приближенное решение системы.



# *Введение в Pandas*

---

Мы не ожидаем от читателей данной книги каких-либо знаний о Pandas. Однако мы широко используем эту библиотеку на протяжении всей книги. По ходу дела пытаемся объяснить код, но не всегда возможно осветить все подробности.

В этом приложении мы представляем более подробное введение в Pandas, охватывающее все функции, которые используются в главах книги.

## **Г.1. PANDAS**

Pandas – библиотека Python для работы с табличными данными. Это популярный и удобный инструмент для манипулирования данными, особенно полезный при подготовке данных для обучения моделей машинного обучения.

Если вы используете Anaconda, то в нем уже предустановлены Pandas. Если нет, то установите библиотеку с помощью pip:

```
pip install pandas
```

Чтобы поэкспериментировать с Pandas, создадим блокнот под названием appendix-d-pandas и будем использовать его для запуска кода из этого приложения.

Сначала нам нужно импортировать Pandas:

```
import pandas as pd
```

Как и в случае с NumPy, мы следуем соглашению и используем псевдоним pd вместо полного имени.

Мы начнем изучение Pandas с его основных структур данных: датафреймов и серий.

## Г.1.1. Датафрейм

В Pandas *датафрейм* — просто таблица: структура данных со строками и столбцами (рис. Г.1).

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

**Рис. Г.1.** Датафрейм в Pandas: таблица с пятью строками и восемью столбцами

Чтобы создать датафрейм, нам понадобятся некоторые данные, которые мы поместим в таблицу. Это может быть список списков с какими-то значениями:

```
data = [
    ['Nissan', 'Stanza', 1991, 138, 4, 'MANUAL', 'sedan', 2000],
    ['Hyundai', 'Sonata', 2017, None, 4, 'AUTOMATIC', 'Sedan', 27150],
    ['Lotus', 'Elise', 2010, 218, 4, 'MANUAL', 'convertible', 54990],
    ['GMC', 'Acadia', 2017, 194, 4, 'AUTOMATIC', '4dr SUV', 34450],
    ['Nissan', 'Frontier', 2017, 261, 6, 'MANUAL', 'Pickup', 32340],
]
```

Эти данные взяты из набора данных для прогнозирования цен, который мы используем в главе 2: у нас есть определенные характеристики автомобиля, такие как модель, марка, год выпуска и тип трансмиссии.

При создании датафрейма нам нужно знать, что содержит каждый из столбцов, поэтому мы создадим список с именами столбцов:

```
columns = [
    'Make', 'Model', 'Year', 'Engine HP', 'Engine Cylinders',
    'Transmission Type', 'Vehicle_Style', 'MSRP'
]
```

Теперь мы готовы создать из всего этого датафрейм. Для этого используем `pd.DataFrame`:

```
df = pd.DataFrame(data, columns=columns)
```

Он создает датафрейм с пятью строками и восемью столбцами (см. рис. Г.1).

Первое, что мы можем сделать с датафреймом, — взглянуть на первые несколько строк, чтобы получить представление о том, что находится внутри. Для этого мы используем метод `head`:

```
df.head(n=2)
```

Он показывает первые две строки датафрейма. Количество отображаемых строк регулируется параметром `n` (рис. Г.2).

```
df.head(n=2)
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150

**Рис. Г.2.** Предварительный просмотр содержимого датафрейма с помощью метода `head`

В качестве альтернативы для создания датафрейма мы можем использовать список словарей:

```
data = [
    {
        "Make": "Nissan",
        "Model": "Stanza",
        "Year": 1991,
        "Engine HP": 138.0,
        "Engine Cylinders": 4,
        "Transmission Type": "MANUAL",
        "Vehicle Style": "sedan",
        "MSRP": 2000
    },
    ... # more rows
]
```

```
df = pd.DataFrame(data)
```

В этом случае нам не нужно указывать имена столбцов: Pandas автоматически берет их из полей словарей.

## Г.1.2. Серии

Каждый столбец в датафрейме представляет собой *серию* — специальную структуру данных, содержащую значения одного типа. В некотором смысле это очень похоже на одномерные массивы NumPy.

Мы можем получить доступ к значениям столбца двумя способами. Первый — можем использовать точечную нотацию (рис. Г.3, А):

```
df.Make
```

Второй способ — использовать обозначения в скобках (рис. Г.3, Б):

```
df['Make']
```

## 468 Приложение Г. Введение в Pandas

df.Make		df['Make']	
0	Nissan	0	Nissan
1	Hyundai	1	Hyundai
2	Lotus	2	Lotus
3	GMC	3	GMC
4	Nissan	4	Nissan
Name: Make, dtype: object		Name: Make, dtype: object	
А. Точечная нотация		Б. Обозначение в квадратных скобках	

**Рис. Г.3.** Два способа доступа к столбцу датафрейма

Результат точно такой же: серия Pandas со значениями из столбца `Make`. Если имя столбца содержит пробелы или другие специальные символы, то мы можем использовать только обозначение в скобках. Например, получить доступ к колонке `HP` двигателя можем лишь с помощью скобок:

```
df['Engine HP']
```

Обозначение в скобках также является более гибким. Мы можем сохранить имя столбца в переменной и использовать его для доступа к содержимому:

```
col_name = 'Engine HP'  
df[col_name]
```

Если нам нужно выбрать подмножество столбцов, то мы снова используем скобки, но со списком имен вместо единственной строки:

```
df[['Make', 'Model', 'MSRP']]
```

Это возвращает датафрейм только с тремя столбцами (рис. Г.4).

df[['Make', 'Model', 'MSRP']]			
	Make	Model	MSRP
0	Nissan	Stanza	2000
1	Hyundai	Sonata	27150
2	Lotus	Elise	54990
3	GMC	Acadia	34450
4	Nissan	Frontier	32340

**Рис. Г.4.** Чтобы выбрать подмножество столбцов датафрейма, используйте скобки со списком имен

Чтобы добавить столбец в датафрейм, мы также используем обозначение в скобках:

```
df['id'] = ['nis1', 'hyu1', 'lot2', 'gmc1', 'nis2']
```

У нас есть пять строк в датафрейме, поэтому список со значениями также должен содержать пять значений. В результате мы получаем еще один столбец `id` (рис. Г.5).

```
df['id'] = ['nis1', 'hyu1', 'lot2', 'gmc1', 'nis2']
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP	<code>id</code>
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000	nis1
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150	hyu1
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990	lot2
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450	gmc1
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340	nis2

**Рис. Г.5.** Чтобы добавить новый столбец, используйте обозначения в скобках

В этом случае идентификатор не существовал, поэтому мы добавили новый столбец в конец датафрейма. Если идентификатор существует, то этот код перезаписывает существующие значения:

```
df['id'] = [1, 2, 3, 4, 5]
```

Теперь содержимое столбца `id` изменяется (рис. Г.6).

```
df['id'] = [1, 2, 3, 4, 5]
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP	<code>id</code>
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000	1
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150	2
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990	3
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450	4
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340	5

**Рис. Г.6.** Чтобы изменить содержимое столбца, также используйте обозначения в квадратных скобках

Чтобы удалить столбец, используйте оператор `del`:

```
del df['id']
```

После его запуска этот столбец исчезает из датафрейма.

### Г.1.3. Index

Как датафрейм (рис. Г.7, А), так и серия (рис. Г.7, Б) содержат слева номера; эти номера называются *индексом*. Индекс описывает, как мы можем получить доступ к строкам из датафрейма (или серии).

## 470 Приложение Г. Введение в Pandas

(A) `df[['Make', 'Model', 'MSRP']]` (B) `df.Make`

	Make	Model	MSRP
0	Nissan	Stanza	2000
1	Hyundai	Sonata	27150
2	Lotus	Elise	54990
3	GMC	Acadia	34450
4	Nissan	Frontier	32340

[0] Nissan  
[1] Hyundai  
[2] Lotus  
[3] GMC  
[4] Nissan  
Name: Make, dtype: object

**Рис. Г.7.** Как датафрейм, так и серия имеют индекс — цифры слева

Мы можем получить индекс датафрейма, используя свойство `index`:

```
df.index
```

Поскольку мы не указывали индекс при создании датафрейма, он использует индекс по умолчанию — серию автоматически добавляемых чисел, начинающихся с 0:

```
RangeIndex(start=0, stop=5, step=1)
```

Индекс ведет себя так же, как объект серии, поэтому все, что работает для серии, работает и для индекса.

Серия имеет только один индекс, а у датафрейма их два: один для доступа к строкам, а другой для доступа к столбцам. Мы уже использовали индекс для столбцов при выборе отдельных столбцов из датафрейма:

`df['Make']` ← Использует индекс столбца для получения столбца Make

Чтобы получить имена столбцов, мы используем свойство `columns` (рис. Г.8):

```
df.columns
```

```
Index(['Make', 'Model', 'Year', 'Engine HP', 'Engine Cylinders',
       'Transmission Type', 'Vehicle_Style', 'MSRP'],
      dtype='object')
```

**Рис. Г.8.** Свойство `columns` содержит имена столбцов

### Г.1.4. Доступ к строкам

Мы можем получить доступ к строкам двумя способами: используя `iloc` и `loc`.

Начнем с `iloc`. Мы используем это свойство для доступа к строкам датафрейма, используя их позиции. Например, чтобы получить доступ к первой строке датафрейма, используйте индекс 0:

```
df.iloc[0]
```

Этот код возвращает содержимое первой строки:

```
Make          Nissan
Model         Stanza
Year          1991
Engine HP     138
Engine Cylinders 4
Transmission Type MANUAL
Vehicle_Style  sedan
MSRP          2000
Name: 0, dtype: object
```

Чтобы получить подмножество строк, передайте список с целыми числами — номерами строк:

```
df.iloc[[2, 3, 0]]
```

В результате получается еще один датафрейм, содержащий только нужные нам строки (рис. Г.9).

```
df.iloc[[2, 3, 0]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

**Рис. Г.9.** Использование iloc для доступа к строкам датафрейма

Мы можем использовать `iloc` для перетасовки содержимого датафрейма. В нашем датафрейме у нас пять строк. Следовательно, мы можем создать список целых чисел от 0 до 4 и перетасовать его. Затем мы можем использовать перетасованный список в `iloc`; таким образом мы получим датафрейм со всеми перетасованными строками.

Реализуем это. Сначала мы создаем диапазон размера 5 с помощью NumPy:

```
import numpy as np

idx = np.arange(5)
```

Код создает массив с целыми числами от 0 до 4:

```
array([0, 1, 2, 3, 4])
```

Теперь мы можем перетасовать этот массив:

```
np.random.seed(2)
np.random.shuffle(idx)
```

## 472 Приложение Г. Введение в Pandas

В результате мы получаем

```
array([2, 4, 1, 3, 0])
```

Наконец мы используем этот массив при вызове `iloc`, чтобы получить строки в перемешанном порядке:

```
df.iloc[idx]
```

В результате строки переупорядочиваются в соответствии с номерами в `idx` (рис. Г.10).

```
df.iloc[idx]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

**Рис. Г.10.** Использование `iloc` для перетасовки строк датафрейма

Это не изменяет датафрейм, который у нас есть в `df`. Но мы можем присвоить переменной `df` новый датафрейм:

```
df = df.iloc[idx]
```

В результате `df` теперь содержит перетасованный датафрейм.

В этом перетасованном фрейме данных мы все еще можем использовать `iloc` для получения строк, используя их позиции. Например, если мы передадим `[0, 1, 2]` в `iloc`, то получим первые три строки (рис. Г.11).

```
df.iloc[[0, 1, 2]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150

**Рис. Г.11.** При использовании `iloc` мы получаем строки по их позиции

Однако вы, вероятно, заметили, что цифры слева больше не являются последовательными: при перетасовке датафрейма мы перетасовали и индексы (рис. Г.12).

df

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
1	Hyundai	Sonata	2017	Nan	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

**Рис. Г.12.** При перетасовке строк датафрейма мы также меняем индексы: фрейм больше не является последовательным

Взглянем на индексы:

```
df.index
```

Теперь все по-другому:

```
Int64Index([2, 4, 1, 3, 0], dtype='int64')
```

Чтобы использовать эти индексы для доступа к строкам, нам понадобится `loc` вместо `iloc`. Например:

```
df.loc[[0, 1]]
```

В результате мы получаем датафрейм со строками, проиндексированными 0 и 1 — последнюю строку и строку в середине (рис. Г.13).

df.loc[[0, 1]]

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	Nan	4	AUTOMATIC	Sedan	27150

**Рис. Г.13.** При использовании loc мы получаем строки, используя индекс, а не позицию

Это серьезное отличие от `iloc`: `iloc` не использует индексы. Сравним их:

```
df.iloc[[0, 1]]
```

В этом случае мы также получаем датафрейм с двумя строками, но это первые две строки с индексами 2 и 4 (рис. Г.14).

df.iloc[[0, 1]]

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

**Рис. Г.14.** В отличие от loc, iloc получает строки по позиции, а не по индексу. В этом случае мы получаем строки в позициях 0 и 1 (с индексами 2 и 4 соответственно)

Таким образом, `iloc` вообще не принимает индекс в расчет; функция использует только фактическую позицию.

Можно заменить индекс и вернуть ему значение по умолчанию. Для этого мы можем использовать `reset_index`:

```
df.reset_index(drop=True)
```

Команда создает новый датафрейм с последовательной индексацией (рис. Г.15).

	Make	Model	Year	Engine HP
2	Lotus	Elise	2010	218.0
4	Nissan	Frontier	2017	261.0
1	Hyundai	Sonata	2017	NaN
3	GMC	Acadia	2017	194.0
0	Nissan	Stanza	1991	138.0

	Make	Model	Year	Engine HP
0	Lotus	Elise	2010	218.0
1	Nissan	Frontier	2017	261.0
2	Hyundai	Sonata	2017	NaN
3	GMC	Acadia	2017	194.0
4	Nissan	Stanza	1991	138.0

**Рис. Г.15.** Мы можем сбросить индексацию к последовательной с помощью `reset_index`

## Г.1.5. Разделение датафрейма

Мы также можем использовать `iloc` для выбора подмножеств датафрейма. Предположим, требуется разделить датафрейм на три части: обучающую, проверочную и тестовую. Мы будем использовать 60 % данных для обучения (три строки), 20 % для проверки (одна строка) и 20 % для тестирования (одна строка):

```
n_train = 3
n_val = 1
n_test = 1
```

Для выбора диапазона строк мы используем оператор среза (`:`). Это работает для датафреймов так же, как и для списков.

Таким образом, для разделения датафрейма мы делаем следующее:

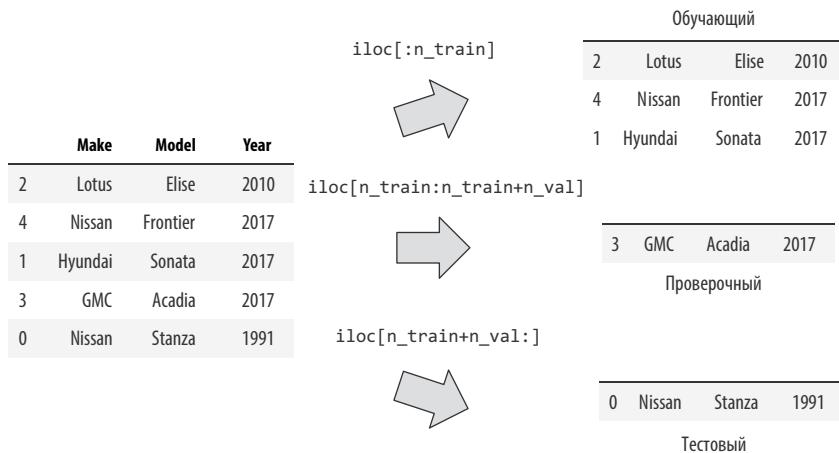
```
df_train = df.iloc[:n_train]
df_val = df.iloc[n_train:n_train+n_val]
df_test = df.iloc[n_train+n_val:]
```

В ① мы получаем обучающий набор: `iloc[:n_train]` выбирает строки от начала датафрейма до строки перед `n_train`. Для `n_train=3` он выбирает строки 0, 1 и 2. Стока 3 не включена.

В ❷ мы получаем проверочный набор: `iloc[n_train:n_train+n_val]` выбирает строки от 3 до  $3 + 1 = 4$ . Выбор не включительный, поэтому берется только строка 3.

В ❸ мы получаем тестовый набор: `iloc[n_train+n_val:]` выбирает строки из  $3 + 1 = 4$  до конца датафрейма. В нашем случае это всего лишь строка 4.

В результате мы имеем три датафрейма (рис. Г.16).



**Рис. Г.16.** Использование `iloc` с оператором двоеточия для разделения датафрейма на обучающий, проверочный и тестовый датафреймы

Дополнительную информацию о срезах в Python можно найти в приложении Б.

Мы рассмотрели основные структуры данных Pandas, и теперь разберемся, что мы можем с ними делать.

## Г.2. ОПЕРАЦИИ

Pandas – отличный инструмент для манипулирования данными, поддерживающий широкий спектр операций. Можно разбить эти операции на поэлементные, суммирующие, фильтрацию, сортировку, группировку и многое другое. В данном разделе мы рассмотрим все эти операции.

### Г.2.1. Операции по элементам

В Pandas серии поддерживают *поэлементные* операции. Так же, как и в NumPy, поэлементные операции применяются к каждому элементу в серии, и в результате мы получаем другую серию.

## 476 Приложение Г. Введение в Pandas

Все основные арифметические операции выполняются поэлементно: сложение (+), вычитание (-), умножение (\*) и деление (/). Поэлементные операции не требуют написания никаких циклов: Pandas делает это за нас.

Например, мы можем умножить каждый элемент серии на 2:

```
df['Engine HP'] * 2
```

В результате получается еще одна серия с каждым элементом, умноженным на 2 (рис. Г.17).

df['Engine HP']		df['Engine HP'] * 2	
0	218.0	0	436.0
1	261.0	1	522.0
2	NaN	2	NaN
3	194.0	3	388.0
4	138.0	4	276.0
Name: Engine HP, dtype: float64		Name: Engine HP, dtype: float64	

А. Оригинальная серия      Б. Результат умножения

**Рис. Г.17.** Как и в случае с массивами NumPy, все основные арифметические операции для серий выполняются поэлементно

Как и в случае с арифметикой, логические операции также выполняются поэлементно:

```
df['Year'] > 2000
```

Данное выражение возвращает серию Boolean со значением True для элементов, превышающих 2000 (рис. Г.18).

df['Year']		df['Year'] > 2000	
0	2010	0	True
1	2017	1	True
2	2017	2	True
3	2017	3	True
4	1991	4	False
Name: Year, dtype: int64		Name: Year, dtype: bool	

А. Оригинальная серия      Б. Результат умножения

**Рис. Г.18.** Логические операции применяются поэлементно: в результатах содержатся значения True для всех элементов, удовлетворяющих условию

Мы можем комбинировать несколько логических операций с логическим «И» (&) или логическим «ИЛИ» (|):

```
(df['Year'] > 2000) & (df['Make'] == 'Nissan')
```

Результатом также является серия. Логические операции полезны для фильтрации, которую мы рассмотрим далее.

## Г.2.2. Фильтрация

Часто нам нужно выбрать подмножество строк в соответствии с некоторыми критериями. Для этого мы используем логические операции вместе с обозначением в скобках.

Например, чтобы выбрать все автомобили Nissan, поместите условие в скобки:

```
df[df['Make'] == 'Nissan']
```

В результате у нас будет еще один датафрейм, содержащий только Nissan (рис. Г.19).

```
df[df['Make'] == 'Nissan']
```

Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
1	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup 32340
4	Nissan	Stanza	1991	138.0	4	MANUAL	sedan 2000

**Рис. Г.19.** Чтобы отфильтровать строки, поместите условие фильтрации в квадратные скобки

Если нам нужно более сложное условие выбора, то мы объединяем несколько условий с логическими операторами, такими как «И» (&) и «ИЛИ» (|).

Например, для выбора автомобилей, выпущенных после 2010 года с автоматической коробкой передач, мы используем «И» (рис. Г.20):

```
df[(df['Year'] > 2010) & (df['Transmission Type'] == 'AUTOMATIC')]
```

```
df[(df['Year'] > 2010) & (df['Transmission Type'] == 'AUTOMATIC')]
```

Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
2	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan 27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV 34450

**Рис. Г.20.** Чтобы использовать несколько критериев выбора, объедините их логическим И (&)

## Г.2.3. Операции со строками

Для массивов NumPy можно выполнять только поэлементные арифметические и логические операции, а Pandas поддерживает и операции со строками: использование нижнего регистра, замену подстрок и все другие операции, которые можно выполнять со строковыми объектами.

## 478 Приложение Г. Введение в Pandas

Взглянем на `Vehicle_Style`, который является одним из столбцов в датафрейме. Мы видим некоторые несоответствия в данных: иногда имена начинаются со строчных букв, а иногда с прописных (рис. Г.21).

```
df['Vehicle_Style']  
0    convertible  
1        Pickup  
2        Sedan  
3      4dr SUV  
4       sedan  
Name: Vehicle_Style, dtype: object
```

**Рис. Г.21.** Столбец `Vehicle_Style` с несоответствиями в данных

Чтобы решить эту проблему, мы можем перевести все в нижний регистр. Для обычных строк Python мы бы использовали функцию `lower` и применили ее ко всем элементам серии. В Pandas вместо написания цикла мы используем специальный инструмент доступа `str` — он выполняет операции со строками поэлементно и позволяет нам избежать явного написания цикла `for`:

```
df['Vehicle_Style'].str.lower()
```

В результате получается новая серия со всеми строками, оформленными в нижнем регистре (рис. Г.22).

```
df['Vehicle_Style'].str.lower()  
0    convertible  
1        pickup  
2        sedan  
3      4dr suv  
4       sedan  
Name: Vehicle_Style, dtype: object
```

**Рис. Г.22.** Чтобы перевести все строки серии в нижний регистр, используйте `lower`

Кроме того, можно связать несколько операций со строками, используя средство доступа `str` несколько раз (рис. Г.23):

```
df['Vehicle_Style'].str.lower().str.replace(' ', '_')
```

```
df['Vehicle_Style'].str.lower().str.replace(' ', '_')  
0    convertible  
1        pickup  
2        sedan  
3      4dr_suv  
4       sedan  
Name: Vehicle_Style, dtype: object
```

**Рис. Г.23.** Для замены символов во всех строках ряда используйте метод `replace`. Можно объединить несколько методов в одну строку

Здесь мы изменяем все буквы на строчные и одновременно заменяем пробелы символами подчеркивания.

Имена столбцов нашего датафрейма также не согласованы: иногда в них присутствуют пробелы, а иногда — подчеркивания (рис. Г.24).

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	MSRP
0	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
1	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
2	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

**Рис. Г.24.** Датафрейм: имена столбцов не согласованы

Мы также можем использовать строковые операции для нормализации имен столбцов:

```
df.columns.str.lower().str.replace(' ', '_')
As a result, we have:
Index(['make', 'model', 'year', 'engine_hp', 'engine_cylinders',
       'transmission_type', 'vehicle_style', 'msrp'],
      dtype='object')
```

Этот код возвращает новые имена, но не изменяет имена столбцов датафрейма. Чтобы изменить их, нам нужно присвоить результаты `df.columns`:

```
df.columns = df.columns.str.lower().str.replace(' ', '_')
```

Сделав это, мы поменяем имена столбцов (рис. Г.25).

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
1	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340
2	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000

**Рис. Г.25.** Датафрейм после нормализации имен столбцов

Мы можем решить подобные проблемы с несогласованностью во всех столбцах нашего датафрейма. Для этого нам нужно выделить все столбцы со строковыми данными и нормализовать их.

## 480 Приложение Г. Введение в Pandas

Чтобы выбрать все строки, мы можем использовать свойство `dtype` датафрейма (рис. Г.26).

```
df.dtypes
make          object
model         object
year           int64
engine_hp     float64
engine_cylinders  int64
transmission_type   object
vehicle_style    object
msrp           int64
dtype: object
```

**Рис. Г.26.** Свойство `dtypes` возвращает типы каждого столбца датафрейма

Для всех столбцов `strings` установлен `dtype object`. Следовательно, если мы хотим выделить именно их, то используем фильтрацию:

```
df.dtypes[df.dtypes == 'object']
```

Таким образом мы получаем серию только со столбцами, где `dtype` равен `object` (рис. Г.27).

```
df.dtypes[df.dtypes == 'object']
make          object
model         object
transmission_type  object
vehicle_style    object
dtype: object
```

**Рис. Г.27.** Чтобы получить только столбцы со строками, выберите `dtype object`

Фактические имена хранятся в индексе, поэтому нам нужно их получить:

```
df.dtypes[df.dtypes == 'object'].index
```

Это дает нам следующие имена столбцов:

```
Index(['make', 'model', 'transmission_type', 'vehicle_style'], dtype='object')
```

Теперь мы можем с помощью полученного списка перебирать строки столбцов и применять нормализацию к каждому столбцу по отдельности:

```
string_columns = df.dtypes[df.dtypes == 'object'].index

for col in string_columns:
    df[col] = df[col].str.lower().str.replace(' ', '_')
```

Вот что мы имеем после запуска кода (рис. Г.28).

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	lotus	elise	2010	218.0	4	manual	convertible	54990
1	nissan	frontier	2017	261.0	6	manual	pickup	32340
2	hyundai	sonata	2017	NaN	4	automatic	sedan	27150
3	gmc	acadia	2017	194.0	4	automatic	4dr_suv	34450
4	nissan	stanza	1991	138.0	4	manual	sedan	2000

**Рис. Г.28.** Как имена столбцов, так и значения нормализованы:  
имена записаны в нижнем регистре, а пробелы заменены подчеркиванием

Далее мы рассмотрим другой тип операций: суммирующие операции.

## Г.2.4. Суммирующие операции

Как и в NumPy, в Pandas есть поэлементные операции, которые создают новую серию, а также операции суммирования, которые создают сводку — одно или несколько чисел.

Операции суммирования весьма полезны для проведения исследовательского анализа данных. Для числовых полей операции аналогичны тем, что мы имеем в NumPy. Например, чтобы вычислить среднее значение всех значений в столбце, мы используем метод `mean`:

```
df.msrp.mean()
```

Другие методы включают в себя

- `sum` — вычисляет сумму всех значений;
- `min` — получает наименьшее число в серии;
- `max` — получает наибольшее число в серии;
- `std` — вычисляет стандартное отклонение.

Вместо того чтобы вычислять эти вещи по отдельности, мы можем использовать `describe`, чтобы получить все значения сразу:

```
df.msrp.describe()
```

Он создает сводку с количеством строк, средним значением, минимальным и максимальным значением, а также стандартным отклонением и другими характеристиками:

count	5.000000
mean	30186.000000
std	18985.044904
min	2000.000000

```
25%      27150.00000
50%      32340.00000
75%      34450.00000
max      54990.00000
Name: msrp, dtype: float64
```

Когда мы вызываем `mean` для всего датафрейма, он вычисляет среднее значение для всех числовых столбцов:

```
df.mean()
```

В нашем случае у нас четыре числовых столбца, поэтому мы получаем среднее значение для каждого:

```
year           2010.40
engine_hp     202.75
engine_cylinders  4.40
msrp          30186.00
dtype: float64
```

Аналогично мы можем использовать `describe` для датафрейма:

```
df.describe()
```

Поскольку `describe` уже возвращает серию, когда мы вызываем его для датафрейма, мы также получаем датафрейм (рис. Г.29).

	year	engine_hp	engine_cylinders	msrp
count	5.00	4.00	5.00	5.00
mean	2010.40	202.75	4.40	30186.00
std	11.26	51.30	0.89	18985.04
min	1991.00	138.00	4.00	2000.00
25%	2010.00	180.00	4.00	27150.00
50%	2017.00	206.00	4.00	32340.00
75%	2017.00	228.75	4.00	34450.00
max	2017.00	261.00	6.00	54990.00

**Рис. Г.29.** Чтобы получить сводную статистику по всем числовым характеристикам, используйте метод `describe`

## Г.2.5. Отсутствующие значения

Ранее мы не заостряли на этом внимание, но в наших данных отсутствует значение: мы не знаем значение `engine_hp` для строки 2 (рис. Г.30).

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	lotus	elise	2010	218.0	4	manual	convertible	54990
1	nissan	frontier	2017	261.0	6	manual	pickup	32340
2	hyundai	sonata	2017	Nan	4	automatic	sedan	27150
3	gmc	acadia	2017	194.0	4	automatic	4dr_suv	34450
4	nissan	stanza	1991	138.0	4	manual	sedan	2000

**Рис. Г.30.** В нашем датафрейме отсутствует одно значение

Мы можем узнать, какие значения отсутствуют, используя метод `isnull`:

```
df.isnull()
```

Метод возвращает новый датафрейм, где ячейка имеет значение `True`, если соответствующее значение отсутствует в исходном датафрейме (рис. Г.31).

```
df.isnull()
```

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False
2	False	False	False	True	False	False	False	False
3	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False

**Рис. Г.31.** Чтобы найти недостающие значения, используйте метод `isnull`

Однако когда у нас большие датафреймы, просмотр всех значений непрактичен.

Мы можем легко суммировать их, запустив метод `sum` для результатов:

```
df.isnull().sum()
```

Он возвращает серию с количеством пропущенных значений в столбце. В нашем случае только `engine_hp` имеет отсутствующие значения (рис. Г.32).

```
df.isnull().sum()
```

make	0
model	0
year	0
engine_hp	1
engine_cylinders	0
transmission_type	0
vehicle_style	0
msrp	0
dtype: int64	

**Рис. Г.32.** Чтобы найти столбцы с отсутствующими значениями, используйте `isnull`, за которым следует `sum`

## 484 Приложение Г. Введение в Pandas

Чтобы заменить отсутствующие значения какими-то фактическими значениями, мы используем `fillna`. Например, мы можем заполнить недостающие значения нулями:

```
df.engine_hp.fillna(0)
```

В результате мы получаем новую серию, в которой `NAN` заменяются на `0`:

```
0    218.0
1    261.0
2     0.0
3    194.0
4    138.0
Name: engine_hp, dtype: float64
```

В качестве альтернативы мы можем заменить его, получив среднее значение:

```
df.engine_hp.fillna(df.engine_hp.mean())
```

В этом случае `NAN` заменяются средним значением:

```
0    218.00
1    261.00
2    202.75
3    194.00
4    138.00
Name: engine_hp, dtype: float64
```

Метод `fillna` возвращает новую серию. Таким образом, если нам нужно удалить недостающие значения из нашего датафрейма, то нужно занести результаты обратно:

```
df.engine_hp = df.engine_hp.fillna(df.engine_hp.mean())
```

Теперь мы получаем датафрейм без пропущенных значений (рис. Г.33).

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	lotus	elise	2010	218.00	4	manual	convertible	54990
1	nissan	frontier	2017	261.00	6	manual	pickup	32340
2	hyundai	sonata	2017	202.75	4	automatic	sedan	27150
3	gmc	acadia	2017	194.00	4	automatic	4dr_suv	34450
4	nissan	stanza	1991	138.00	4	manual	sedan	2000

**Рис. Г.33.** Датафрейм без пропущенных значений

## Г.2.6. Сортировка

Операции, которые мы рассмотрели ранее, в основном использовались для серий. Мы также можем выполнять операции с датафреймами.

Сортировка — одна из таких операций: она перестраивает строки в датафрейме таким образом, чтобы они были отсортированы по значениям некоторого столбца (или нескольких столбцов).

Например, отсортируем датафрейм по MSRP. Для этого мы используем метод `sort_values`:

```
df.sort_values(by='msrp')
```

Результатом будет новый датафрейм, в котором строки сортируются от наименьшего MSRP (2000) до наибольшего (54990) (рис. Г.34).

```
df.sort_values(by='msrp')
```

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
4	nissan	stanza	1991	138.00	4	manual	sedan	2000
2	hyundai	sonata	2017	202.75	4	automatic	sedan	27150
1	nissan	frontier	2017	261.00	6	manual	pickup	32340
3	gmc	acadia	2017	194.00	4	automatic	4dr_suv	34450
0	lotus	elise	2010	218.00	4	manual	convertible	54990

**Рис. Г.34.** Для сортировки строк датафрейма используйте `sort_values`

Если мы хотим, чтобы первыми появлялись наибольшие значения, то задаем параметру `ascending` значение `False`:

```
df.sort_values(by='msrp', ascending=False)
```

Теперь у нас есть MSRP 54990 в первой строке и 2000 в последней (рис. Г.35).

```
df.sort_values(by='msrp', ascending=False)
```

	make	model	year	engine_hp	engine_cylinders	transmission_type	vehicle_style	msrp
0	lotus	elise	2010	218.00	4	manual	convertible	54990
3	gmc	acadia	2017	194.00	4	automatic	4dr_suv	34450
1	nissan	frontier	2017	261.00	6	manual	pickup	32340
2	hyundai	sonata	2017	202.75	4	automatic	sedan	27150
4	nissan	stanza	1991	138.00	4	manual	sedan	2000

**Рис. Г.35.** Чтобы отсортировать строки датафрейма в порядке убывания, используйте `ascending=False`

## Г.2.7. Группировка

Pandas предлагает довольно много операций суммирования: `sum`, `mean` и многие другие. Ранее мы видели, как применить их для вычисления сводки по всему датафрейму. Однако иногда это нужно сделать для каждой группы — например, рассчитать среднюю цену для каждого типа передачи.

## 486 Приложение Г. Введение в Pandas

В SQL мы бы написали что-то наподобие этого:

```
SELECT
    transmission_type,
    AVG(msrp)
FROM
    cars
GROUP BY
    transmission_type;
```

В Pandas мы используем метод `groupby`:

```
df.groupby('transmission_type').msrp.mean()
```

В результате получается средняя цена за тип трансмиссии:

```
transmission_type
automatic    30800.000000
manual       29776.666667
Name: msrp, dtype: float64
```

Если бы в SQL нам требовалось также вычислить количество записей для каждого типа вместе со средней ценой, то мы бы добавили еще один оператор в `SELECT`:

```
SELECT
    transmission_type,
    AVG(msrp),
    COUNT(msrp)
FROM
    cars
GROUP BY
    transmission_type
```

В Pandas мы используем `groupby`, за которым следует `agg` (сокращение от «агрегировать»):

```
df.groupby('transmission_type').msrp.agg(['mean', 'count'])
```

В результате мы получаем датафрейм (рис. Г.36).

	mean	count
transmission_type		
automatic	30800.000000	2
manual	29776.666667	3

**Рис. Г.36.** При группировании мы можем применить несколько агрегатных функций, используя метод `agg`

Pandas является довольно эффективным инструментом для манипулирования данными и часто используется для их подготовки перед обучением модели машинного обучения. Информация из этого приложения поможет вам разобраться в коде, представленном в нашей книге.

# AWS SageMaker

---

AWS SageMaker — набор сервисов от AWS, связанных с машинным обучением. SageMaker позволяет легко создать сервер на AWS с установленным на нем Jupyter. Блокноты уже настроены: в них есть большинство необходимых нам библиотек, включая NumPy, Pandas, Scikit-learn и TensorFlow, поэтому мы можем просто использовать их для своих проектов!

## Д.1. БЛОКНОТЫ AWS SAGEMAKER

Есть две причины, по которым блокноты SageMaker очень полезны при обучении нейронных сетей:

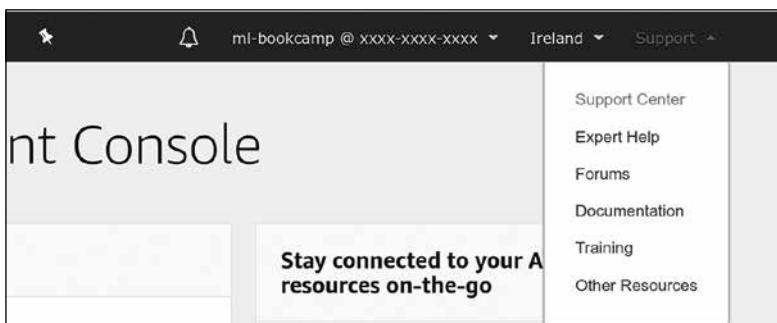
- нам не нужно беспокоиться о настройке TensorFlow и всех библиотек;
- можно арендовать компьютер с графическим процессором, который позволит обучать нейронные сети намного быстрее.

Чтобы использовать графический процессор, следует настроить квоты по умолчанию. В следующем подразделе мы разберемся, как это сделать.

### Д.1.1. Увеличение лимитов квот графического процессора

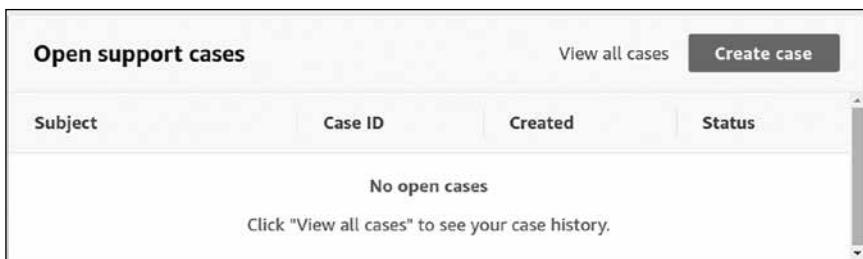
Каждая учетная запись в AWS имеет ограничения по квотам. Например, если наш лимит квоты на количество экземпляров с графическими процессорами равен 10, то мы не можем запросить одиннадцатый экземпляр с графическим процессором. По умолчанию лимит квоты равен нулю, что означает невозможность арендовать машину с графическим процессором без изменения пределов квот.

Чтобы запросить увеличение, откройте центр поддержки в консоли AWS: нажмите **Support** в правом верхнем углу и выберите **Support Center** (рис. Д.1).



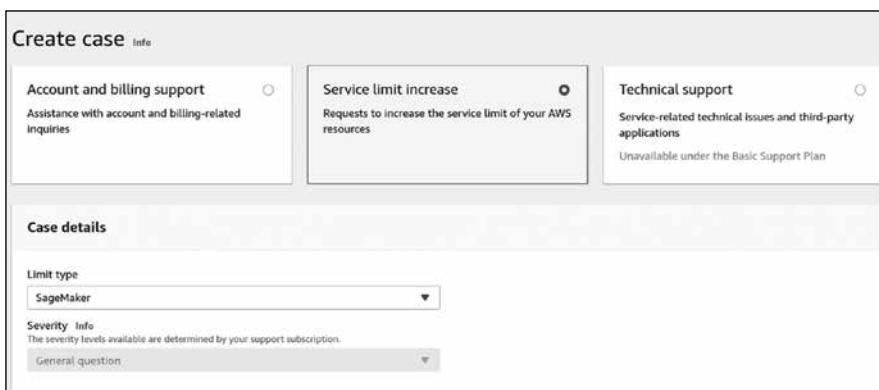
**Рис. Д.1.** Чтобы открыть центр поддержки, нажмите Support ▶ Support Center

Далее нажмите кнопку Create Case (рис. Д.2).



**Рис. Д.2.** В центре поддержки нажмите кнопку Create Case

Теперь выберите опцию **Service limit increase**. В разделе **Case details** выберите **SageMaker** из выпадающего списка **Limit type** (рис. Д.3).



**Рис. Д.3.** При создании нового обращения выберите Service Limit Increase ▶ SageMake

После этого заполните форму увеличения квоты (рис. Д.4):

- Region — выберите ближайший к вам или самый дешевый. Вы можете ознакомиться с ценами здесь: <https://aws.amazon.com/sagemaker/pricing/>;
- Resource type — SageMaker Notebooks;
- Limit — ml.p2.xlarge instances для машины с одним графическим процессором;
- New limit value — 1.

The screenshot shows a configuration interface for increasing a quota. It has four main sections: 'Region' (EU (Ireland)), 'Resource Type' (SageMaker Notebooks), 'Limit' (ml.p2.xlarge Instances), and 'New limit value' (1).

**Рис. Д.4.** Увеличьте лимит для ml.p2.xlarge до одного экземпляра

Наконец опишите, почему вам необходимо увеличить лимиты квот. Например, вы можете ввести «Я бы хотел обучить нейронную сеть с помощью графического процессора» (рис. Д.5).

The screenshot shows the 'Case description' section of the quota increase request. It includes a 'Use case description' input field containing the text 'I'd like to train a neural network using a GPU machine'. Below the input field is a note stating 'Maximum 5000 characters (4946 remaining)'.

**Рис. Д.5.** Нам нужно объяснить, почему мы хотим увеличить лимит

Все готово, теперь нажмите **Submit**.

После этого мы увидим некоторые детали запроса. Вернувшись в Support Center, мы увидим новое обращение в списке открытых (рис. Д.6).

Open support cases		View all cases		Create case
Subject	Case ID	Created	Status	
Limit Increase: SageMaker	7403143411	29 seconds ago	Unassigned	

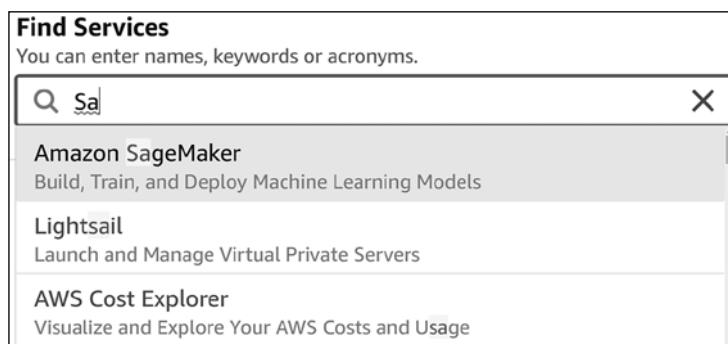
**Рис. Д.6.** Список открытых обращений в службу поддержки

Обычно обработка запроса и увеличение лимитов занимает от одного до двух дней.

Как только лимит будет увеличен, мы сможем создать экземпляр Jupyter Notebook с графическим процессором.

## Д.1.2. Создание экземпляра блокнота

Чтобы создать блокнот Jupyter в SageMaker, сначала найдите SageMaker в списке сервисов (рис. Д.7).



**Рис. Д.7.** Чтобы найти SageMaker, введите SageMaker в поле поиска

### ПРИМЕЧАНИЕ

На блокноты SageMaker не распространяется бесплатный уровень, поэтому аренда блокнота Jupyter стоит денег.

Для экземпляра с одним графическим процессором (ml.p2.xlarge) стоимость одного часа на момент написания составляет

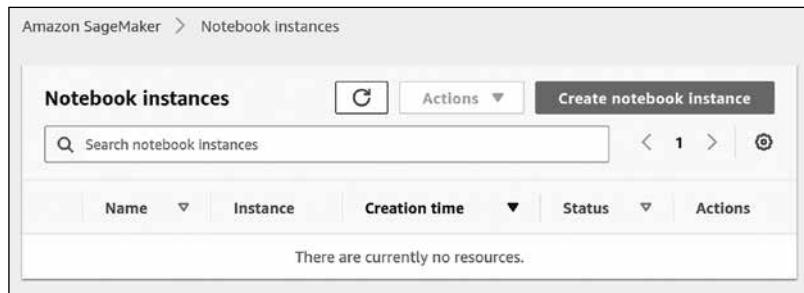
- во Франкфурте: 1856 долларов;
- в Ирландии: 1361 долларов;
- в Северной Вирджинии: 1,26 долларов.

Завершение проекта из главы 7 потребует от одного до двух часов.

### **ПРИМЕЧАНИЕ**

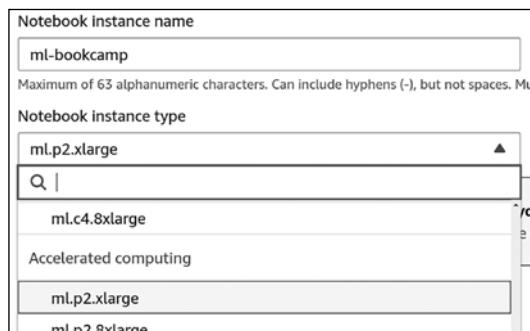
Убедитесь, что находитесь в том же регионе, где вы запросили увеличение лимитов квот.

В SageMaker выберите Notebook Instances, а затем нажмите кнопку Create Notebook (рис. Д.8).



**Рис. Д.8.** Чтобы создать блокнот Jupyter, нажмите Jupyter Notebook

Далее нам нужно настроить экземпляр. Сначала введите имя и тип экземпляра. Поскольку нас интересует экземпляр графического процессора, выберите ml.p2.xlarge в разделе Accelerated Computing (рис. Д.9).

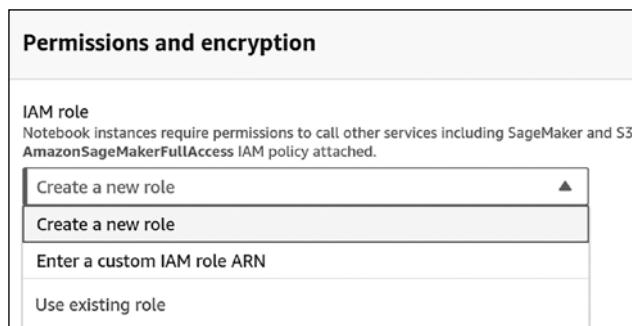


**Рис. Д.9.** Раздел Accelerated Computing содержит экземпляры с графическими процессорами

В Additional Configuration введите 5 GB в поле Volume Size. Благодаря этому нам должно хватить места как для хранения набора данных, так и для сохранения наших моделей.

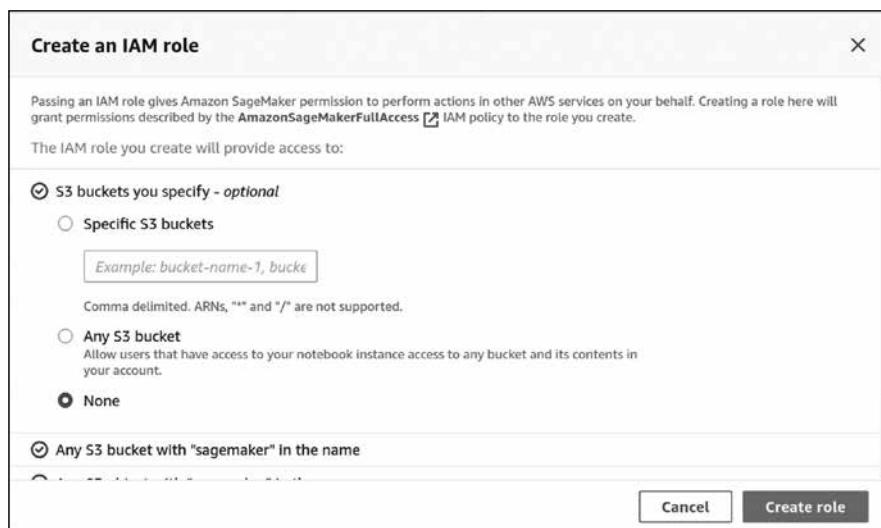
Если вы ранее использовали SageMaker и у вас уже есть для него роль IAM, то выберите ее в разделе IAM Role.

Но если вы делаете это впервые, то выберите Create a New Role (рис. Д.10).



**Рис. Д.10.** Чтобы использовать блокнот SageMaker, нам нужно создать для него роль IAM

При создании роли сохраните значения по умолчанию и нажмите кнопку **Create Role** (рис. Д.11).



**Рис. Д.11.** Значений по умолчанию для новой роли IAM вполне достаточно

Остальные параметры оставьте без изменений:

- Root access — Enable;
- Encryption key — No custom encryption;
- Network — No VPC;
- Git repositories — None.

Наконец нажмите **Create Notebook Instance**, чтобы запустить процесс.

Если по каким-либо причинам вы видите сообщение об ошибке **ResourceLimitExceeded** (рис. Д.12), убедитесь, что:

- вы запросили увеличение лимитов квот для типа экземпляра **ml.p2.xlarge**;
- запрос был обработан;
- вы пытаетесь создать блокнот в том же регионе, где вы запросили увеличение.



**Рис. Д.12.** Если вы видите сообщение об ошибке **ResourceLimitExceeded**, то вам необходимо увеличить лимиты квот

После создания экземпляра блокнот появится в списке экземпляров блокнотов (рис. Д.13).

The screenshot shows the 'Notebook instances' page in the Amazon SageMaker console. At the top, there is a success message: 'Success! Your notebook instance is being created.' followed by instructions to open the instance when it's in service. Below this, the page title is 'Amazon SageMaker > Notebook instances'. The main area is titled 'Notebook instances' and contains a table with the following data:

Name	Instance	Creation time	Status	Actions
ml-bookcamp	ml.p2.xlarge	Oct 08, 2020 05:06 UTC	Pending	-

**Рис. Д.13.** Все получилось! Экземпляр блокнота создан

## 494 Приложение Д. AWS SageMaker

Теперь нам нужно дождаться, пока блокнот сменит статус с Pending на InService; это может занять от одной до двух минут.

Как только он будет в состоянии InService, его можно будет использовать (рис. Д.14). Чтобы получить доступ к нему, нажмите Open Jupyter.

Далее мы узнаем, как использовать его с TensorFlow.

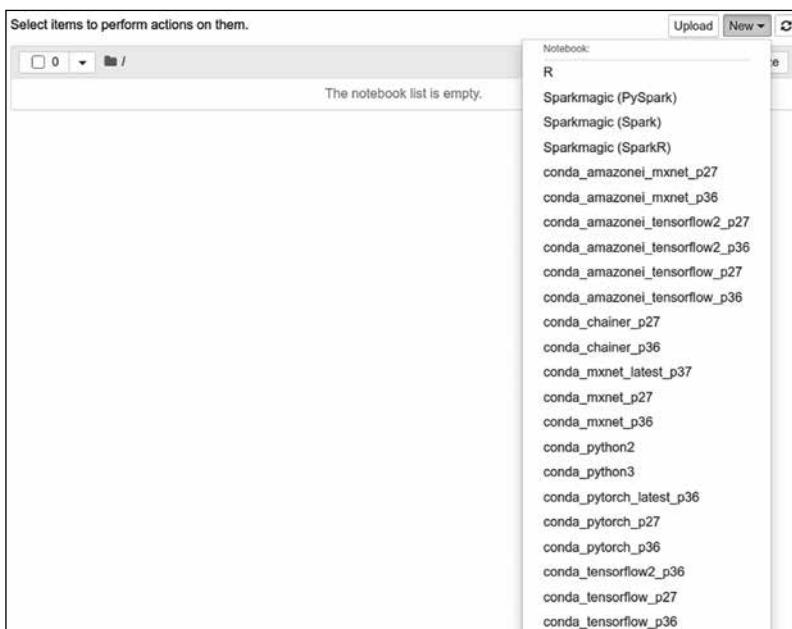
Name	Instance	Creation time	Status	Actions
ml-bookcamp	mLp2.xlarge	Oct 08, 2020 05:06 UTC	InService	Open Jupyter   Open JupyterLab

**Рис. Д.14.** Новый экземпляр блокнота запущен и готов к использованию

### Д.1.3. Обучение модели

После нажатия кнопки Open Jupyter мы видим знакомый интерфейс Jupyter Notebook.

Чтобы создать новый блокнот, нажмите New и выберите conda\_tensorflow2\_p36 (рис. Д.15).



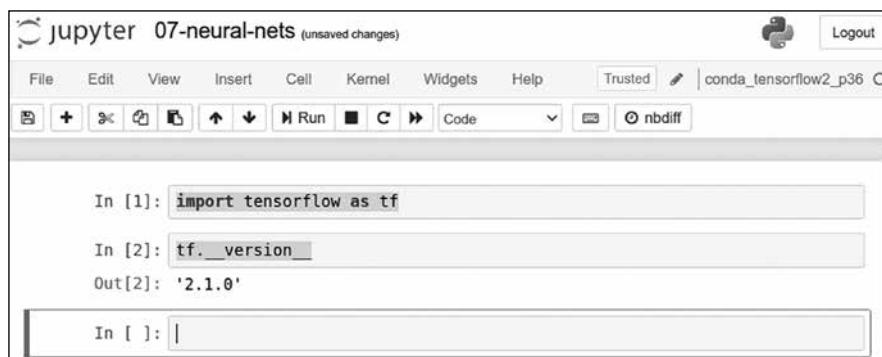
**Рис. Д.15.** Чтобы создать новый блокнот с TensorFlow, выберите conda\_tensorflow2\_p36

Этот блокнот содержит Python версии 3.6 и TensorFlow версии 2.1.0. На момент написания статьи это новейшая версия TensorFlow, доступная в SageMaker.

Теперь импортируйте TensorFlow и проверьте его версию:

```
import tensorflow as tf
tf. version
```

Версия должна быть 2.1.0 или выше (рис. Д.16).



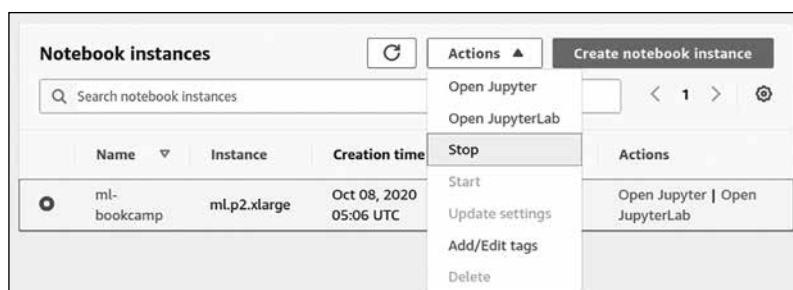
The screenshot shows a Jupyter Notebook interface with the title "jupyter 07-neural-nets (unsaved changes)". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Logout. Below the toolbar is a toolbar with various icons for file operations like Open, Save, Print, and Run. The notebook area contains two cells: In [1]: `import tensorflow as tf`, which has been run; and In [2]: `tf. version`, which has also been run. The output for In [2] is Out[2]: '2.1.0'. A new cell In [ ]: is currently being typed.

**Рис. Д.16.** Для наших примеров нам нужна как минимум версия TensorFlow 2.1.0

Теперь перейдите к главе 7 и обучите нейронную сеть! После завершения обучения нам нужно отключить блокнот.

## Д.1.4. Отключение блокнота

Чтобы остановить блокнот, сначала выберите экземпляр, который вы хотите остановить, а затем выберите действие Stop в раскрывающемся списке Actions (рис. Д.17).



**Рис. Д.17.** Чтобы отключить блокнот, выберите действие Stop

После этого блокнот изменит статус с `InService` на `Stopping`. Может пройти несколько минут, прежде чем он полностью остановится и изменит статус со `Stopping` на `Stopped`.

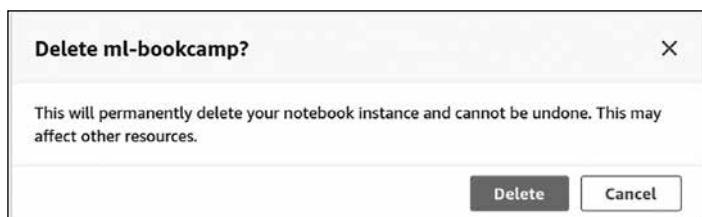
#### **ПРИМЕЧАНИЕ**

Когда мы останавливаем блокнот, весь наш код и данные сохраняются. При следующем запуске мы сможем продолжить с того места, на котором остановились.

#### **ВАЖНО**

Экземпляры блокнотов стоят дорого, поэтому убедитесь, что вы случайно не оставили его включенным. На SageMaker не распространяется бесплатный уровень, поэтому если вы забудете его отключить, то в конце месяца получите огромный счет. В AWS есть способ задать бюджет, чтобы избежать огромных счетов. Документация об управлении затратами в AWS доступна на <https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/budgets-managing-costs.html>. Будьте осторожны и выключайте свой блокнот, если он вам больше не нужен.

Как только вы закончите работу над проектом, можете удалить блокнот. Выберите блокнот, а затем нажмите `Delete` из выпадающего списка (рис. Д.18). Блокнот должен находиться в состоянии `Stopped`, прежде чем вы сможете удалить его.



**Рис. Д.18.** После того как вы закончите главу 7, можете удалить блокнот

Сначала он изменит статус со `Stopped` на `Deleting`, а через 30 секунд исчезнет из списка блокнотов.