

ДЛЯ
ПРОФИ



Машинное обучение
и облачные технологии

Прагматичный

ИИ

НОЙ ГИФТ

Pragmatic AI

An Introduction to Cloud-Based Machine Learning

Noah Gift

◆Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

для
ПРОФЕССИОНАЛОВ

Прагматичный ИИ

Машинное обучение
и облачные технологии

Ной Гифт



Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону
Самара · Минск

2019

ББК 32.813+32.973.23-018
УДК 004.89+004.45
Г51

Гифт Ной

Г51 Прагматичный ИИ. Машинное обучение и облачные технологии. — СПб.: Питер, 2019. — 304 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1061-2

Искусственный интеллект — это мощный инструмент в руках современного архитектора, разработчика и аналитика.

Облачные технологии — ваш путь к укрощению искусственного интеллекта.

Тщательно изучив эту незаменимую книгу от Ноя Гифта, легендарного эксперта по языку Python, вы легко научитесь писать облачные приложения с использованием средств искусственного интеллекта и машинного обучения, решать реалистичные задачи из таких востребованных и актуальных областей, как спортивный маркетинг, управление проектами, ценообразование, сделки с недвижимостью.

Все примеры разобраны на языке Python, № 1 в сфере современных стремительных вычислений.

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

ББК 32.813+32.973.23-018
УДК 004.89+004.45

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

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

ISBN 978-0134863863 англ.
ISBN 978-5-4461-1061-2

© 2019 Pearson Education, Inc.
© Перевод на русский язык ООО Издательство «Питер», 2019
© Издание на русском языке, оформление ООО Издательство «Питер», 2019
© Серия «Для профессионалов», 2019

Краткое содержание

Предисловие.....	14
Благодарности.....	19
Об авторе.....	23

Часть I. Введение в прагматичный ИИ

Глава 1. Что такое ИИ	26
Глава 2. ИИ и инструменты машинного обучения	55
Глава 3. Спартанский жизненный цикл ИИ	82

Часть II. ИИ в облаке

Глава 4. Разработка ИИ в облачной среде с помощью облачной платформы Google	100
Глава 5. Разработка ИИ в облачной среде с помощью веб-сервисов Amazon	119

Часть III. Создание реальных приложений ИИ с нуля

Глава 6. Прогноз популярности в соцсетях в НБА.....	144
Глава 7. Создание интеллектуального бота Slack в AWS.....	188
Глава 8. Извлечение полезной информации об управлении проектами из учетной записи GitHub-организации	206
Глава 9. Динамическая оптимизация виртуальных узлов EC2 в AWS	232
Глава 10. Недвижимость	254
Глава 11. Промышленная эксплуатация ИИ для пользовательского контента....	270

Приложения

Приложение А. Аппаратные ускорители для ИИ.....	298
Приложение Б. Выбор размера кластера	300

Оглавление

Предисловие.....	14
Кому стоит прочитать эту книгу.....	14
Структура издания.....	15
Примеры кода	17
Условные обозначения	18
Благодарности.....	19
Об авторе.....	23

Часть I. Введение в прагматичный ИИ

Глава 1. Что такое ИИ	26
Функциональное введение в Python.....	27
Процедурные операторы	28
Вывод результатов	28
Создание и использование переменных	28
Множественные процедурные операторы.....	29
Сложение чисел	29
Склеивание строк.....	29
Сложные операторы.....	29
Строки и форматирование строк	30
Сложение и вычитание чисел.....	33
Умножение десятичных чисел	33
Использование показательных функций.....	33

Преобразование между различными числовыми типами данных	34
Округление	34
Структуры данных	34
Словари	35
Списки	36
Функции	36
Использование управляющих конструкций	45
Циклы for	45
Циклы while	46
Операторы if/else	47
Промежуточные вопросы	49
Резюме	52
Глава 2. ИИ и инструменты машинного обучения	55
Экосистема исследования данных языка Python: IPython, Pandas, NumPy, блокнот Jupiter, Sklearn	56
Язык R, RStudio, Shiny и ggplot	57
Электронные таблицы: Excel и Google Sheets	58
Разработка облачных приложений ИИ с помощью веб-сервисов Amazon	58
Интеграция разработки и эксплуатации на AWS	59
Непрерывная поставка	59
Создание среды разработки ПО для AWS	60
Настройки проекта Python для AWS	63
Интеграция с блокнотом Jupiter	67
Интеграция утилит командной строки	70
Интеграция AWS CodePipeline	74
Основные настройки Docker для исследования данных	79
Другие сервисы сборки: Jenkins, CircleCI и Travis	80
Резюме	80
Глава 3. Спартанский жизненный цикл ИИ	82
Прагматическая петля обратной связи при промышленной эксплуатации	83
AWS SageMaker	87
Петля обратной связи AWS Glue	89
AWS Batch	93
Петли обратной связи на основе Docker	95
Резюме	97

Часть II. ИИ в облаке

Глава 4. Разработка ИИ в облачной среде с помощью облачной платформы Google	100
Обзор GCP	101
Colaboratory	102
Datalab	105
Расширяем возможности Datalab с помощью Docker и реестра контейнеров Google.....	105
Запуск полнофункциональных машин с помощью Datalab	106
BigQuery	108
Облачные сервисы ИИ компании Google	111
Тензорные процессоры Google и TensorFlow	115
Резюме	118
Глава 5. Разработка ИИ в облачной среде с помощью веб-сервисов Amazon	119
Создание решений дополненной и виртуальной реальности на основе AWS.....	122
Компьютерное зрение: создание конвейеров AR/VR с помощью EFS и Flask.....	122
Создание конвейера инженерии данных с помощью EFS, Flask и Pandas.....	125
Резюме	141

Часть III. Создание реальных приложений ИИ с нуля

Глава 6. Прогноз популярности в соцсетях в НБА.....	144
Постановка задачи	145
Сбор данных	145
Получение данных из труднодоступных источников	168
Получение данных о просмотрах страниц «Википедии» спортсменов	168
Получение данных о вовлеченности в Twitter спортсменов.....	173
Изучаем данные об игроках НБА	176
Машинное обучение без учителя для данных об игроках НБА	181
Построение фасетного графика по игрокам НБА на языке R.....	182
Собираем все воедино: команды, игроков, социальный авторитет и рекламные отчисления	183
Дальнейшие прагматичные шаги и учебные материалы	187
Резюме	187

Глава 7. Создание интеллектуального бота Slack в AWS	188
Создание бота	188
Преобразование библиотеки в утилиту командной строки	189
Выводим бот на новый уровень с помощью сервиса AWS Step Functions	191
Настройка учетных данных IAM	193
Завершение создания пошаговой функции	203
Резюме	205
Глава 8. Извлечение полезной информации об управлении проектами из учетной записи GitHub-организации	206
Обзор проблем, возникающих при управлении программными проектами	206
Создание исходного каркаса проекта исследования данных	209
Сбор и преобразование данных	211
Обработка GitHub-организации в целом.....	213
Формирование предметно-ориентированной статистики.....	214
Подключение проекта по исследованию данных к интерфейсу командной строки.....	216
Исследование GitHub-организаций с помощью блокнота Jupiter	218
Изучаем метаданные файлов проекта CPython	221
Изучаем файлы, удаленные из проекта CPython	225
Развертывание проекта в каталоге пакетов Python.....	228
Резюме	231
Глава 9. Динамическая оптимизация виртуальных узлов EC2 в AWS	232
Выполнение заданий на платформе AWS.....	232
Спотовые виртуальные узлы	232
Теория спотовых виртуальных узлов и история цен на них	233
Создание утилиты и блокнота для сравнения цен на спотовые виртуальные узлы на основе машинного обучения.....	236
Написание модуля запуска спотового виртуального узла	243
Написание более сложного модуля запуска для спотового виртуального узла	250
Резюме	252
Глава 10. Недвижимость	254
Исследование цен на недвижимость в США	254
Интерактивная визуализация данных в Python	257
Кластеризация по порядку размера и цене	260
Резюме	269

Глава 11. Промышленная эксплуатация ИИ для пользовательского контента....	270
Получившее премию Netflix решение не было внедрено в промышленную эксплуатацию	271
Ключевые понятия рекомендательных систем	272
Использование фреймворка Surprise в языке Python	273
Облачные решения для создания рекомендательных систем	276
Проблемы, возникающие на практике при работе с рекомендациями	277
Облачный NLP и анализ тональности высказываний	282
NLP на платформе Azure.....	283
NLP на платформе GCP	286
Изучаем API сущностей	287
Бессерверный конвейер ИИ промышленного уровня для NLP на платформе AWS	290
Резюме	295

Приложения

Приложение А. Аппаратные ускорители для ИИ.....	298
Приложение Б. Выбор размера кластера	300

ОТЗЫВЫ

Фантастическое дополнение к списку литературы, предназначенной для фанатов новых технологий! Столько всего можно сказать про эту книгу! Ной Гифт создал действительно практическое руководство для всех, кто имеет дело с коммерческим применением машинного обучения. Она не только рассказывает, как использовать машинное обучение для больших наборов данных, но и дает читателю представление о внутренних взаимосвязях данной технологии. Книга позволит множеству команд разработчиков и исследователей данных с первых дней наладить эффективную разработку и поддержку самых сложных проектов.

*Нивас Дурайрадж (Nivas Durairaj),
специалист по технической поддержке, AWS
(сертифицированный архитектор решений AWS уровня Professional)*

Прекрасная книга для тех, кто хотел бы заглянуть глубже в технологии создания конвейеров и утилит машинного обучения промышленного уровня, которые действительно помогли бы командам ваших исследователей данных, инженеров-аналитиков и специалистов по интеграции разработки и эксплуатации в сфере обработки данных. Даже опытные разработчики нередко впустую тратят время и силы на определенные задачи. Часто книги по разработке ПО и университетские курсы даже не упоминают о важных для перехода к промышленной эксплуатации шагах. У Ноя есть настоящий дар находить прагматичные подходы к развертыванию программного обеспечения, существенно ускоряющие процесс разработки и поставки ПО. Благодаря его сосредоточенности и энтузиазму становится возможным быстрое создание поистине уникальных программных решений.

Ключ к созданию пригодных для промышленной эксплуатации конвейеров машинного обучения — автоматизация. Чтобы создать подходящую для промышленной эксплуатации систему, необходимо автоматизировать и масштабировать задачи и шаги, выполняемые инженерами вручную во время фазы исследования или создания

прототипа. В этой книге можно найти множество полезных и интересных упражнений, которые помогут разработчикам на языке Python автоматизировать свои конвейеры и перенести их в облако.

В настоящее время я работаю с большими данными, конвейерами машинного обучения, Python, AWS, облачными сервисами Google и Azure в онлайн-риелторском агентстве Roofstock.com. Размер нашей базы данных, предназначенной для аналитической обработки, приближается к 500 млн строк. В этой книге я нашел для себя множество практических советов и упражнений, значительно повысивших мою личную производительность. Рекомендую!

*Майкл Вирлин (Michael Vierling),
ведущий инженер, компания Roofstock*

*Эта книга посвящается моим родным и близким, которые
всегда рядом в трудную минуту: моей жене Леа Гифт; моей
маме Шари Гифт; моему сыну Лиаму Гифту и моему учителю
доктору Джозефу Богену*

Предисловие

Около 20 лет назад я работал в компании Caltech в Пасадене и мечтал, что когда-нибудь буду ежедневно работать с ИИ (искусственным интеллектом). Тогда, в начале 2000-х, для мечтаний об ИИ было не самое подходящее время. Несмотря на это, я достиг желаемого, и эта книга подводит итоги увлечения всей моей жизни — ИИ и научной фантастикой. Мне очень повезло работать в Caltech бок о бок с крупнейшими специалистами по ИИ, и, вне всякого сомнения, этот опыт сыграл свою роль при написании данной книги.

Помимо ИИ, меня всегда неудержимо притягивали автоматизация и практичность. Книга, которую вы держите в руках, охватывает и эти темы тоже. Как закаленный, опытный менеджер, постоянно поставляющий программные продукты и выживающий среди ужасных, чудовищных, никудышных технологий, я стал очень практичным человеком. То, что не развернуто в промышленной эксплуатации, — не в счет. Не автоматизировано — значит не работает. Надеюсь, что эта книга вдохновит вас разделить мою точку зрения.

Кому стоит прочитать эту книгу

Эта книга предназначена для всех, кого интересуют ИИ, машинное обучение, облачные вычисления, а также любое сочетание данных тем. Как программисты, так и не-программисты найдут тут для себя крупинки полезной информации. Многие студенты, с которыми я пересекался на семинарах

в NASA, PayPal и в Калифорнийском университете в Дэвисе, смогли усвоить эти мои идеи даже при очень ограниченном опыте программирования или вообще без такового. В данной книге активно используется Python — один из лучших языков для новичков в программировании.

В то же время здесь рассматривается множество таких продвинутых тем, как использование платформ облачных вычислений (например, AWS, GCP и Azure), а также программирование машинного обучения и ИИ. Продвинутые технологи, свободно владеющие Python, облачными вычислениями и машинным обучением, тоже найдут для себя много полезных идей, которые смогут сразу применить в своей текущей работе.

Структура издания

Данная книга разбита на три части: часть I «Введение в прагматичный ИИ»; часть II «ИИ в облаке» и часть III «Создание реальных приложений ИИ с нуля». В части I главы 1–3 являются вводными.

- Глава 1 «Что такое ИИ» включает обзор целей книги и очень краткое руководство по Python, содержащее ровно столько информации, сколько нужно пользователю для понимания кода Python в этой книге.
- Глава 2 «ИИ и инструменты машинного обучения» охватывает жизненный цикл систем сборки, использование командной строки и блокнотов Jupyter в рамках проекта по исследованию данных.
- Глава 3 «Спартанский жизненный цикл ИИ» показывает петлю обратной связи при прагматической эксплуатации проектов. Здесь описываются такие утилиты и фреймворки, как Docker, SageMaker из AWS, а также тензорные процессоры, предназначенные для использования с библиотекой TensorFlow (TensorFlow, Processing Units, TPU).

В части II главы 4 и 5 рассматривают AWS и облачную инфраструктуру Google.

- Глава 4 «Разработка ИИ в облачной среде с помощью облачной платформы Google» описывает облачную платформу Google (Google Cloud Platform, GCP) и некоторые из предлагаемых ею уникальных, удобных для разработчика возможностей. В ней охвачены такие сервисы, как TPU, Colaboratory и Datalab.

- Глава 5 «Разработка ИИ в облачной среде с помощью веб-сервисов Amazon» углубляется в вопросы последовательности выполняемых при работе с AWS действий, например использования спотовых виртуальных узлов, CodePipeline, Boto и тестирования программ на его основе, а также включает высокоуровневый обзор сервисов.

В части III главы 6–11 охватывают прикладные приложения ИИ с примерами.

- Глава 6 «Прогноз популярности в соцсетях в НБА» основана на моей работе в стартапе, нескольких статьях и докладе на конференции Strata. Охвачены такие темы, как «Что определяет стоимость команды?», «Растет ли число фанатов на играх в результате побед команды?», «Коррелирует ли зарплата игрока с его деятельностью в соцсетях?».
- Глава 7 «Создание интеллектуального бота Slack в AWS» рассказывает о создании бессерверного чат-бота для скрапинга веб-сайтов и отправки информации обратно на Slack.
- Глава 8 «Извлечение полезной информации об управлении проектами из учетной записи GitHub-организации» исследует распространенный источник поведенческих данных — метаданные GitHub. Для поиска бесценных поведенческих данных мы воспользуемся библиотекой Pandas, блокнотами Jupyter и утилитой командной строки click.
- Глава 9 «Динамическая оптимизация виртуальных узлов EC2 в AWS» демонстрирует возможности выбора оптимальной цены с помощью приемов машинного обучения на основе сигналов AWS.
- Глава 10 «Недвижимость» посвящена изучению цен на дома — местных и в масштабах всей страны — с помощью машинного обучения и интерактивных графиков.
- Глава 11 «Промышленная эксплуатация ИИ для пользовательского контента» рассказывает о применении ИИ для работы с пользовательским контентом. Рассматриваются такие вопросы, как анализ тональности высказываний и рекомендательные системы.

Приложение А «Аппаратные ускорители для ИИ» описывает аппаратные ускорители, специально рассчитанные на работу с ИИ. В качестве примера аппаратного ускорителя для ИИ можно привести TPU от компании Google.

Приложение Б «Выбор размера кластера» рассматривает этот процесс скорее как искусство, а не как точную науку, несмотря на наличие некоторых методик, делающих его более прозрачным.

Примеры кода

Каждую главу этой книги сопровождает блокнот или набор блокнотов Jupiter. Они были разработаны на протяжении нескольких последних лет и представляют собой результат написания статей, а также проведения семинаров и курсов.



Все исходные коды книги можно найти в виде блокнотов Jupiter здесь: <https://github.com/noahgift/pragmaticai>.

Многие примеры также включают сборочные файлы, подобные следующему:

setup:

```
python3 -m venv ~/.pragai
```

install:

```
pip install -r requirements.txt
```

test:

```
cd chapter7; py.test --nbval-lax notebooks/*.ipynb
```

lint:

```
pylint --disable=W,R,C *.py
```

lint-warnings:

```
pylint --disable=R,C *.py
```

Сборочные файлы — прекрасный способ организации проекта по исследованию данных на языке Python или R. Особенно они удобны для настройки среды, линтинга исходного кода, выполнения тестов и развертывания кода. Кроме того, устранить целый класс проблем позволяют изолированные среды, такие как `virtualenv`. Поразительно, сколько моих студентов сталкивались с одной и той же проблемой: установили какой-то пакет для одного интерпретатора Python, а использовали другой интерпретатор. Или не могли заставить программу работать из-за конфликта двух пакетов.

В целом решение этой проблемы — применять для каждого проекта виртуальную среду и всегда выбирать именно ее при работе с соответствующим проектом. Использование при планировании проекта лишь малой толики времени значительно предотвращает будущие проблемы. Применение сборочных файлов, линтинг, тесты в блокнотах Jupiter, система сборки SaaS — рекомендуемые практики, заслуживающие всяческого одобрения и распространения.

Условные обозначения

В этой книге используются следующие типографские обозначения.

Участки кода внутри текста, названия объектов, классов, методов, имена папок и файлов, расширения файлов, пути и пользовательский ввод выглядят так: «Теперь можно преобразовать возвращаемые оценки в объект `DataFrame` библиотеки `Pandas` для разведочного анализа данных».

Код, начинающийся со строк вида `In [2]`, обозначает выводимые в консоль результаты работы программы на языке Python. Зачастую схоже с сопровождающими книгу примерами в блокнотах `Jupyter`, доступными на `GitHub`.

Блок кода выглядит так:

```
#Переменные среды приложения
REGION = "us-east-1"
APP = "nlp-api"
NLP_TABLE = "nlp-table"
```

Веб-адреса выделены таким шрифтом: <https://github.com>.

Новые термины и ключевые слова выделяются курсивом.

Благодарности

Я благодарен Лоре Левин (Laura Lewin), предоставившей мне возможность опубликовать эту книгу в издательстве Pearson, как и всем остальным членам команды издательства: Малобике Чакраборти (Malobika Chakraborty) и Шери Реплин (Sheri Replin). Также я хотел бы поблагодарить технических рецензентов: Чао-Сюаня Шена (Chao-Hsuan Shen) (<https://www.linkedin.com/in/lucashsuan/>), Кеннеди Бермана (Kennedy Behrman) (<https://www.linkedin.com/in/kennedybehrman/>) и Гая Эрнеста (Guy Ernest) (<https://www.linkedin.com/in/guyernest/>) из компании Amazon. Они сыграли важную роль в доведении этой книги до ума.

В целом я хотел бы поблагодарить Caltech. Именно во время работы там, в далеко не лучшие для этой технологии времена, я заинтересовался ИИ. Я помню, как один профессор из Caltech говорил мне, что ИИ — пустая трата времени, но я все равно хотел этим заниматься. Я поставил себе целью к 40-летию начать писать серьезные ИИ-программы и свободно владеть несколькими языками программирования — и достиг этого. Меня всегда очень мотивировало, когда мне запрещали делать что-то, так что спасибо, профессор!

Я также встретил там несколько весьма влиятельных людей, включая доктора Джозефа Богена (Joseph Bogen), нейрофизиолога и эксперта в области теорий сознания. Мы обсуждали за обедами нейронные сети, истоки сознания, математический анализ и работу доктора в лаборатории Кристофа

Коха (Christof Koch). Слишком мало сказать, что он повлиял на мою жизнь, и я горжусь тем, что сегодня работаю над нейронными сетями, в частности, благодаря этим беседам за обедом 18 лет назад.

Я повстречал в Caltech и других людей, с которыми не общался так тесно, но которые все же повлияли на меня: доктора Дэвида Балтимора (David Baltimore), на которого я работал, доктора Дэвида Гудстайна (David Goodstein), на которого я тоже работал, доктора Фэй-Фэй Ли (Fei-Fei Li) и доктора Тайтуса Брауна (Titus Brown), который «подсадил» меня на Python.

Спорт всегда играл большую роль в моей жизни. Какое-то время я всерьез занимался баскетболом, трекинговыми гонками и даже алтимат-фрисби. В Калифорнийском политехническом в Сан-Луис-Обиспо я познакомился с Шелдоном Блокбаргером (Sheldon Blockburger), олимпийским десятиборцем, который учил меня бегать дистанцию 8 по 200 м менее чем за 27 с и дистанцию 300 м до тех пор, пока меня не начинало тошнить. Я до сих пор помню, как он говорил: «Менее процента населения достаточно дисциплинированы для этого упражнения». Такая тренировка внутренней дисциплины сыграла колоссальную роль в моем становлении как разработчика ПО.

Я всегда крутился возле спортсменов мирового класса, меня притягивали в них энергетика, оптимизм и желание побеждать. В последние несколько лет я, тренируясь в Empower в Сан-Франциско, Калифорния, открыл для себя бразильское джиу-джитсу (Brazilian jiu-jitsu, BJJ). В значительной мере благодаря тамошним тренерам — Тареку Азиму (Tareq Azim), Йосефу Азиму (Yossef Azim) и Джошу Макдональду (Josh McDonald) — я стал мыслить как боец. В частности, меня вдохновлял Джош Макдональд, спортсмен мирового класса в нескольких видах спорта, его тренировки сыграли колоссальную роль в том, что я дописал данную книгу. Я очень рекомендую этот спортзал и этих тренеров всем техническим специалистам в Области залива Сан-Франциско.

Перечисленные первоначальные контакты привели меня к знакомству с множеством других людей из мира боевых искусств, например из школы боевых искусств NorCal Fight Alliance (Санта-Роза), управляемой Дэйвом Терреллом (Dave Terrell). Я тренировался с Дэйвом Терреллом, Джейкобом Хардгроувом (Jacob Hardgrove), Коллином Хартом (Collin Hart), Джастином Соммером (Justin Sommer) и другими, и все они бескорыстно делились

своими знаниями и размышлениями о боевых искусствах. Ничего в моей жизни пока что не может сравниться с тем, как 110-килограммовый обладатель черного пояса снова и снова выжимал из меня все соки, день за днем. Знание, что я способен выдержать это, сильно помогло мне перенести все тяготы написания данной книги. Мне очень повезло тренироваться в этом спортзале, и я рекомендую его всем техническим специалистам в Области залива.

И последняя благодарность из сферы ВJJ — Академии джиу-джитсу Мауи в городе Хаику на острове Мауи (на Гавайях), вдохновлявшей меня при написании этой книги. Я взял паузу в прошлом году, чтобы решить для себя, чем мне хотелось бы заниматься дальше, и тренировался с профессором Луисом Эредиа (Luis Heredia) и Джоэлем Буэ (Joel Bouhey). Они оба великолепные учителя, и этот опыт очень помог мне при написании данной книги.

Спасибо Заку Стоуну (Zak Stone), менеджеру по продуктам TensorFlow, за возможность раннего доступа к TPU и помощь с облачной платформой Google. Спасибо также Гаю Эрнесту (Guy Ernest) из компании Amazon за советы по сервисам AWS. Спасибо Полу Шили (Paul Shealy) из Microsoft за советы по облачным решениям Azure.

Еще одна организация, которую я хотел бы поблагодарить, — Калифорнийский университет в Дэвисе. Я получил диплом магистра в области бизнес-аналитики, и это оказало большое влияние на мою жизнь. Мне повезло встретить таких замечательных профессоров, как доктор Дэвид Вудрафф (David Woodruff), который разрешил мне программировать на Python для курса оптимизации и помог с обладающей большими возможностями библиотекой Puomo. Я должен также поблагодарить профессора Диксона Луи (Dickson Louie) — замечательного наставника и доктора Хемана Бхаргаву (Nemant Bhargava), предоставившего мне возможность преподавать машинное обучение в Калифорнийском университете в Дэвисе. Доктор Норм Мэтлофф (Norm Matloff) также бескорыстно помогал мне совершенствоваться в машинном обучении и статистике, за что я ему очень благодарен. Я также благодарен студентам моего курса BAX-452, опробовавшим часть материалов данной книги, и персоналу MSBA¹: Эми Расселл (Amy Russell) и Шачи Говил (Shachi Govil). Хочу сказать спасибо

¹ Магистерская программа в области естественных наук по бизнес-аналитике (Master of Science in Business Analytics). — *Здесь и далее примеч. пер.*

и еще одному другу — Марио Искьердо (Mario Izquierdo, <https://github.com/marioizquierdo>) — великолепному разработчику, разбрасывавшемуся прекрасными идеями по поводу развертывания систем на практике.

И наконец, я хотел бы поблагодарить нескольких друзей из эпохи моей работы в стартапе: Джерри Кастро (Jerry Castro, <https://www.linkedin.com/in/jerry-castro-4bbb631/>) и Кеннеди Бермана (<https://www.linkedin.com/in/kennedybehman/>) — удивительно надежных, трудолюбивых и безотказных. Значительная часть материалов для этой книги была создана, когда мы работали бок о бок в окопах стартапа, который в итоге потерпел крах. Именно подобный опыт раскрывает подлинный характер человека, и для меня было честью работать вместе с ними и быть их другом.

Об авторе

Ной Гифт (Noah Gift) — преподаватель и консультант в магистратуре Калифорнийского университета в Дэвисе в программе MSBA. Преполагает выпускникам машинное обучение и проводит консультации по машинному обучению и облачным архитектурам для студентов и преподавательского состава. Ему принадлежит около 100 технических публикаций, в том числе две книги, по различным темам: от облачного машинного обучения до интеграции разработки и эксплуатации (DevOps). Он — сертифицированный архитектор решений AWS и сертифицированный специалист в области машинного обучения AWS, помогавший в свое время создавать систему сертификации AWS по машинному обучению. У него также есть диплом магистра в области бизнес-аналитики Калифорнийского университета в Дэвисе, а также диплом магистра естественных наук в области компьютерных информационных систем от Калифорнийского университета в Лос-Анджелесе и диплом бакалавра естественных наук в области диетологии от Калифорнийского политехнического университета, Сан-Луис-Обиспо.

В профессиональном отношении у Ноя почти 20 лет опыта программирования на Python, он является коллегиальным членом Python Software Foundation. Он работал, в частности, на должностях технического директора, генерального директора, технического директора-консультанта и архитектора облачных решений. Свой опыт он получил в множестве разнообразных компаний, включая ABC, Caltech, Sony Imageworks, Disney

Feature Animation, Weta Digital, AT&T, Turner Studios и Linden Lab. В последние десять лет он отвечал за вывод на глобальный рынок множества новых продуктов в различных компаниях, принесших миллионы долларов прибыли. В настоящее время он консультирует в стартапах и других компаниях по вопросам машинного обучения, облачной архитектуры и в качестве основателя компании Pragmatic AI Labs дает консультации уровня технического директора.

Узнать больше о Ное вы можете, подписавшись на него на GitHub по адресу <https://github.com/noahgift/>, посетив сайт его компании Pragmatic AI Labs <https://pauml.com>, его личный сайт <http://noahgift.com> или связавшись с ним на LinkedIn <https://www.linkedin.com/in/noahgift/>.

Часть I

Введение

в прагматичный ИИ

1

Что такое ИИ

Не путайте процесс с результатом.

Джон Вуден
(John Wooden)

Если вы взяли в руки данную книгу, то, вероятно, вас интересует прагматичный ИИ. Книг, курсов и вебинаров по современным методам машинного и глубокого обучения вполне достаточно. Не хватает лишь информации о том, как довести проект до состояния, при котором появляется принципиальная возможность использования всех продвинутых методов. Именно для этого предназначена книга — чтобы сократить разрыв между теорией и практической реализацией связанных с искусственным интеллектом проектов.

Во многих случаях обучить свою модель невозможно по причине простой нехватки времени, ресурсов и навыков для реализации. Всегда есть лучшее решение, чем двигаться по заведомо провальному пути. А практикующие прагматичный ИИ всегда используют оптимальную для ситуации методику. В некоторых случаях это может означать вызов программного интерфейса приложения (application programming interface, API) с предварительно обученным методом. Еще одна прагматичная методика ИИ состоит в создании намеренно менее эффективной модели для упрощения понимания и развертывания в промышленной эксплуатации.

В 2009 г. Netflix объявил знаменитый конкурс, предложив приз \$1 млн команде разработчиков, которая сможет повысить точность рекомендаций компании на 10 %. Конкурс был очень увлекательным и опередил свое время в отношении науки о данных. Не многие знают, однако, что победивший алгоритм так и не был реализован вследствие потенциально больших затрат на разработку

и внедрение (<https://www.wired.com/2012/04/netflix-prize-costs/>). Вместо него были использованы некоторые алгоритмы команды, достигшей улучшения результатов «лишь» на 8,43 %. Это прекрасное подтверждение того, что подлинная цель многих задач ИИ — практичность, а вовсе не идеальный результат.

Данная книга уделяет основное внимание тому, как оказаться той самой командой со второго места, чье решение на самом деле попадет в промышленную эксплуатацию. Это связующая нить всей книги, чья цель — перевод кода в промышленную эксплуатацию, а не получение им формального титула «лучшее решение», которое никогда не станет программным продуктом.

Функциональное введение в Python

Python — потрясающий язык программирования, способный на множество различных вещей. Есть мнение, что Python ни в чем не демонстрирует исключительных результатов, но достаточно хорош в абсолютном большинстве приложений. Подлинная сильная сторона этого языка — преднамеренное отсутствие сложности. Python допускает также программирование в множестве разных стилей. Его вполне можно использовать для выполнения операторов в процедурном стиле, построчно. Но можно также и в качестве сложного объектно-ориентированного языка программирования, обладающего такими продвинутыми возможностями, как метаклассы и множественное наследование.

При изучении языка Python, особенно в контексте науки о данных, на некоторых его частях можно не заострять внимания. Можно даже сказать, что многие его составные части, особенно некоторые объектно-ориентированные возможности, никогда не используются при написании блокнотов Jupiter. Вместо этого имеет смысл использовать альтернативный подход с упором на функции. Данный раздел вкратце познакомит вас с языком Python, который можно рассматривать как новый Microsoft Excel.

Один из недавних моих аспирантов сказал мне, что до моего курса машинного обучения его беспокоил сложный вид соответствующего кода. Но после нескольких месяцев работы с языком Python и блокнотами Jupiter он почувствовал уверенность при использовании Python для решения задач науки о данных. Я убежден, исходя из виденного мной во время преподавания, что любой пользователь Excel может научиться использовать блокноты Jupiter на Python.

Стоит отметить, что в развертывании кода в промышленной эксплуатации Jupyter может играть роль механизма доставки, а может и не играть. В последнее время стали очень популярны новые фреймворки, вроде Databricks, SageMaker и Datalab, создающие возможности введения в промышленную эксплуатацию с помощью Jupyter, но чаще всего Jupyter используется для выполнения различных экспериментов.

Процедурные операторы

Для работы со следующими примерами у вас должен быть установлен Python версии не ниже 3.6. Скачать последнюю версию Python можно по адресу <https://www.python.org/downloads/>. Процедурные операторы фактически представляют собой операторы, допускающие построчное выполнение. Ниже перечислены типы выполняемых процедурных операторов.

- Блокнот Jupyter.
- Командная оболочка IPython.
- Интерпретатор Python.
- Сценарии Python.

Вывод результатов

Вывод результатов в Python очень прост. Функция `print` получает входные данные и выводит их в консоль:

```
In [1]: print("Hello world")
...:
Hello world
```

Создание и использование переменных

Переменные создаются путем присваивания. В следующем примере переменной присваивается значение, после чего она выводится в консоль посредством объединения двух операторов через точку с запятой. Стиль с подобным использованием точек с запятой можно часто встретить в блокнотах Jupyter, но в коде и библиотеках, предназначенных для промышленной эксплуатации, на него обычно смотрят с неодобрением.

```
In [2]: variable = "armbar"; print(variable)
armbar
```

Множественные процедурные операторы

Полное решение задачи может представлять собой обычный процедурный код, показанный ниже. Такой стиль уместен в блокнотах Jupyter, но редко встречается в коде, предназначенном для промышленной эксплуатации.

```
In [3]: attack_one = "kimura"
...: attack_two = "arm triangle"
...: print("In Brazilian jiu-jitsu a common attack is a:", attack_one)
...: print("Another common attack is a:", attack_two)
...:
In Brazilian jiu-jitsu a common attack is a: kimura
Another common attack is a: arm triangle
```

Сложение чисел

Python можно использовать и в качестве калькулятора. Прекрасный способ привыкнуть к языку — начать использовать его вместо Microsoft Excel или приложения-калькулятора.

```
In [4]: 1+1
...:
Out[4]: 2
```

Склеивание строк

Можно складывать и строки:

```
In [6]: "arm" + "bar"
...:
Out[6]: 'armbar'
```

Сложные операторы

Можно создавать и более сложные операторы — с использованием структур данных вроде переменной `belt`, представляющей собой список:

```
In [7]: belts = ["white", "blue", "purple", "brown", "black"]
...: for belt in belts:
...:     if "black" in belt:
...:         print("The belt I want to earn is:", belt)
...:     else:
...:         print("This is not the belt I want to end up with:", belt)
```

```
...:
This is not the belt I want to end up with: white
This is not the belt I want to end up with: blue
This is not the belt I want to end up with: purple
This is not the belt I want to end up with: brown
The belt I want to earn is: black
```

Строки и форматирование строк

Строки (strings) — последовательности символов. Они часто форматируются программно. Практически все программы на языке Python применяют строки для отправки сообщений пользователям. Однако важно хорошо представлять себе несколько базовых понятий.

- ❑ Строки можно создавать с помощью одинарных, двойных и тройных/двойных кавычек.
- ❑ Строки можно форматировать.
- ❑ Строки могут находиться в нескольких кодировках, включая Unicode.
- ❑ Существует множество методов работы со строками. В редакторе или командной оболочке IPython можно увидеть список этих методов, выделенных с помощью автозаполнения табуляцией.

```
In [8]: basic_string = ""
```

```
In [9]: basic_string.
        capitalize()  encode()          format()
isalpha()  islower()   istitle()        lower()
           casefold()  endswith()       format_map()
isdecimal() isnumeric()  isupper()        lstrip()
           center()   expandtabs()     index()
isdigit()  isprintable() join()            maketrans()  >
           count()   find()          isalnum()
isidentifier() isspace()    ljust()         partition()
```

Простейший тип строки

Простейший тип строки — переменная, которой присвоено значение в виде фразы в кавычках. Кавычки могут быть тройными, двойными или одинарными.

```
In [10]: basic_string = "Brazilian jiu-jitsu"
```

Разбиение строк

Строку можно превратить в список, разбив ее по пробелам или другим символам:

```
In [11]: #Разбиение по пробелам (по умолчанию)
...: basic_string.split()
Out[11]: ['Brazilian', 'jiu-jitsu']
```

```
In [12]: #Разбиение по дефисам
...: string_with_hyphen = "Brazilian-jiu-jitsu"
...: string_with_hyphen.split("-")
...:
Out[12]: ['Brazilian', 'jiu-jitsu']
```

Все заглавные буквы

В Python есть множество удобных встроенных функций для преобразования строк. Вот так, например, можно преобразовать все символы в строке в верхний регистр:

```
In [13]: basic_string.upper()
Out[13]: 'BRAZILIAN JIU JITSU'
```

Срезы строк

Строки можно разрезать на части, указывая на длину:

```
In [14]: #Получить первые два символа строки
...: basic_string[:2]
Out[14]: 'Br'
In [15]: #Получить длину строки
...: len(basic_string)
Out[15]: 19
```

Склеивание строк

Строки можно склеивать путем конкатенации или присваивания строки переменной с последующим прибавлением к ней строк. Такой стиль приемлем и интуитивно понятен для блокнотов Jupiter, но из соображений производительности лучше использовать в коде для промышленной эксплуатации f-строки.

```
In [16]: basic_string + " is my favorite Martial Art"
Out[16]: 'Brazilian jiu-jitsu is my favorite Martial Art'
```

Сложное форматирование строк

Один из лучших способов форматирования строк в современном Python 3 — использование f-строк:

```
¶ In [17]: f'I love practicing my favorite Martial Art,
           {basic_string}'
...:
Out[17]: 'I love practicing my favorite Martial Art,
         Brazilian jiu-jitsu'
```

Для обертывания строк можно использовать тройные кавычки

Иногда может понадобиться взять фрагмент текста и присвоить его переменной. Проще всего сделать это в Python с помощью заключения фразы в тройные кавычки:

```
In [18]: f"""
...: This phrase is multiple sentences long.
...: The phrase can be formatted like simpler sentences,
...: for example, I can still talk about my favorite
...: Martial Art {basic_string}
...: """
Out[18]: '\nThis phrase is multiple sentences long.\nThe phrase
can be formatted like simpler sentences,\nfor example,
I can still talk about
my favorite Martial Art Brazilian jiu-jitsu\n'
```

Разрывы строк можно удалить с помощью метода Replace

Вышеприведенная длинная строка содержала символы разрывов строки — символы `\n`, которые можно удалить с помощью метода `replace`:

```
In [19]: f"""
...: This phrase is multiple sentences long.
...: The phrase can be formatted like simpler sentences,
...: for example, I can still talk about my favorite
...: Martial Art {basic_string}
...: """.replace("\n", "")
Out[19]: 'This phrase is multiple sentences long. The phrase can be
formatted like simpler sentences, for example, I can still talk about
my favorite Martial Art Brazilian jiu-jitsu'
```

Числа и арифметические операции

В Python есть встроенный калькулятор. Он способен выполнять множество простых и сложных арифметических операций без всяких дополнительных библиотек.

Сложение и вычитание чисел

Язык Python также демонстрирует гибкость в форматировании фраз на основе f-строк:

```
In [20]: steps = (1+1)-1
...: print(f"Two Steps Forward: One Step Back = {steps}")
...:
Two Steps Forward: One Step Back = 1
```

Умножение десятичных чисел

В язык Python также встроена поддержка десятичных чисел, что упрощает решение арифметических задач:

```
In [21]:
...: body_fat_percentage = 0.10
...: weight = 200
...: fat_total = body_fat_percentage * weight
...: print(f"I weight 200lbs, and {fat_total}lbs of that is fat")
...:
I weight 200lbs, and 20.0lbs of that is fat
```

Использование показательных функций

С помощью библиотеки `math` можно легко вычислить, например, 2 в третьей степени:

```
In [22]: import math
...: math.pow(2,3)
```

```
Out[22]: 8.0
```

Другой способ:

```
>>> 2**3
8
```

Преобразование между различными числовыми типами данных

В Python существует множество форм чисел. Вот две наиболее распространенные:

- ❑ целые числа;
- ❑ числа с плавающей запятой.

```
In [23]: number = 100
...: num_type = type(number).__name__
...: print(f"{number} is type [{num_type}]")
...:
100 is type [int]
In [24]: number = float(100)
...: num_type = type(number).__name__
...: print(f"{number} is type [{num_type}]")
...:
100.0 is type [float]
```

Округление

Десятичное число с большим количеством знаков можно округлить, например, до двух знаков после запятой:

```
In [26]: too_many_decimals = 1.912345897
...: round(too_many_decimals, 2)
Out[26]: 1.91
```

Структуры данных

В Python есть несколько часто используемых базовых структур данных:

- ❑ списки;
- ❑ словари.

Словари и списки — основные «рабочие лошадки» языка Python, но в нем есть и другие структуры данных, например кортежи, счетчики и т. д., также заслуживающие изучения.

Словари

Словари, как и сам Python, отлично подходят для решения множества задач. В следующем примере в словарь помещается список атакующих приемов бразильского джиу-джитсу. Ключом является прием, а значением — половина тела, к которой он применяется:

```
In [27]: submissions = {"armbar": "upper_body",
...:                   "arm_triangle": "upper_body",
...:                   "heel_hook": "lower_body",
...:                   "knee_bar": "lower_body"}
...:
```

Распространенный паттерн работы со словарями — итерация по словарю с помощью метода `items`. В следующем примере в консоль выводятся ключ и значение:

```
In [28]: for submission, body_part in submissions.items():
...:     print(f"The {submission} is an attack \
...:           on the {body_part}")
...:
```

```
The armbar is an attack on the upper_body
The arm_triangle is an attack on the upper_body
The heel_hook is an attack on the lower_body
The knee_bar is an attack on the lower_body
```

Словари можно также использовать для фильтрации. В нижеприведенном примере выводятся только болевые приемы (submission attack) на верхнюю половину тела (upper body):

```
In [29]: print(f"These are upper_body submission attacks\
...: in Brazilian jiu-jitsu:")
...: for submission, body_part in submissions.items():
...:     if body_part == "upper_body":
...:         print(submission)
...:
```

```
These are upper_body submission attacks in Brazilian jiu-jitsu:
armbar
arm_triangle
```

Можно также выбирать ключи и значения словарей:

```
In [30]: print(f"These are keys: {submissions.keys()}")
...: print(f"These are values: {submissions.values()}")
```

```
...:
These are keys: dict_keys(['armbar', 'arm_triangle',
    'heel_hook', 'knee_bar'])
These are values: dict_values(['upper_body', 'upper_body',
    'lower_body', 'lower_body'])
```

СПИСКИ

Списки часто используются в языке Python. Они дают возможность реализации последовательных коллекций. Списки могут включать в качестве элементов словари, точно так же, как и словари могут включать списки.

```
In [31]: list_of_bjj_positions = ["mount", "full-guard",
    "half-guard", "turtle",
    "side-control", "rear-mount",
    "knee-on-belly", "north-south",
    "open-guard"]
```

```
...:
In [32]: for position in list_of_bjj_positions:
...:     if "guard" in position:
...:         print(position)
...:
full-guard
half-guard
open-guard
```

Можно выбирать элементы списков посредством срезов:

```
In [35]: print(f'First position: {list_of_bjj_positions[:1]}')
...: print(f'Last position: {list_of_bjj_positions[-1]}')
...: print(f'First three positions:\
    {list_of_bjj_positions[0:3]}')
...:
First position: ['mount']
Last position: ['open-guard']
First three positions: ['mount', 'full-guard', 'half-guard']
```

ФУНКЦИИ

Функции — стандартные блоки программирования при исследовании данных на языке Python, а равно и способ создания логичных, удобных в тестировании структур. На протяжении десятилетий велись споры о том,

какой стиль программирования на языке Python лучше — функциональный или объектно-ориентированный. В этом разделе вы не найдете ответа на данный вопрос, но увидите, почему полезно знать основы работы с функциями в Python.

Написание функций

Важнейший навык для программиста на языке Python — умение писать функции. Владение основными навыками работы с функциями означает возможность использования практически всего, что предлагает язык.

Простая функция

Простейшие функции лишь возвращают значение:

```
In [1]: def favorite_martial_art():
...:     return "bjj"
In [2]: favorite_martial_art()
Out[2]: "bjj"
```

Документирование функций

Снабдить функцию документацией никогда не будет лишним. В блокнотах Jupyter и IPython для просмотра docstring достаточно обратиться к функции по имени с добавлением после объекта символа `?`, как показано ниже:

```
In [2]: favorite_martial_art_with_docstring?
Signature: favorite_martial_art_with_docstring()
Docstring: This function returns the name of my favorite martial art
File:      ~/src/functional_intro_to_python/
Type:      function
```

Описания docstring-функций можно вывести в консоль, обратившись к атрибуту `__doc__`:

```
In [4]: favorite_martial_art_with_docstring.__doc__
...:
Out[4]: 'This function returns the name of my favorite martial art'
```

Аргументы функций: позиционные, именованные

Наиболее полезны те функции, в которые можно передавать аргументы. В приведенной ниже функции обрабатываются новые значения для аргумента

times. Аргумент этой функции «позиционный», неименованный. Позиционные аргументы обрабатываются в порядке их создания:

```
In [5]: def practice(times):
...:     print(f"I like to practice {times} times a day")
...:
```

```
In [6]: practice(2)
I like to practice 2 times a day
```

```
In [7]: practice(3)
I like to practice 3 times a day
```

Позиционные аргументы: обрабатываются по порядку. Позиционные аргументы функций обрабатываются в порядке их описания. Поэтому их можно легко создавать и легко же перепутать.

```
In [9]: def practice(times, technique, duration):
...:     print(f"I like to practice {technique},\
...:           {times} times a day, for {duration} minutes")
...:
```

```
In [10]: practice(3, "leg locks", 45)
I like to practice leg locks, 3 times a day, for 45 minutes
```

Именованные аргументы: обрабатываются по ключу или значению, могут иметь значения по умолчанию. Одно из удобных свойств именованных аргументов — возможность задавать значения по умолчанию и указывать явным образом только те из них, которые нужно изменить.

```
In [12]: def practice(times=2, technique="kimura", duration=60):
...:     print(f"I like to practice {technique},\
...:           {times} times a day, for {duration} minutes")
```

```
In [13]: practice()
I like to practice kimura, 2 times a day, for 60 minutes
```

```
In [14]: practice(duration=90)
I like to practice kimura, 2 times a day, for 90 minutes
```

****kwargs и *args.** Синтаксисы ****kwargs** или ***args** позволяют передавать в функции динамические аргументы. Однако применять эти виды синтаксиса следует осмотрительно, поскольку они затрудняют понимание кода. При использовании там, где это целесообразно, они предоставляют широкие возможности.

```
In [15]: def attack_techniques(**kwargs):
...:     """Принимает произвольное число именованных аргументов"""
...:
...:     for name, attack in kwargs.items():
...:         print(f"This is an attack I would like\
...:             to practice: {attack}")
...:
```

```
In [16]: attack_techniques(arm_attack="kimura",
...:                       leg_attack="straight_ankle_lock",
...:                       neck_attack="arm_triangle")
...:
This is an attack I would like to practice: kimura
This is an attack I would like to practice: straight_ankle_lock
This is an attack I would like to practice: arm_triangle
```

Передача функции словаря именованных аргументов. Синтаксис `**kwargs` можно использовать для передачи всех аргументов сразу:

```
In [19]: attacks = {"arm_attack": "kimura",
...:                "leg_attack": "straight_ankle_lock",
...:                "neck_attack": "arm_triangle"}
In [20]: attack_techniques(**attacks)
This is an attack I would like to practice: kimura
This is an attack I would like to practice: straight_ankle_lock
This is an attack I would like to practice: arm_triangle
```

Передача функций в качестве аргументов. Объектно-ориентированное программирование — популярный способ программирования, но не единственный возможный в Python. В качестве вспомогательного стиля для организации конкурентности и исследования данных вполне подходит функциональное программирование.

В следующем примере благодаря передаче функции в качестве аргумента появляется возможность использования ее внутри другой функции:

```
In [21]: def attack_location(technique):
...:     """Возвращает место применения приема"""
...:
...:     attacks = {"kimura": "arm_attack",
...:               "straight_ankle_lock": "leg_attack",
...:               "arm_triangle": "neck_attack"}
...:     if technique in attacks:
...:         return attacks[technique]
```

```
...:     return "Unknown"
...:
```

```
In [22]: attack_location("kimura")
Out[22]: 'arm_attack'
In [24]: attack_location("bear hug")
...:
Out[24]: 'Unknown'
```

```
In [25]: def multiple_attacks(attack_location_function):
...:     """Принимает на входе функцию распределения приемов
...:         по категориям и возвращает место применения приема"""
...:
...:     new_attacks_list = ["rear_naked_choke",
...:         "americana", "kimura"]
...:     for attack in new_attacks_list:
...:         attack_location = attack_location_function(attack)
...:         print(f"The location of attack {attack} \
...:             is {attack_location}")
...:
```

```
In [26]: multiple_attacks(attack_location)
The location of attack rear_naked_choke is Unknown
The location of attack americana is Unknown
The location of attack kimura is arm_attack
```

Замыкания и каррирование функций. *Замыкания* (closures) — функции, содержащие другие функции. В языке Python их часто применяют для отслеживания состояния. В приведенном ниже примере внешняя функция `attack_counter` отслеживает число атакующих приемов. Внутренняя функция `attack_filter` использует ключевое слово `nonlocal` языка Python 3 для изменения значения переменной во внешней функции.

Такой подход носит название *каррирования функций* (functional currying) и позволяет создавать на основе общих функций специализированные. Как показано ниже, подобные функции могут лечь в основу простой компьютерной игры или подсчета статистики поединка в смешанных единоборствах:

```
In [1]: def attack_counter():
...:     """Подсчитывает количество атак на конкретную половину
...:         тела"""
...:     lower_body_counter = 0
...:     upper_body_counter = 0
...:     def attack_filter(attack):
```

```

...:     nonlocal lower_body_counter
...:     nonlocal upper_body_counter
...:     attacks = {"kimura": "upper_body",
...:               "straight_ankle_lock": "lower_body",
...:               "arm_triangle": "upper_body",
...:               "keylock": "upper_body",
...:               "knee_bar": "lower_body"}
...:     if attack in attacks:
...:         if attacks[attack] == "upper_body":
...:             upper_body_counter += 1
...:         if attacks[attack] == "lower_body":
...:             lower_body_counter += 1
...:     print(f"Upper Body Attacks {upper_body_counter},\
...:           Lower Body Attacks {lower_body_counter}")
...:     return attack_filter
...:

```

```
In [2]: fight = attack_counter()
```

```
In [3]: fight("kimura")
Upper Body Attacks 1, Lower Body Attacks 0
```

```
In [4]: fight("knee_bar")
Upper Body Attacks 1, Lower Body Attacks 1
```

```
In [5]: fight("keylock")
Upper Body Attacks 2, Lower Body Attacks 1
```

Функции, использующие ключевое слово `yield` (генераторы). Отложенное вычисление — удобный стиль программирования, одним из примеров которого являются генераторы. Генераторы выдают значение в нужный момент времени.

Нижеприведенный пример возвращает бесконечную случайную последовательность атакующих приемов. Отложенность здесь заключается в том, что, хотя количество значений бесконечно, возвращаются они только при вызове функции.

```
In [6]: def lazy_return_random_attacks():
...:     """Каждый раз возвращает атакующие приемы"""
...:     import random
...:     attacks = {"kimura": "upper_body",
...:               "straight_ankle_lock": "lower_body",
...:               "arm_triangle": "upper_body",

```

```

...:         "keylock": "upper_body",
...:         "knee_bar": "lower_body"}
...:     while True:
...:         random_attack = random.choices(list(attacks.keys()))
...:         yield random_attack
...:

```

In [7]: attack = lazy_return_random_attacks()

In [8]: next(attack)

Out[8]: ['straight_ankle_lock']

```

In [9]: attacks = {"kimura": "upper_body",
...:               "straight_ankle_lock": "lower_body",
...:               "arm_triangle": "upper_body",
...:               "keylock": "upper_body",
...:               "knee_bar": "lower_body"}
...:

```

```

In [10]: for _ in range(10):
...:     print(next(attack))
...:

```

```

['keylock']
['arm_triangle']
['arm_triangle']
['arm_triangle']
['knee_bar']
['arm_triangle']
['knee_bar']
['kimura']
['arm_triangle']
['kimura']

```

Декораторы: функции, служащие адаптерами для других функций.

Еще одна полезная возможность Python — синтаксис декоратора, позволяющий обернуть одну функцию в другую. В нижеприведенном примере написан декоратор, добавляющий к каждому вызову функции паузу случайной длины. В сочетании с вышеописанным «бесконечным» генератором атакующих приемов он генерирует паузы случайной длины между вызовами функций.

```

In [12]: def randomized_speed_attack_decorator(function):
...:     """Вносит элемент случайности в скорость атак"""

```

```

...:
...: import time
...: import random
...:
...: def wrapper_func(*args, **kwargs):
...:     sleep_time = random.randint(0,3)
...:     print(f"Attacking after {sleep_time} seconds")
...:     time.sleep(sleep_time)
...:     return function(*args, **kwargs)
...: return wrapper_func

```

```

In [13]: @randomized_speed_attack_decorator
...: def lazy_return_random_attacks():
...:     """Каждый раз возвращает атакующие приемы"""
...:     import random
...:     attacks = {"kimura": "upper_body",
...:               "straight_ankle_lock": "lower_body",
...:               "arm_triangle": "upper_body",
...:               "keylock": "upper_body",
...:               "knee_bar": "lower_body"}
...:     while True:
...:         random_attack = random.choices(list(attacks.keys()))
...:         yield random_attack
...:
In [14]: for _ in range(10):
...:     print(next(lazy_return_random_attacks()))
...:

```

```

Attacking after 1 seconds
['knee_bar']
Attacking after 0 seconds
['arm_triangle']
Attacking after 2 seconds
['knee_bar']

```

Использование метода `apply` в библиотеке `Pandas`. Последний мой урок на тему функций — использование их для обработки объектов `DataFrame` библиотеки `Pandas`. Один из базовых принципов `Pandas` — применение операции `apply` к столбцу вместо организации итераций по всем значениям. Этот пример демонстрирует округление всех чисел до целых:

```

In [1]: import pandas as pd
...: iris = pd.read_csv('https://raw.githubusercontent.com/mwaskom/
...: seaborn-data/master/iris.csv')

```

```

...: iris.head()
...:
Out[1]:
  sepal_length  sepal_width  petal_length  petal_width  species
0           5.1           3.5           1.4           0.2  setosa
1           4.9           3.0           1.4           0.2  setosa
2           4.7           3.2           1.3           0.2  setosa
3           4.6           3.1           1.5           0.2  setosa
4           5.0           3.6           1.4           0.2  setosa

```

```

In [2]: iris['rounded_sepal_length'] = \
        iris[['sepal_length']].apply(pd.Series.round)
...: iris.head()
...:

```

```

Out[2]:
  sepal_length  sepal_width  petal_length  petal_width  species \
0           5.1           3.5           1.4           0.2  setosa
1           4.9           3.0           1.4           0.2  setosa
2           4.7           3.2           1.3           0.2  setosa
3           4.6           3.1           1.5           0.2  setosa
4           5.0           3.6           1.4           0.2  setosa

```

```

  rounded_sepal_length
0                5.0
1                5.0
2                5.0
3                5.0
4                5.0

```

Мы воспользовались встроенной функцией, но можно было написать и применить к столбцу и свою, пользовательскую. В приведенном далее примере числа умножаются на 100. Другой способ сделать это — организовать цикл, преобразовать данные, после чего записать их обратно. В Pandas гораздо проще и понятнее будет применить взамен этого пользовательскую функцию.

```

In [3]: def multiply_by_100(x):
...:     """Умножает на 100"""
...:     return x*100
...: iris['100x_sepal_length'] = \
iris[['sepal_length']].apply(multiply_by_100)
...: iris.head()
...:

```

```

  rounded_sepal_length  100x_sepal_length
0                5.0                510.0

```

1	5.0	490.0
2	5.0	470.0
3	5.0	460.0
4	5.0	500.0

Использование управляющих конструкций

В этом разделе описываются основные управляющие конструкции языка Python. В обычном Python основным компонентом управляющих конструкций является цикл `for`. Стоит отметить, однако, что циклы `for` редко используются с библиотекой `Pandas` и то, что отлично работает в Python, плохо вписывается в парадигму `Pandas`. Некоторые примеры:

- циклы `for`;
- циклы `while`;
- операторы `if/else`;
- `Try/Except`;
- выражения-генераторы;
- списковое включение;
- сопоставление с шаблоном.

Всем программам нужно как-то управлять последовательностью выполнения. В этом разделе описаны некоторые из методов такого управления.

Циклы `for`

Цикл `for` — одна из важнейших управляющих конструкций языка Python. Один из часто используемых паттернов — применение функции `range` для генерации диапазона значений с последующим проходом по ним в цикле:

```
In [4]: res = range(3)
...: print(list(res))
...:
[0, 1, 2]
In [5]: for i in range(3):
...:     print(i)
...:
0
1
2
```

Цикл for по списку. Еще один распространенный паттерн — проход в цикле по списку:

```
In [6]: martial_arts = ["Sambo", "Muay Thai", "BJJ"]
...: for martial_art in martial_arts:
...:     print(f"{martial_art} has influenced\
              modern mixed martial arts")
...:
Sambo has influenced modern mixed martial arts
Muay Thai has influenced modern mixed martial arts
BJJ has influenced modern mixed martial arts
```

Циклы while

Циклы while часто используются в качестве способа выполнения цикла вплоть до момента, когда выполнится некое условие. Их часто применяют для создания бесконечных циклов. В следующем примере цикл while используется, чтобы отфильтровать один из двух типов атакующих приемов, возвращаемых функцией:

```
In [7]: def attacks():
...:     list_of_attacks = ["lower_body", "lower_body",
...:                       "upper_body"]
...:     print(f"There are a total of {len(list_of_attacks)}\
              attacks coming!")
...:     for attack in list_of_attacks:
...:         yield attack
...: attack = attacks()
...: count = 0
...: while next(attack) == "lower_body":
...:     count +=1
...:     print(f"crossing legs to prevent attack #{count}")
...: else:
...:     count +=1
...:     print(f"This is not a lower body attack, \
              I will cross my arms for #{count}")
...:
There are a total of 3 attacks coming!
crossing legs to prevent attack #1
crossing legs to prevent attack #2
This is not a lower body attack, I will cross my arms for #3
```

Операторы if/else

Использование операторов if/else — распространенный способ ветвления логики программы. В следующем примере для выбора одной из веток применяется if/elif. Если ни одна из веток не подходит, выполняется последний оператор else:

```
In [8]: def recommended_attack(position):
...:     """Рекомендует атакующий прием в зависимости от позы
...:         противника"""
...:     if position == "full_guard":
...:         print(f"Try an armbar attack")
...:     elif position == "half_guard":
...:         print(f"Try a kimura attack")
...:     elif position == "full_mount":
...:         print(f"Try an arm triangle")
...:     else:
...:         print(f"You're on your own, \
...:             there is no suggestion for an attack")
```

```
In [9]: recommended_attack("full_guard")
```

```
Try an armbar attack
```

```
In [10]: recommended_attack("z_guard")
```

```
You're on your own, there is no suggestion for an attack
```

Выражения-генераторы

Выражения-генераторы расширяют идею ключевого слова `yield` за счет отложенных вычислений последовательностей. Преимущество выражений-генераторов состоит в том, что ничего не вычисляется и не помещается в оперативную память до момента фактического вычисления. Именно поэтому в нижеприведенном примере можно оперировать генерируемой бесконечной последовательностью случайно выбранных атак.

В этом конвейере генераторов атака в нижнем регистре, например, `"arm_triangle"` преобразуется в `"ARM_TRIANGLE"`, а затем удаляется символ подчеркивания: `"ARM TRIANGLE"`.

```
In [11]: def lazy_return_random_attacks():
...:     """Каждый раз возвращает атакующие приемы"""
...:     import random
```

```

...:     attacks = {"kimura": "upper_body",
...:                 "straight_ankle_lock":"lower_body",
...:                 "arm_triangle":"upper_body",
...:                 "keylock": "upper_body",
...:                 "knee_bar": "lower_body"}
...:     while True:
...:         random_attack = random.choices(list(attacks.keys()))
...:         yield random_attack
...:
...: #Переводит все атаки в верхний регистр
...: upper_case_attacks = \
...:     (attack.pop().upper() for attack in \
...:      lazy_return_random_attacks())

```

In [12]: next(upper_case_attacks)

Out[12]: 'ARM_TRIANGLE'

In [13]: ## Конвейер генераторов: одно выражение связывается в цепочку
со следующим

```

...: #Переводит все атаки в верхний регистр
...: upper_case_attacks = \
...:     (attack.pop().upper() for attack in \
...:      lazy_return_random_attacks())
...: #Удаляет символ подчеркивания
...: remove_underscore = \
...:     (attack.split("_") for attack in \
...:      upper_case_attacks)
...: #Создает новую фразу
...: new_attack_phrase = \
...:     (" ".join(phrase) for phrase in \
...:      remove_underscore)
...:

```

In [19]: next(new_attack_phrase)

Out[19]: 'STRAIGHT ANKLE LOCK'

In [20]: for number in range(10):

```

...:     print(next(new_attack_phrase))
...:

```

KIMURA

```
KEYLOCK
STRAIGHT ANKLE LOCK
...
```

Списковое включение

Списковое включение синтаксически напоминает выражения-генераторы, но, в отличие от них, вычисляется в памяти. Кроме того, оно реализовано в виде оптимизированного кода на языке С, часто значительно превышающего по производительности обычный цикл `for`:

```
In [21]: martial_arts = ["Sambo", "Muay Thai", "BJJ"]
new_phrases = [f"Mixed Martial Arts is influenced by \
               {martial_art}" for martial_art in martial_arts]
In [22]: print(new_phrases)
['Mixed Martial Arts is influenced by Sambo', \
 'Mixed Martial Arts is influenced by Muay Thai', \
 'Mixed Martial Arts is influenced by BJJ']
```

Промежуточные вопросы

После обсуждения всех этих базовых вещей важно разобраться не просто в том, как написать код, а в том, как написать код, удобный для сопровождения. Один из способов — создать библиотеку; другой — использовать уже готовый код путем установки сторонней библиотеки. Вообще говоря, основная идея — минимизировать сложность путем разбиения.

Написание библиотеки на языке Python

Достаточно скоро после начала создания вашего проекта вам понадобится писать библиотеки. Основное, что нужно знать: в репозитории есть каталог `funclib`, внутри которого находится файл `__init__.py`. Для создания библиотеки необходимо поместить в этот каталог модуль с какой-либо функцией внутри него.

Создаем файл:

```
touch funclib/funcmod.py
```

Помещаем в этот файл функцию:

```
"""Простой модуль"""  
  
def list_of_belts_in_bjj():  
    """Возвращает список поясов бразильского джиу-джитсу"""  
  
    belts = ["white", "blue", "purple", "brown", "black"]  
    return belts
```

Импорт библиотеки

В Jupyter, если библиотека находится в каталоге на уровень выше, можно добавить для импорта его в системный путь посредством команды `sys.path.append`. После этого можно импортировать модуль с помощью пространства имен ранее созданного каталога/файла/функции:

```
In [23]: import sys;sys.path.append("../")  
        ...: from funclib import funcmod  
In [24]: funcmod.list_of_belts_in_bjj()  
Out[24]: ['white', 'blue', 'purple', 'brown', 'black']
```

Установка других библиотек с помощью команды `pip install`

Установить другие библиотеки можно с помощью команды `pip install`. Отмечу, что система управления пакетами `conda` (<https://conda.io/docs/user-guide/tasks/manage-pkgs.html>) может служить альтернативой и заменой `pip`. Если вы используете `conda`, то сможете использовать и `pip`, поскольку среда `conda` является заменой `virtualenv`, но также может и напрямую устанавливать пакеты.

Для установки пакета `pandas` выполните следующую команду:

```
pip install pandas
```

В качестве альтернативного варианта пакеты можно установить с помощью файла `requirements.txt`:

```
> cat requirements.txt  
pylint  
pytest  
pytest-cov
```

```
click
jupyter
nbval
```

```
> pip install -r requirements.txt
```

Вот пример использования небольшой библиотеки в блокноте Jupyter. Стоит отметить важный нюанс: создать в блокноте Jupyter гигантскую паутину процедурного кода очень легко. А проще всего избежать этого, создавая библиотеки с последующим их тестированием и импортированием.

```
"""Простой модуль"""
```

```
import pandas as pd
```

```
def list_of_belts_in_bjj():
    """Возвращает список поясов бразильского джиу-джитсу"""
```

```
    belts = ["white", "blue", "purple", "brown", "black"]
    return belts
```

```
def count_belts():
    """Использует библиотеку Pandas для подсчета числа поясов"""
```

```
    belts = list_of_belts_in_bjj()
    df = pd.DataFrame(belts)
    res = df.count()
    count = res.values.tolist()[0]
    return count
```

```
In [25]: from funclib.funcmod import count_belts
...:
```

```
In [26]: print(count_belts())
...:
```

```
5
```

Классы

В блокнотах Jupyter можно итеративно использовать классы и взаимодействовать с ними. Простейший тип класса представляет собой просто наименование, как показано ниже:

```
class Competitor: pass
```

Экземпляры этого класса могут представлять собой различные объекты:

```
In [27]: class Competitor: pass
In [28]:
...: conor = Competitor()
...: conor.name = "Conor McGregor"
...: conor.age = 29
...: conor.weight = 155
In [29]: nate = Competitor()
...: nate.name = "Nate Diaz"
...: nate.age = 30
...: nate.weight = 170
In [30]: def print_competitor_age(object):
...:     """Выводит статистику по возрасту спортсменов"""
...:
...:     print(f"{object.name} is {object.age} years old")
In [31]: print_competitor_age(nate)
Nate Diaz is 30 years old
In [32]:
...: print_competitor_age(conor)
Conor McGregor is 29 years old
```

Различия классов и функций

Ключевые различия классов и функций:

- функции гораздо легче обсуждать;
- состояние функций (обычно) существует лишь внутри функции, а состояние класса может сохраняться и за ее пределами;
- классы предлагают более высокий уровень абстракции за счет повышения сложности.

Резюме

Эта глава подготавливает почву для всей остальной книги. Она началась с краткого руководства по «функциональному» Python и его использованию для разработки приложений машинного обучения. В последующих главах мы обсудим подробнее конкретные детали машинного обучения на основе облачных сервисов.

Ключевые компоненты машинного обучения можно разбить на несколько широких категорий. Методы машинного обучения с учителем применяются, когда правильный ответ заранее известен. Примером может служить прогноз цен на жилье на основе предыдущих исторических данных. Методы *машинного обучения без учителя* (unsupervised machine learning) применяются, когда правильный ответ не известен заранее. Самый распространенный пример машинного обучения без учителя — алгоритмы кластеризации.

В случае *машинного обучения с учителем* (supervised machine learning) данные заранее снабжены метками, а цель — правильный прогноз. С практической точки зрения ИИ существуют некоторые методики, способные еще более расширить возможности машинного обучения с учителем. Один из таких подходов — *перенос обучения* (transfer learning), представляющий собой подгонку предварительно обученной модели к новой задаче с помощью намного меньшего набора данных. Еще один подход — *активное обучение* (active learning), при котором алгоритм сам выбирает, какие обучающие данные ему нужно получить посредством дорогостоящего маркирования данных вручную, в зависимости от потенциальной пользы этих данных.

В отличие от него при машинном обучении без учителя никаких меток нет, а цель состоит в выявлении в данных скрытых закономерностей. Существует и третий тип машинного обучения — *обучение с подкреплением* (reinforcement learning), используемое реже и в этой книге подробно не рассматриваемое. Обучение с подкреплением часто используется, например, для того, чтобы научить компьютер играть в игры Atari по исходным пикселям или играть в го (<https://gogameguru.com/i/2016/03/deepmind-mastering-go.pdf>). Глубокое обучение — методика машинного обучения, часто используемая в фермах графических процессоров (graphics processing unit, GPU) с помощью провайдеров облачных сервисов. Глубокое обучение часто используется для решения задач классификации, например для распознавания образов, но подходит и для других задач. В настоящее время существует более дюжины компаний, работающих над микросхемами для глубокого обучения, что наглядно демонстрирует, насколько важным становится глубокое обучение для специалистов, работающих в сфере машинного обучения.

Существует две разновидности машинного обучения с учителем: регрессия и классификация. Методы машинного обучения с учителем на основе регрессии служат для прогноза непрерывных значений. Методы машинного обучения с учителем на базе классификации необходимы в основном для прогноза меток на основе исторических данных.

Наконец, в этой книге мы рассмотрим ИИ в контексте как автоматизации, так и имитации познавательных функций. Множество готовых ИИ-решений сейчас доступно через API крупных компаний, занимающихся разработкой ПО: Google, Amazon, Microsoft и IBM. Мы подробно рассмотрим вопрос интеграции этих API в проекты ИИ.

2

ИИ и инструменты машинного обучения

Мы говорим про тренировку.
Мы говорим про тренировку.
Мы не говорим про игру.

*Ален Айверсон
(Allen Iverson)*

Машинному обучению и ИИ посвящается все больше статей, видео, курсов и дипломов. Гораздо меньше в литературе охвачен соответствующий набор инструментальных средств. Каковы основные навыки, необходимые для исследователей данных, которые создают программы машинного обучения для промышленной эксплуатации? Какие основные процессы должны быть налажены в компании, чтобы она могла разрабатывать надежные автоматизированные системы для прогнозов?

Сложность набора инструментов, необходимых для успешной работы в области науки о данных, сильно выросла в сложности в одних областях и сократилась в других. Блокноты Jupiter — одно из новшеств, снизивших сложность разработки решений, как и облачные вычисления и системы сборки SaaS. Философия DevOps касается многих сфер, включая безопасность и надежность данных. Взлом компании Equifax в 2017 г. и скандал с Facebook и Cambridge Analytica в 2018-м, приведшие к утечке информации о значительной доле американских граждан, продемонстрировали, что безопасность данных — одна из самых насущных проблем современности. Именно эти вопросы и рассматриваются в данной главе, включая рекомендации по разработке процессов, которые повышали бы надежность и безопасность систем машинного обучения.

Экосистема исследования данных языка Python: IPython, Pandas, NumPy, блокнот Jupiter, Sklearn

Экосистема Python уникальна своей историей. Я помню, как впервые услышал про Python в 2000-м, работая в Caltech. Тогда это был очень малоизвестный язык, но уже понемногу приобретавший популярность в научных кругах, как потенциально прекрасно подходящий для обучения основам компьютерных наук.

Одна из проблем обучения основам компьютерных наук посредством таких языков, как C или Java, — масса дополнительных нюансов, о которых приходится думать, помимо базовых понятий вроде циклов `for`. Прошло 20 лет — и эта проблема практически решена благодаря языку Python, ставшему, вероятно, стандартом для обучения основам компьютерных наук во всем мире.

Неудивителен и колоссальный прогресс языка Python в трех наиболее интересующих меня областях: DevOps, облачной архитектуре и, конечно, науке о данных. Все эти области представляются мне очевидным образом взаимосвязанными. Меня позабавил побочный эффект такой эволюции, выразившийся в возможности вносить вклад во весь стек технологий компании с помощью одного-единственного языка программирования; более того, оказалось, что делать это можно очень быстро и с потрясающими возможностями масштабирования благодаря таким сервисам, как AWS Lambda.

Блокноты Jupiter — тоже весьма интересные штуковины. Около десяти лет назад я был соавтором книги, в которой для большинства примеров использовался IPython (в общем, как и в книге, которую вы держите в руках), и с тех пор я практически и не переставал его использовать. Так что, когда Jupiter обрел популярность, он оказался в самый раз. При работе с Python меня все время ожидают приятные сюрпризы.

По мере моего роста как аналитика, однако, я немало времени провел, работая с языком R, и постепенно полюбил его стиль, весьма отличающийся от стиля Python. Например, встроенные в язык объекты `DataFrame`, функции построения графиков и продвинутые статистические функции, а также возможности более «чистого» функционального программирования.

По сравнению с языком R потенциал применения Python на практике гораздо больше, если принимать во внимание возможности интеграции

с облачными сервисами и предназначенные для этого библиотеки, но для целей «чистой» науки о данных он представляется плохо сбалансированным. Например, при работе с библиотекой Pandas никогда не задействуется цикл `for`, а в обычном Python это весьма распространенное явление. При совместном использовании обычного Python с библиотеками Pandas, scikit-learn и NumPy возникают определенные конфликты парадигм, но если вы пользователь Python, то легко с ними справитесь.

Язык R, RStudio, Shiny и ggplot

Даже если вы используете только Python, все равно вам не помешает знакомство с языком R и его инструментами. Экосистема языка R обладает возможностями, которые заслуживают обсуждения. Скачать его можно с одного из зеркальных сайтов по адресу <https://cran.r-project.org/mirrors.html>.

Основная интегрированная среда разработки (integrated development environment, IDE) для языка R — RStudio (<https://www.rstudio.com/>). Она обладает множеством замечательных возможностей, включая возможность создания приложений Shiny (<http://shiny.rstudio.com/>) и приложений на языке разметки R — Markdown (<https://rmarkdown.rstudio.com/>). Если вы только начинаете работать с языком R, обязательно попробуйте RStudio.

Shiny еще развивающаяся технология, но заслуживает упоминания в контексте интерактивного анализа данных. Она позволяет создавать на основе кода на чистом языке R интерактивные веб-приложения, пригодные для промышленной эксплуатации. В галерее приложений есть множество вдохновляющих примеров (<https://shiny.rstudio.com/gallery/>).

Еще одна сильная сторона языка R — передовые статистические библиотеки. Язык R был создан в городе Окленд, Новая Зеландия, именно для специалистов по статистике и заслуженно пользуется хорошей репутацией в данном сообществе. Уже одного этого достаточно, чтобы рекомендовать использовать R.

Наконец, библиотека построения графиков ggplot также очень хороша и настолько совершенна, что мне зачастую случалось выгружать данные из проекта Python в виде CSV-файла, загружать их в RStudio и создавать прекрасные визуализации на языке R с помощью библиотеки ggplot. Пример этого показан в главе 6 «Прогноз популярности в соцсетях в НБА».

Электронные таблицы: Excel и Google Sheets

Excel часто критиковали в последние годы. Но как вспомогательные, а не основные средства Excel и Google Sheets очень удобны. В частности, Excel — великолепный инструмент для подготовки и очистки данных. Во время процесса быстрой разработки прототипа для задачи исследования данных очищать и форматировать данные будет быстрее в Excel.

Аналогично Google Sheets — замечательная технология для решения реальных задач. В главе 4 «Разработка ИИ в облачной среде с помощью облачной платформы Google» показан пример того, насколько легко программно создавать электронные таблицы Google. Возможно, электронные таблицы — устаревшая технология, но они все равно очень удобны при создании решений для промышленной эксплуатации.

Разработка облачных приложений ИИ с помощью веб-сервисов Amazon

Веб-сервисы Amazon (AWS) — безоговорочный лидер среди облачных сервисов. В Amazon есть почти полтора десятка принципов лидерства (<https://www.amazon.jobs/principles>), намечающих в общих чертах парадигму для работников компании. В конце списка располагается пункт «Достигайте результатов». Именно это и делает их облачная платформа, начиная со своего появления в 2006 г. Цены непрерывно падают, быстрыми темпами совершенствуются существующие сервисы, и добавляются новые.

В последние годы технологии больших данных, ИИ и машинного обучения на AWS сделали колоссальный скачок вперед. Наблюдались и определенные подвижки в отношении бессерверных технологий, подобных AWS Lambda. Во многих новых технологиях начальная версия цепляется за вчерашний день, а следующее поколение многое из этого вчерашнего дня отбрасывает. Истоки первой версии облака Amazon явно лежали в центре обработки данных (ЦОД): виртуальные машины, реляционные базы данных и т. п. Следующее поколение, бессерверное, представляет собой нативную облачную технологию. Операционная система и прочие нюансы абстрагируются, остается только код для решения задач.

Подобный продвинутый способ написания кода идеально подходит для облачных приложений машинного обучения и ИИ. В данной главе описывается применение этих новых технологий к созданию облачных приложений ИИ.

Интеграция разработки и эксплуатации на AWS

Мне часто приходится слышать от исследователей данных и разработчиков фразу: «Интеграция разработки и эксплуатации — не моя задача». Но интеграция разработки и эксплуатации (DevOps) — не задача, а образ мышления. Самые сложные виды ИИ относятся к автоматизации сложных задач, выполняемых человеком, например управления автомобилем. Интеграция разработки и эксплуатации имеет много общего с этим образом мышления. И действительно, почему бы инженеру, занимающемуся машинным обучением, не создать эффективную петлю обратной связи для развертывания ПО в промышленной эксплуатации?

Облачные технологии — и, в частности, облачные технологии AWS — сделали возможными автоматизацию и эффективность в невиданных доселе масштабах. Вот некоторые из решений, осуществляющих интеграцию разработки и эксплуатации: виртуальные узлы Spot, OpsWorks, Elastic Beanstalk, Lambda, CodeStar, CodeCommit, CodeBuild, CodePipeline и CodeDeploy. В данной главе вы найдете примеры всех¹ этих сервисов вместе с обзором их использования с точки зрения специалиста по машинному обучению.

Непрерывная поставка

В среде непрерывной поставки программное обеспечение всегда готово к выпуску очередной версии. Принципиальная схема такой среды соответствует схеме работы заводского конвейера. Существует замечательный онлайн-ресурс, <http://www.devopsessentialsaws.com/>, разработанный совместно издательством Pearson Education и компанией Stelligent, на котором обсуждаются многие из этих вопросов. Там вы найдете отличный обзор интеграции разработки и эксплуатации на AWS.

¹ Автор несколько преувеличивает, на самом деле не всех.

Создание среды разработки ПО для AWS

При работе с AWS легко упустить такую маленькую, но важную деталь, как настройка основной среды разработки. А поскольку настройки Python составляют для многих разработчиков алгоритмов машинного обучения столь значительную часть этого процесса, то имеет смысл поставить его во главу угла. Настройки включают создание сборочного файла (`makefile`), виртуальной среды, добавление в командную оболочку `bash` или `zsh` команды быстрого доступа для переключения в нее, автоматического выполнения профиля AWS в текущей среде и т. д.

Небольшое примечание относительно сборочных файлов: они впервые появились в 1976 г. в Лабораториях Белла и используются в качестве утилиты сборки с отслеживанием зависимостей. Системы сборки могут оказаться очень сложными, но для многих проектов машинного обучения они практически бесценны. Одна из причин этого — доступность их в любой Unix/Linux без необходимости установки дополнительного программного обеспечения. Плюс это, так сказать, стандарт, понятный всем разработчикам в проекте.

Я довольно часто использовал виртуальные среды в киноиндустрии, которую можно считать одной из первых отраслей промышленности, связанных с большими данными. Даже в конце 2000-х в студиях, где я работал, были файловые серверы петабайтного (или около того) размера, и утилиты для задания переменных среды для проекта были единственным способом справиться с деревом каталогов.

Во время моего программирования на Python для студии Weta Digital и фильма «Аватар» объем данных на файловых серверах вырос настолько, что дальнейший рост оказался невозможен и приходилось синхронизировать громадные копии данных по нескольким файловым серверам. На самом деле один из моих второстепенных проектов как раз и был посвящен налаживанию работы Python на файловых серверах. А поскольку система импорта языка Python очень жадно ищет импорты, то иногда запуск сценария занимал до 30 с, так как он просматривал добрую сотню тысяч файлов по своим путям. Мы обнаружили это с помощью утилиты `strace` на операционной системе Linux, после чего «взломали» интерпретатор Python, заставив его игнорировать пути Python.

Virtualenv в Python и conda из дистрибутива Anaconda делают примерно то же, с чем я столкнулся при работе в киноиндустрии. Они создают для проекта изолированные переменные среды. Это далее позволяет пользователю переключаться между проектами, над которыми он работает, без каких-либо конфликтов версий библиотек.

В листинге 2.1 приведено начало простого сборочного файла.

Листинг 2.1. Простой сборочный файл проекта на языке Python для AWS

```
setup:
    python3 -m venv ~/.pragai-aws
install:
    pip install -r requirements.txt
```

Использование сборочного файла — рекомендуемая практика, поскольку он служит своеобразной точкой отсчета для сборки проекта локально, на сервере сборки, в контейнере Docker и для промышленной эксплуатации. Для использования этого сборочного файла в новом репозитории Git наберите:

```
→ pragai-aws git:(master) X make setup
python3 -m venv ~/.pragai-aws
```

Данная команда создаст новую виртуальную среду Python в каталоге ~/.pragai-aws. Как упоминается и в других главах книги, обычно имеет смысл создать псевдоним, чтобы выполнять для среды за один раз команду source и переход (cd) в ее каталог. Пользователи командной оболочки Z или bash могут отредактировать файлы .zshrc или .bashrc соответственно, добавив в них псевдоним для git checkout этого репозитория.

```
Alias pawstop="cd ~/src/pragai-aws &&\
    source ~/.pragai-aws/bin/activate"
```

И далее выполнение команды source для данной среды требует лишь написания псевдонима:

```
→ pragai-aws git:(master) X pawstop
(.pragai-aws) → pragai-aws git:(master) X
```

Трюк, благодаря которому все работает, заключается в использовании сценария activate. Этот же сценарий служит удобным механизмом контроля других переменных среды проекта, а именно PYTHONPATH и AWS_PROFILE, которые мы

подробно рассмотрим далее. Следующий этап настройки проекта для AWS состоит в создании учетной записи, если у вас ее еще нет, и пользователя в этой учетной записи (опять же если у вас его еще нет). У Amazon есть прекрасные инструкции по созданию пользовательских учетных записей для идентификации и управления доступом (identity and access management, IAM): http://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html.

После завершения настройки учетной записи (в соответствии с официальными инструкциями AWS) необходимо создать поименованный профиль. Для этого тоже есть отличное руководство от AWS: <http://docs.aws.amazon.com/cli/latest/userguide/cli-multiple-profiles.html>. Главное здесь — создавать такие учетные записи, чтобы было понятно, что проект использует профиль с конкретным именем или ролью (см. подробности в документации AWS: <http://docs.aws.amazon.com/cli/latest/userguide/cli-roles.html>).

Для установки утилиты командной строки AWS и Boto3 (Boto3 — последняя версия Boto на момент написания данной книги) внесите их в файл `requirements.txt` и выполните команду `make install`:

```
(.pragai-aws) → ✗ make install
pip install -r requirements.txt
```

После установки утилиты командной строки AWS необходимо настроить ее для нового пользователя:

```
(.pragai-aws) → ✗ aws configure --profile pragai
AWS Access Key ID [*****XQDA]:
AWS Secret Access Key [*****nmkH]:
Default region name [us-west-2]:
Default output format [json]:
```

Теперь можно использовать в утилите командной строки AWS параметр `--profile`. Для тестирования этой возможности проще всего попытаться вывести список содержимого одного из наборов данных машинного обучения, размещенных на AWS, например проекта GDELT (глобальная база событий, языков и тональностей, Global Database of Events, Language and Tone): <https://aws.amazon.com/public-datasets/gdelt/>.

```
(.pragai-aws) → aws s3 cp\
s3://gdelt-open-data/events/1979.csv .
fatal error: Unable to locate credentials
```

Благодаря выбору параметра `--profile` команда `download` может извлечь нужный файл из хранилища S3. Ничего сложного здесь нет, если не забывать всегда указывать флаг `--profile`, но при интенсивном использовании командной строки AWS это достаточно утомительно.

```
(.pragai-aws) aws --profile pragai s3 cp\
    s3://gdelt-open-data/events/1979.csv .
download: s3://gdelt-open-data/events/1979.csv to ./1979.csv
(.pragai-aws) → du -sh 1979.csv
110M    1979.csv
```

Чтобы решить эту проблему, можно поместить переменную `AWS_PROFILE` в сценарий `activate` для `virtualenv`:

```
(.pragia-aws) → vim ~/.pragai-aws/bin/activate
#Export AWS Profile
AWS_PROFILE="pragai"
export AWS_PROFILE
```

Теперь при запуске виртуальной среды в текущем окружении автоматически используется соответствующий профиль AWS.

```
(.pragia-aws) → echo $AWS_PROFILE
pragai
(.pragia-aws) → aws s3 cp\
    s3://gdelt-open-data/events/1979.csv .
download: s3://gdelt-open-data/events/1979.csv to ./1979.csv
```

Настройки проекта Python для AWS

Следующее, что нужно сделать после настройки `virtualenv` и указания учетных данных AWS, — задать конфигурацию кода Python. Для повышения эффективности и продуктивности разработки очень важна правильная структура проекта. Вот пример создания простой структуры проекта Python/AWS:

```
(.pragia-aws) → mkdir paws
(.pragia-aws) → touch paws/__init__.py
(.pragia-aws) → touch paws/s3.py
(.pragia-aws) → mkdir tests
(.pragia-aws) → touch tests/test_s3.py
```

Далее мы можем написать простой модуль S3, использующий этот макет проекта. Для создания функции скачивания файла из хранилища S3 применяется библиотека Boto3. Импортируется также библиотека журналирования (листинг 2.2).

Листинг 2.2. Модуль S3

```
"""
Методы S3 для библиотеки PAWS
"""

import boto3
from sensible.loginit import logger

log = logger("Paws")

def boto_s3_resource():
    """Создаем ресурс "S3" библиотеки boto3"""

    return boto3.resource("s3")

def download(bucket, key, filename, resource=None):
    """Скачиваем файл из хранилища S3"""

    if resource is None:
        resource = boto_s3_resource()
    log_msg = "Attempting download: {bucket}, {key}, {filename}."\
        format(bucket=bucket, key=key, filename=filename)
    log.info(log_msg)
    resource.meta.client.download_file(bucket, key, filename)
    return filename
```

Чтобы использовать эту только что сформированную библиотеку из командной строки IPython, достаточно нескольких строк кода для создания пространства имен paws:

```
In [1]: from paws.s3 import download
```

```
In [2]: download(bucket="gdelt-open-data",\
    key="events/1979.csv", filename="1979.csv")
2017-09-02 11:28:57,532 - Paws - INFO - Attempting download:
    gdelt-open-data, events/1979.csv, 1979.csv
```

После успешного запуска библиотеки можно заняться созданием переменной `PYTHONPATH` в сценарии `activate`, которая отражала бы расположение данной библиотеки:

```
#Export PYTHONPATH
PYTHONPATH="paws"
export PYTHONPATH
```

Далее мы снова запускаем виртуальную среду в текущем окружении с помощью настроенного ранее псевдонима `pawstop`:

```
(.pragia-aws) → pawstop
(.pragia-aws) → echo $PYTHONPATH
```

Теперь можно создать модульный тест с помощью двух удобных библиотек, предназначенных для тестирования AWS, — *pytest* и *moto*. *Moto* используется для имитационного моделирования AWS, а *pytest* представляет собой фреймворк тестирования. Увидеть это можно в листинге 2.3. Для создания временных ресурсов применяются фикстуры из библиотеки *pytest*, после чего с помощью *moto* создаются имитационные объекты, моделирующие действия *Boto*. Тестовая функция `test_download` далее контролирует, создан ли ресурс должным образом. Обратите внимание, что для настоящего тестирования функции `download` нужно передать в нее объект ресурса (листинг 2.3). Это отличный пример того, как тесты для кода повышают его надежность.

Листинг 2.3. Тестирование модуля S3

```
import pytest
import boto3
from moto import mock_s3
from paws.s3 import download

@pytest.yield_fixture(scope="session")
def mock_boto():
    """Настройка имитационных объектов"""

    mock_s3().start()
    output_str = 'Something'
    resource = boto3.resource('s3')
    resource.create_bucket(Bucket="gdelt-open-data")
    resource.Bucket("gdelt-open-data").\
        put_object(Key="events/1979.csv",
```

```

                                Body=output_str)
yield resource
mock_s3().stop()

def test_download(mock_boto):
    """Тестирование функции скачивания из хранилища S3"""

    resource = mock_boto
    res = download(resource=resource, bucket="gdelt-open-data",
                   key="events/1979.csv", filename="1979.csv")
    assert res == "1979.csv"
```

Для установки нужных для тестирования проекта библиотек содержимое файла `requirements.txt` должно выглядеть следующим образом:

```
awscli
boto3
moto
pytest
pylint
sensible
jupyter
pytest-cov
pandas
```

Для установки пакетов нужно выполнить команду `make install`. А для дальнейшего запуска тестов сборочный файл необходимо модифицировать так, чтобы он приобрел следующий вид:

```
setup:
    python3 -m venv ~/.pragia-aws

install:
    pip install -r requirements.txt

test:
    PYTHONPATH=. && pytest -vv --cov=paws tests/*.py

lint:
    pylint --disable=R,C paws
```

Далее запускаем тесты и получаем следующую информацию об охвате ими кода:

```
(.pragia-aws) → pragai-aws git:(master) X make test
PYTHONPATH=. && pytest -vv --cov=paws tests/*.py
```

```

=====
test session starts =====
platform darwin -- Python 3.6.2, pytest-3.2.1,
/Users/noahgift/.pragia-aws/bin/python3
cachedir: .cache
rootdir: /Users/noahgift/src/pragai-aws, inifile:
plugins: cov-2.5.1
collected 1 item

tests/test_s3.py::test_download PASSED

----- coverage: platform darwin, python 3.6.2-final-0
Name                Stmts  Miss  Cover
-----
paws/__init__.py      0      0  100%
paws/s3.py            12      2   83%
-----
TOTAL                  12      2   83%

```

Существует множество вариантов настройки Pylint, но я предпочитаю отображать только предупреждения и ошибки для проекта непрерывной поставки: `pylint --disable=R,C paws`. Результаты работы утилиты `lint` при этом выглядят следующим образом:

```

(.pragia-aws) → pragai-aws git:(master) ✗ make lint
pylint --disable=R,C paws
No config file found, using default configuration

```

```

-----
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)

```

Наконец, не помешает создать оператор `all` для установки, линтинга и тестирования: `all: install lint test`. После этого команда `make all` будет запускать все три действия последовательно.

Интеграция с блокнотом Jupiter

Возможность работы блокнота Jupiter с макетом проекта и автоматическое тестирование блокнотов могут принести немало пользы. Для этого мы создадим блокнот в каталоге `notebooks` и папку `data` в корневом каталоге извлеченного проекта: `mkdir -p notebooks`. Далее выполняем команду `jupyter notebook` и создаем новый блокнот с названием `paws.ipynb`. В этом блокноте можно воспользоваться вышеприведенной библиотекой для скачивания

CSV-файла и небольшого исследования данных с помощью библиотеки Pandas. Для начала мы добавляем в конец корневого пути .. и импортируем Pandas:

```
In [1]: #Добавляем в путь корневой каталог выгрузки проекта
...: import sys
...: sys.path.append("../")
...: import pandas as pd
...:
```

Загружаем созданную ранее библиотеку и скачиваем CSV-файл:

```
In [2]: from paws.s3 import (boto_s3_resource, download)
```

```
In [3]: resource = boto_s3_resource()
```

```
In [4]: csv_file = download(resource=resource,
...:                        bucket="gdelt-open-data",
...:                        key="events/1979.csv",
...:                        filename="1979.csv")
...:
```

```
2017-09-03 11:57:35,162 - Paws - INFO - Attempting
events/1979.csv, 1979.csv
```

В силу неправильной формы данных приходится прибегнуть к трюку для помещения их в используемый объект DataFrame: `names=range(5)`. К тому же из-за размера 100 Мбайт файл слишком велик для того, чтобы поместиться в репозиторий Git в качестве контрольного набора данных. Он будет усечен и снова сохранен:

```
In [7]: #Загружаем файл, выполняем его усечение и сохраняем
...: df = pd.read_csv(csv_file, names=range(5))
...: df = df.head(100)
...: df.to_csv(csv_file)
...:
```

Считываем файл обратно и генерируем по нему статистику:

```
In [8]: df = pd.read_csv("1979.csv", names=range(5))
...: df.head()
...:
```

```
Out[8]:
  Unnamed: 0
0          NaN
1          NaN
```

```

2      0.0  0\t19790101\t197901\t1979\t1979.0027\t\t\t\t\t...
3      1.0  1\t19790101\t197901\t1979\t1979.0027\t\t\t\t\t...
4      2.0  2\t19790101\t197901\t1979\t1979.0027\t\t\t\t\t...

```

```

      3      4
0      3      4
1      3      4
2  NaN  NaN
3  NaN  NaN
4  NaN  NaN

```

```
In [9]: df.describe()
```

```
Out[9]:
```

```

      Unnamed: 0
count  98.000000
mean   48.500000
std    28.434134
min     0.000000
25%    24.250000
50%    48.500000
75%    72.750000
max    97.000000

```

После того как этот простейший блокнот настроен, его можно интегрировать в систему сборки с помощью плагина `nbval` для `pytest` и добавления его в файл `requirements.txt`. Приведенные ниже строки нужно закомментировать (чтобы файл не скачивался из хранилища S3 при каждом запуске), затем блокнот можно сохранить и закрыть:

```

#csv_file = download(resource=resource,
#                    bucket="gdelt-open-data",
#                    key="events/1979.csv",
#                    filename="1979.csv")
#Загружаем файл, выполняем его усечение и сохраняем
#df = pd.read_csv(csv_file, names=range(5))
#df = df.head(100)
#df.to_csv(csv_file)

```

В сборочный файл можно добавить еще одну строку для тестирования блокнотов:

```
test:
```

```

PYTHONPATH=. && pytest -vv --cov=paws tests/*.py
PYTHONPATH=. && py.test --nbval-lax notebooks/*.ipynb

```

Результаты тестового запуска блокнота будут выглядеть следующим образом:

```
PYTHONPATH=. && py.test --nbval-lax notebooks/*.ipynb
=====
test session starts
=====
platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.34
rootdir: /Users/noahgift/src/pragai-aws, inifile:
plugins: cov-2.5.1, nbval-0.7
collected 8 items

notebooks/paws.ipynb .....

=====
warnings summary =====
notebooks/paws.ipynb::Cell 0
  /Users/noahgift/.pragia-aws/lib/python3.6/site-
packages/jupyter_client/connect.py:157: RuntimeWarning:
  '/var/folders/vl/sskrtrf17nz4nww5zr1b6498000gn/T':
  '/var/folders/vl/sskrtrf17nz4nww5zr1b6498000gn/T'
  RuntimeWarning,

-- Docs: http://doc.pytest.org/en/latest/warnings.html
=====
8 passed, 1 warnings in 4.08 seconds
```

Мы получили допускающую многократное применение и пригодную для тестирования структуру для добавления блокнотов в проект, а также для совместного использования созданной библиотеки. Кроме того, как мы увидим далее в этой главе, данную структуру можно использовать как среду непрерывной поставки. Благодаря тестированию блокнотов Jupiter по мере их сборки эта структура также служит и цели комплексного тестирования проекта машинного обучения.

Интеграция утилит командной строки

В проектах, связанных с разработкой как обычного ПО, так и алгоритмов машинного обучения, часто забывают про утилиты командной строки. Они гораздо удобнее при изучении данных в диалоговом режиме. При облачной архитектуре создать работающий в режиме командной строки

прототип, скажем, приложения на основе Amazon SQS часто можно намного быстрее, чем с помощью обычных методов, например IDE. Чтобы начать создание утилиты командной строки, необходимо внести в файл `requirements.txt` библиотеку Click и выполнить команду `make install`:

```
(.pragia-aws) → tail -n 2 requirements.txt
Click
```

Далее создаем сценарий командной строки в корневом каталоге:

```
(.pragia-aws) → pragai-aws git:(master) touch pcli.py
```

Этот сценарий выполняет набор действий, схожих с действиями, необходимыми для выполнения блокнота Jupiter, за исключением большей гибкости и возможности передачи пользователем аргументов функции из командной строки. В листинге 2.4 фреймворк Click используется для создания адаптера созданной ранее Python-библиотеки.

Листинг 2.4. Утилита командной строки pcli

```
#!/usr/bin/env python

"""
Утилита командной строки для работы с библиотекой PAWS
"""

import sys

import click
import paws
from paws import s3

@click.version_option(paws.__version__)
@click.group()
def cli():
    """Утилита для PAWS"""

    @cli.command("download")
    @click.option("--bucket", help="Name of S3 Bucket")
    @click.option("--key", help="Name of S3 Key")
    @click.option("--filename", help="Name of file")
    def download(bucket, key, filename):
        """Скачиваем файл из S3
        ./paws-cli.py --bucket gdelt-open-data --key \
```

```
events/1979.csv --filename 1979.csv
"""
if not bucket and not key and not filename:
    click.echo("--bucket and --key and --filename are required")
    sys.exit(1)
click.echo(
    "Downloading s3 file with: bucket-\
    {bucket},key{key},filename{filename}".\
    format(bucket=bucket, key=key, filename=filename))
res = s3.download(bucket, key,filename)
click.echo(res)

if __name__ == "__main__":
    cli()
```

Чтобы сделать этот сценарий исполняемым, необходимо добавить в его начало строку для Python с последовательностью символов #!:

```
#!/usr/bin/env python
```

Кроме того, нужно сделать файл исполняемым таким образом:

```
(.pragia-aws) → pragai-aws git:(master) chmod +x pcli.py
```

Наконец, не помешает создать в разделе `__init__.py` библиотеки переменную `__version__` и установить ее значение равным номеру версии в виде строки. Далее можно будет обращаться к ней в сценарии или утилите командной строки.

Чтобы этот сценарий вывел справочную информацию, необходимо выполнить следующие команды:

```
(.pragia-aws) → ./pcli.py --help
Usage: paws-cli.py [OPTIONS] COMMAND [ARGS]...
```

PAWS Tool

Options:

```
--version Show the version and exit.
--help    Show this message and exit.
```

Commands:

```
download Downloads an S3 file ./pcli.py --bucket...
```

Команды для скачивания файла с помощью этого сценария и результат их выполнения:

```
(.pragia-aws) → ./pcl_i.py download -bucket\
gdelt-open-data --key events/1979.csv \
--filename 1979.csv
```

```
Downloading s3 file with: bucket-gdelt-open-data,
keyevents/1979.csv,filename1979.csv
2017-09-03 14:55:39,627 - Paws - INFO - Attempting download:
gdelt-open-data, events/1979.csv, 1979.csv
1979.csv
```

Вот и все, что требуется для добавления в проект машинного обучения полнофункциональных утилит командной строки. Последний шаг – интеграция их в инфраструктуру тестирования. К счастью, в фреймворке Click отлично реализована поддержка тестирования (<http://click.pocoo.org/5/testing/>). С помощью команды `touch tests/test_paws_cli.py` создается новый файл. В листинге 2.5 приведен тестовый код, проверяющий передачу через утилиту командной строки переменной `__version__`.

Листинг 2.5. Тестирование утилиты `pcli` из командной строки с помощью фреймворка Click

```
import pytest
import click
from click.testing import CliRunner

from pcli import cli
from paws import __version__

@pytest.fixture
def runner():
    cli_runner = CliRunner()
    yield cli_runner

def test_cli(runner):
    result = runner.invoke(cli, ['--version'])
    assert __version__ in result.output
```

Чтобы в отчете об охвате тестами кода отображалась новая утилита командной строки, необходимо внести изменения в сборочный файл.

Они видны в нижеприведенных результатах выполнения команды сборки `make test`. Все работает, и охват вычисляется благодаря добавлению `--cov=pli`.

```
(.pragia-aws) → make test
PYTHONPATH=. && pytest -vv --cov=paws --cov=pli tests/*.py
=====
test session starts
=====
platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.34,
/Users/noahgift/.pragia-aws/bin/python3
cachedir: .cache
rootdir: /Users/noahgift/src/pragai-aws, inifile:
plugins: cov-2.5.1, nbval-0.7
collected 2 items

tests/test_paws_cli.py::test_cli PASSED
tests/test_s3.py::test_download PASSED

----- coverage: platform darwin, python 3.6.2-final-0
Name                Stmts  Miss  Cover
-----
paws/__init__.py      1      0  100%
paws/s3.py            12      2   83%
pcli.py               19      7   63%
-----
TOTAL                  32      9   72%
```

Интеграция AWS CodePipeline

Основной каркас работы над проектами AWS — создание, тестирование и сборка. Следующий шаг — интеграция в проект набора программных средств AWS CodePipeline. AWS CodePipeline (конвейер кода AWS) — обладающая очень большими возможностями коллекция утилит для AWS, своеобразный швейцарский нож в области непрерывной поставки. Ее можно расширить в множество сторон. В следующем примере мы зададим простые настройки сервера сборки для реагирования на изменения в GitHub. Сначала создается новый файл с помощью команды `touch buildspec.yml`, а затем этот файл распространяется теми же запускаемыми локально командами сборки, как показано в листинге 2.6.

Для этого мы создадим новый конвейер кода (CodePipeline) в консоли AWS (<https://us-west-2.console.aws.amazon.com/codepipeline/home?region=us-west-2#/create/Name>, пользователь должен быть авторизован для доступа к этой странице). Обратите внимание, что в файле `buildspec.yml` создается фиктивный набор учетных данных в используемом сервером сборки AWS CodeBuild контейнере Docker. Он предназначен для библиотеки `moto`, имитирующей вызовы Python `Boto`.

Листинг 2.6. Тестирование утилиты `pcii` из командной строки с помощью фреймворка `Click`

```
version: 0.2

phases:
  install:
    commands:
      - echo "Upgrade Pip and install packages"
      - pip install --upgrade pip
      - make install
      # Создаем каталог aws
      - mkdir -p ~/.aws/
      # Создаем файл с фиктивными учетными данными
      - echo "[default]\naws_access_key_id = \
        FakeKey\naws_secret_access_key = \
        FakeKey\naws_session_token = FakeKey" >\
        ~/.aws/credentials

  build:
    commands:
      - echo "Run lint and test"
      - make lint
      - PYTHONPATH=".";make test
  post_build:
    commands:
      - echo Build completed on `date`
```

Для того чтобы сборка заработала, необходимо пройти в консоли AWS CodePipeline несколько шагов. Сначала, как показано на рис. 2.1, указываем имя конвейера — `paws`.

На рис. 2.2 в качестве источника кода выбирается `GitHub`, а также указывается имя репозитория `GitHub` и выбирается ветка. В данном случае при каждом изменении ветки `master` инициируется обновление кода.

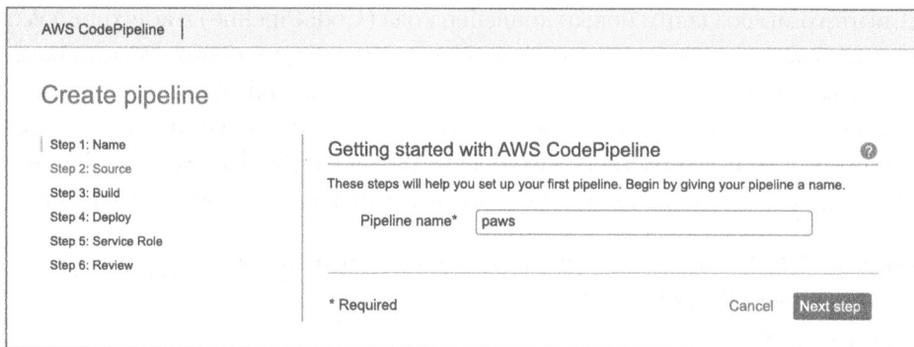


Рис. 2.1. Указываем имя CodePipeline (конвейера кода)

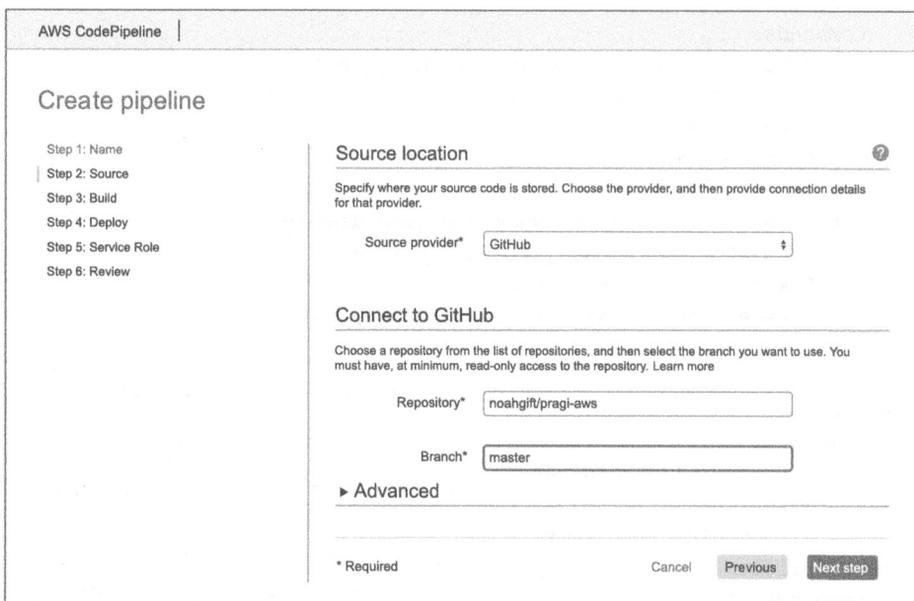


Рис. 2.2. Создаем CodePipeline Source (источник для конвейера кода)

Следующий шаг – самый насыщенный – шаг настроек сборки (рис. 2.3). Самое главное в нем – указание пользовательского образа Docker, демонстрирующего всю полноту возможностей CodePipeline. Кроме того, мы отмечаем на шаге сборки опцию поиска файла `buildspec.yml` в корневом каталоге репозитория GitHub. Это тот файл, который мы создали в листинге 2.5.

AWS CodePipeline |

Create pipeline

- Step 1: Name
- Step 2: Source
- Step 3: Build**
- Step 4: Deploy
- Step 5: Service Role
- Step 6: Review

Build

Choose the build provider that you want to use or that you are already using.

Build provider*

AWS CodeBuild

AWS CodeBuild is a fully managed build service that builds and tests code in the cloud. CodeBuild scales continuously. You only pay by the minute. [Learn more](#)

Configure your project

Select an existing build project
 Create a new build project

Project name*

Description [Remove description](#)

255 characters max

Environment: How to build

Environment image* Use an image managed by AWS CodeBuild
 Specify a Docker image

Custom image type*

Custom image ID

Build specification Use the buildspec.yml in the source code root directory
 Insert build commands

Рис. 2.3. Задаем настройки CodePipeline Build (настройки сборки для конвейера кода)

Рисунок 2.4 демонстрирует шаг развертывания, который мы настраивать не будем. На этом шаге можно выполнить развертывание проекта, скажем, с помощью сервиса Elastic Beanstalk.

На рис. 2.5 показана последняя экранная форма мастера создания конвейера, а на рис. 2.6 — результаты успешной сборки после срабатывания триггера обновления GitHub. На этом основные настройки CodePipeline для проекта завершены, но здесь можно делать еще очень многое. Можно запускать срабатывание лямбда-функций, отправлять SNS-сообщения, запускать несколько одновременных сборок для тестирования, например нескольких версий базы кода для различных версий Python.

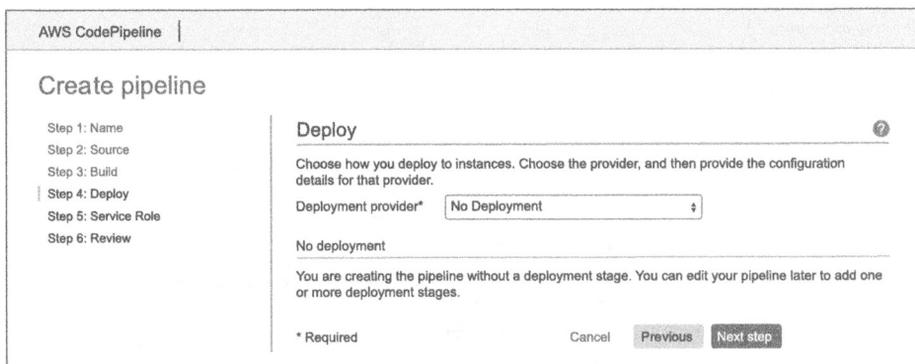


Рис. 2.4. Задаем настройки CodePipeline Deploy (настройки развертывания для конвейера кода)

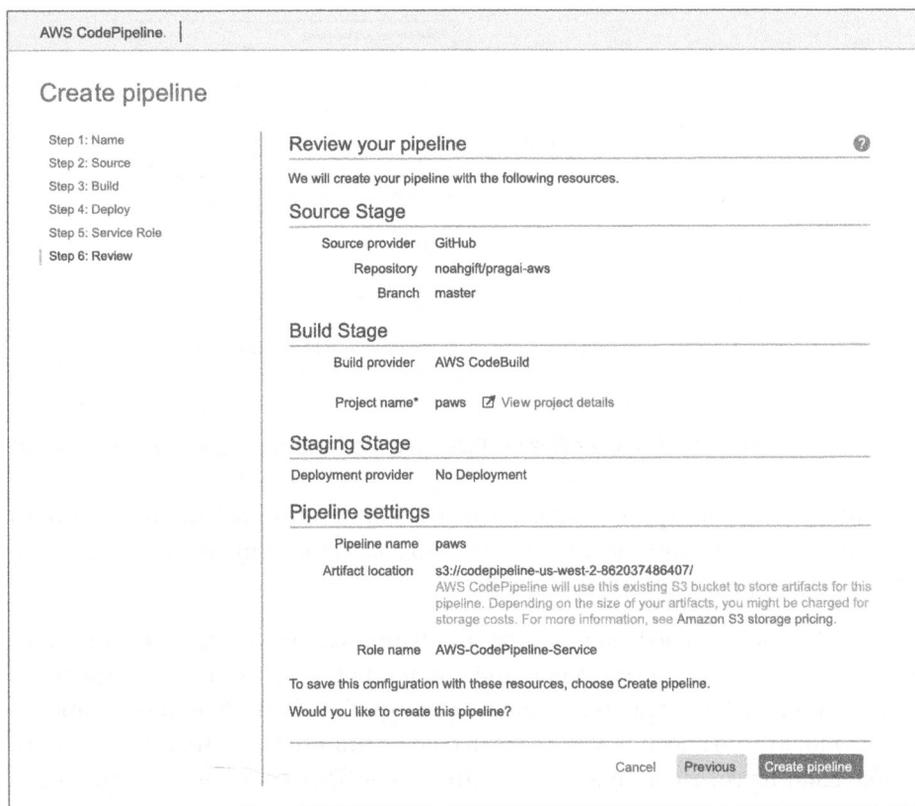


Рис. 2.5. CodePipeline Review (обзор настроек для конвейера кода)

The screenshot shows the AWS CodeBuild console interface. On the left, there are navigation links for 'Build projects' and 'Build history'. The main content area displays the details for a specific build with ID 'paws:8fed00e0-da6d-4254-9539-6a7b0b11bf6d', which has a 'Succeeded' status. Below this, a 'Build' section lists key metadata such as the Build ARN, Project Name, Source Provider (AWS CodePipeline), Repository, Start time (44 minutes ago), End time (43 minutes ago), Status (Succeeded), and Initiator (paws). A 'Phase details' table follows, listing 12 phases from SUBMITTED to COMPLETED, all with a 'Succeeded' status and completion times ranging from 43 to 44 minutes ago. At the bottom, there is a 'Build logs' section with a note that the last 20 lines are shown.

Name	Status	Duration	Completed
▶ SUBMITTED	Succeeded		44 minutes ago
▶ PROVISIONING	Succeeded	7 secs	44 minutes ago
▶ DOWNLOAD_SOURCE	Succeeded	4 secs	44 minutes ago
▶ INSTALL	Succeeded	42 secs	43 minutes ago
▶ PRE_BUILD	Succeeded		43 minutes ago
▶ BUILD	Succeeded	7 secs	43 minutes ago
▶ POST_BUILD	Succeeded		43 minutes ago
▶ UPLOAD_ARTIFACTS	Succeeded		43 minutes ago
▶ FINALIZING	Succeeded	5 secs	43 minutes ago
▶ COMPLETED	Succeeded		

Рис. 2.6. Успешная сборка для конвейера кода

Основные настройки Docker для исследования данных

Студенты, которые ранее не сталкивались с наукой о данных, все время задают мне вопросы по поводу того, что их среда перестала работать. Эта серьезная проблема постепенно уходит в прошлое благодаря таким утилитам, как Docker. Для пользователей компьютеров Apple Mac хотя бы есть определенное сходство среды Unix на Mac и среды промышленной эксплуатации Linux, но Windows отличается коренным образом. Именно поэтому для исследователей данных, которые используют Windows, Docker оказывается очень полезной утилитой с большими возможностями.

Чтобы узнать, как установить Docker в операционных системах OS X, Linux или Windows, можно заглянуть в документацию на сайте <https://www.docker.com/>. Отличное место, чтобы поэкспериментировать со стеком науки о данных, — блокнот Jupiter для науки о данных по адресу <https://hub.docker.com/r/jupyter/>

`datascience-notebook/`. После установки Docker и выполнения команды `docker pull` можно запустить блокнот командой, состоящей из одной строки:

```
docker run -it --rm -p 8888:8888 jupyter/datascience-notebook
```

Для запуска пакетных заданий с помощью AWS Batch команда промышленной эксплуатации может работать с Dockerfile на ноутбуке команды разработчиков, загружая его в исходный код с последующей регистрацией в приватном реестре Docker для AWS. Тогда при последующем запуске пакетных заданий они будут выполняться точно так же, как и на ноутбуке команды разработчиков. Это поистине завтрашний день разработки, и прагматично настроенным командам разработчиков рекомендуется разобраться с Docker. Он экономит массу времени при локальной разработке, и для многих реальных задач с запуском заданий использовать Docker не просто желательно, а обязательно.

Другие сервисы сборки: Jenkins, CircleCI и Travis

Рассмотренный нами в этой главе CodePipeline — программный продукт, предназначенный конкретно для AWS. Есть и другие отличные сервисы, в том числе Jenkins (<https://jenkins.io/>), CircleCI (<https://circleci.com/>) и Travis (<https://travis-ci.org/>). У каждого из них свои слабые и сильные стороны, но в целом они все представляют собой основанные на Docker системы сборки. Это еще один довод в пользу того, чтобы положить Docker в основу своей системы.

Возможно, вас также вдохновит созданный мной пробный проект, осуществляющий сборку с помощью CircleCI: <https://github.com/noahgift/myrepo>. Там вы найдете также учебное видео с семинара, в котором я шаг за шагом рассматривал все нюансы.

Резюме

В этой главе мы охватили основы интеграции разработки и эксплуатации применительно к машинному обучению. Мы разработали примеры конвейера непрерывной поставки, пригодные в качестве стандартных блоков конвейера машинного обучения, и создали структуру проекта. Ведь вы

можете запросто потратить месяцы работы над проектом и в итоге только осознать, что отсутствие базовой инфраструктуры — реальная угроза его жизнеспособности в долгосрочной перспективе.

Наконец, мы подробно рассмотрели Docker, поскольку, скажем прямо, за ним будущее. Любая команда исследователей данных должна знать о нем. Это совершенно необходимая часть развертываемого решения для каждой действительно серьезной задачи вроде создания крупномасштабной системы ИИ для промышленной эксплуатации.

3

Спартанский жизненный цикл ИИ

Я собираюсь завтра выйти на беговую дорожку, пусть даже окажусь смешон, и посмотрим, что получится. Мне нечего терять.

*Уэйд ван Никерк
(Wayde van Niekerk)*

Многое при создании программного обеспечения можно счесть несущественным, но впоследствии оно окажется критически важным. И наоборот, то, что видится важным, может оказаться несущественным. Я открыл для себя полезный эвристический метод: мыслить категориями петель обратной связи. При конкретной задаче ускоряет ли то, что вы делаете, петлю обратной связи или блокирует ее?

Спартанский жизненный цикл ИИ — дух и буква этого эвристического метода. Или вы добиваетесь ускорения и повышения эффективности, или нет. Оптимизация внутреннего кольца петли обратной связи, например улучшение модульных тестов или повышение надежности ETL-сервисов (извлечение, преобразования и загрузки, *extract, transform and load*), должна производиться таким образом, чтобы не блокировать другие петли обратной связи.

В этой главе подобный стиль мышления рассматривается на примерах реальных проблем, допустим: создавать свою собственную модель машинного обучения или вызывать API; быть зависимым от поставщика ПО или создать его своими силами. Мыслить как воин-спартанец в сфере ИИ означает не тратить усилия на то, что не повышает эффективность системы в целом и каждой ее подсистемы в отдельности. Наличие эвристики, которой

можно руководствоваться, очень помогает и придает сил. В конце концов, есть вещи более важные и менее важные, другими словами, подход должен быть прагматичным.

Прагматическая петля обратной связи при промышленной эксплуатации

Петля обратной связи технологии на рис. 3.1 описывает ход мыслей при создании последней. В любой технологии, вне зависимости от того, касается она создания моделей машинного обучения или веб-приложений, есть петля обратной связи, которую следует учитывать. Если петля обратной связи медленная, нарушена или чем-то блокируется, за это придется расплачиваться. В данной главе приведено несколько примеров нарушенных петель обратной связи, встречавшихся мне в местах, где я работал.

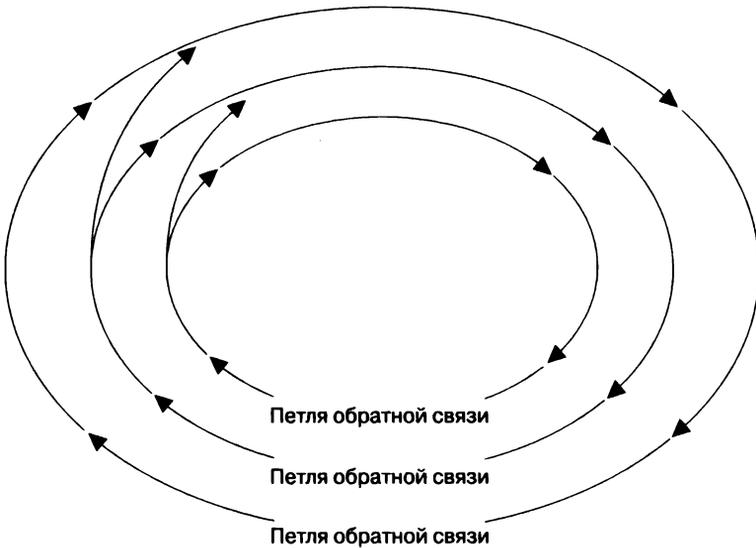


Рис. 3.1. Петля обратной связи технологии

Разработчики, редко фиксирующие в репозитории изменения кода, представляют собой распространенную проблему стартапов и других компаний. Коэффициент, который вычисляется в моем проекте с открытым

исходным кодом `devml` и именуется коэффициентом активности (*active ratio*), отражает среднюю частоту внесения разработчиками кода в репозиторий. Например, есть разработчики, которые делают это в среднем четыре раза за пять рабочих дней недели; есть те, которые вносят код раз в неделю; и в исключительных случаях изменения фиксируются только раз в три месяца.

Разработчики и менеджеры часто пафосно — и ошибочно — говорят о вводящих в заблуждение метриках исходного кода, например о том, что число строк в день ни о чем не говорит. Но то, что рассуждать о метриках исходного кода сложно, отнюдь не значит, что эта информация неважна. Коэффициент активности описывает поведенческие аспекты исходного кода, гораздо более интересные, чем данные о популярности. Хороший пример этого можно найти в реальном мире. Что может значить, если маляр взял субподряд на покраску дома и красил только 25 % отведенного времени? Возможны несколько вариантов, наиболее очевидны такие: маляр выполнил свою работу некачественно или же он хочет обмануть клиента. Менее очевидный, но вполне возможный вариант: проект организован из рук вон плохо. Что если основной подрядчик все время блокирует дорогу к дому бетономешалками и маляр в течение нескольких дней не может попасть на свое рабочее место? Это пример испорченной петли обратной связи.

Разработчиков ПО можно нещадно эксплуатировать, заставляя работать в нечеловеческих условиях по 12 ч в день семь дней в неделю, но при должном реагировании на поведенческие сигналы можно разумно оптимизировать производительность и помочь командам разработчиков работать более результативно. В идеале маляр красит дом с понедельника по пятницу, а команда разработчиков ПО пишет код с понедельника по пятницу. В случае программного обеспечения вопрос еще и в «креативности», отличающей эту сферу деятельности от сферы физического труда, которую также надо учитывать; работа на износ может привести к понижению «креативности», а значит, и качества ПО. Именно поэтому я не советую никому ни смотреть на коэффициент активности и поощрять семидневную рабочую неделю для программистов с ежедневными фиксациями в репозитории изменений кода, ни поощрять 400-дневные «марафоны» с ежедневными фиксациями кода на GitHub.

Вместо этого поведенческие метрики, относящиеся к петлям обратной связи, должны поощрять опытные команды разработчиков присматриваться к тому, что происходит на самом деле. Например, к неэффективному

управлению проектом в команде разработчиков ПО, которая фиксирует код в репозитории лишь пару дней в неделю. Я видел организации, где скрам-мастер или основатель компании вытягивал разработчиков на собрания, которые длились целый день или проходили несколько раз в день, лишь бессмысленно тратя их время. На самом деле, если такую практику не прекратить, компания неизбежно обанкротится. Подобные ситуации вполне можно заметить, если присмотреться к поведенческим метрикам, относящимся к петлям обратной связи.

Еще одна петля обратной связи — петля обратной связи науки о данных. Как компании на деле решают задачи, связанные с данными, и что такое здесь петля обратной связи? По иронии судьбы, многие команды разработчиков фактически ничего не дают компании, поскольку петля обратной связи нарушена. Каким образом нарушена? Чаще всего ее просто нет. Вот несколько вопросов, которые стоит себе задать.

- ❑ Может ли команда выполнять, как ей заблагорассудится, эксперименты на находящейся в промышленной эксплуатации системе?
- ❑ Выполняет ли команда эксперименты в блокнотах Jupiter и Pandas только на маленьких, нереальных наборах данных?
- ❑ С какой частотой команда исследователей данных отправляет обученные модели в промышленную эксплуатацию?
- ❑ Компания не хочет использовать заранее обученные модели машинного обучения, такие как API Google Cloud Vision или Amazon Rekognition, или вообще не знает об их существовании?

Один из секретов Полишинеля относительно машинного обучения заключается в том, что обучаемые инструменты — Pandas, scikit-learn, DataFrames языка R и т. п. — сами по себе плохо работают в технологических процессах при промышленной эксплуатации. Для создания прагматичных решений необходим сбалансированный подход с использованием инструментов для работы с большими данными, таких как PySpark, и проприетарных инструментов, например AWS SageMaker, Google TPU (тензорных процессоров, предназначенных для использования с библиотекой TensorFlow).

Прекрасный пример нарушенной петли обратной связи — база данных SQL для промышленной эксплуатации, разработанная без учета архитектуры машинного обучения. Разработчики радостно сдали написанный код, а связанные с инженерией данных задачи вынуждены всецело полагаться на

доморощенные Python-сценарии, служащие для преобразования таблиц базы данных SQL во что-нибудь полезное (например, в файлы в формате CSV), анализируемое затем с помощью Pandas и scikit-learn по мере надобности. Как менеджеру, мне приходилось встречать множество исследователей данных, которых вообще не волновало попадание модели машинного обучения в промышленную эксплуатацию, хотя должно было бы, в качестве завершения петли обратной связи. В этой сфере промышленности зачастую недостает прагматизма, и одно из решений данной проблемы заключается в сосредоточении усилий на наличии петли обратной связи машинного обучения, связанной с попаданием кода в промышленную эксплуатацию.

На протяжении 2017 и 2018 гг. инструменты для завершения этой петли обратной связи развивались с невероятной скоростью. Именно для этого служат несколько инструментов, упомянутых в данной главе, — например, AWS SageMaker, нацеленный на быстрое создание и обучение моделей машинного обучения и развертывание их на конечных точках API. Еще один инструмент AWS, AWS Glue, служит для управления ETL-процессами посредством подключения источников данных, например баз данных SQL, выполнения над ними операций ETL и записи обратно в другие источники данных, например в S3 или другую базу данных SQL.

У компании Google также есть набор аналогичных инструментов, включая утилиту BigQuery — одну из жемчужин машинного обучения в том смысле, что она способна работать под практически любой рабочей нагрузкой. Другие части экосистемы Google тоже обеспечивают эффективную петлю обратной связи машинного обучения/ИИ, например, тензорные процессоры (<https://cloud.google.com/tpu/>), Datalab (<https://cloud.google.com/datalab/>) и различные API ИИ.

Говорят, что данные — новая нефть; если придерживаться этой аналогии, то нельзя же использовать нефть в двигателе внутреннего сгорания, для этого необходима петля обратной связи для бензина. Так что приходится начинать с поиска и добычи нефти с помощью аппаратуры промышленного класса (аналогично тому, что предоставляют провайдеры облачных сервисов) с последующей ее транспортировкой и очисткой перед отправкой на заправочные станции. Представьте себе ситуацию, когда группа инженеров бурит скважины в земле и пытается в доморощенной лаборатории очистить нефть, грузит бидоны с горючим в грузовики, которые как-то добрались до буровой площадки. Примерно так наука о данных и используется во множестве компаний, и именно поэтому ситуация быстро меняется.

Для многих организаций осознание этой непростой задачи и возможностей, которые сулит ее решение, требует выбора одной из нескольких дорог. Существующее положение дел, когда с данными работают кое-как, на скорую руку, решению этой задачи не способствует, лабораторию необходимо превратить в нефтеперегонный завод, способный выдавать при промышленной эксплуатации высококачественный продукт в больших объемах. Изготовление вышеупомянутых бидонов с высокооктановым горючим на месте бурения приемлемо на практике ровно так же, как исследование данных в вакууме.

AWS SageMaker

SageMaker — великолепный технологический продукт, созданный Amazon и нацеленный на решение упомянутой ранее в этой главе задачи. Он завершает на практике одну из петель обратной связи для машинного обучения. Этот процесс показан на рис. 3.2: сначала выполняется разведочный анализ данных и/или обучение модели. Далее запускается задание, модель обучается, после чего развертывается конечная точка... возможно, для промышленной эксплуатации.

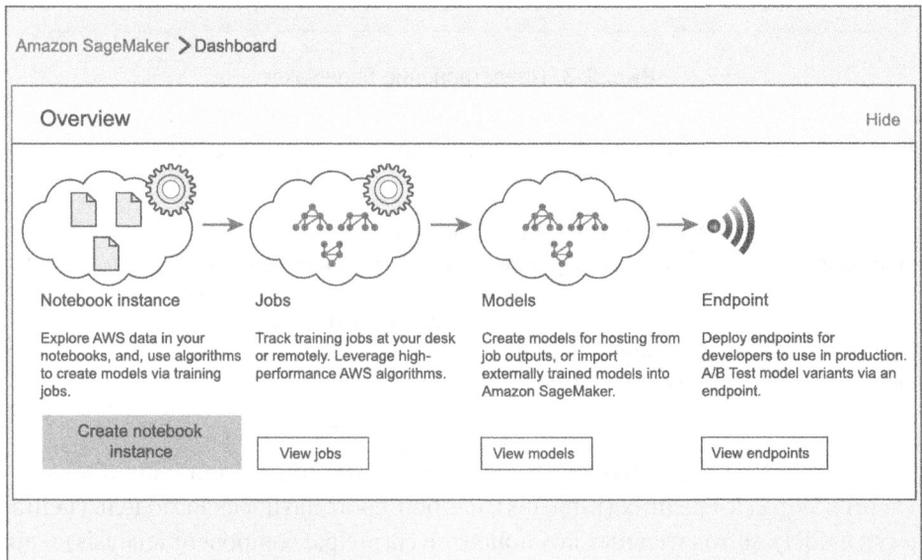


Рис. 3.2. Петля обратной связи SageMaker

Наконец, особенно расширяет возможности SageMaker использование Boto, что можно с легкостью сделать с помощью фреймворка Chalice, или непосредственно AWS Lambda, или даже самой конечной точки, как показано на рис. 3.3.



Рис. 3.3. Представление SageMaker

На практике использование Boto выглядит следующим образом:

```
import boto3
sm_client = boto3.client('runtime.sagemaker')
response = sm_client.invoke_endpoint(EndpointName=endpoint_name,
                                     ContentType='text/x-libsvm',
                                     Body=payload)

result = response['Body'].read()
result = result.decode("utf-8")
print(result)
```

В SageMaker поддерживаются также встроенные модели для машинного обучения: метод k-средних (k-means), нейронная тематическая модель (neural topic model), метод главных компонент (principal component analysis) и др. Алгоритмы SageMaker упакованы в виде образов Docker, и вы можете использовать практически любой из них. Сильная сторона этого подхода: при

использовании реализации метода k-средних из SageMaker вы получаете разумные гарантии производительности и совместимости при генерации повторяемых технологических процессов для промышленной эксплуатации. В то же время, однако, существует возможность генерации хорошо оптимизированных пользовательских алгоритмов в виде повторяемых сборок Docker.

Петля обратной связи AWS Glue

AWS Glue — отличный пример петли обратной связи внутри основной петли обратной связи. В настоящее время наблюдается кризис, связанный с устаревшими базами данных SQL и NoSQL, которые сканируются с помощью плохих сценариев. AWS Glue серьезно продвинулся в решении данной проблемы. Это полностью управляемый ETL-сервис, смягчающий в немалой степени типичные неприятности выполнения ETL.

Рисунок 3.4 демонстрирует, как это все работает.

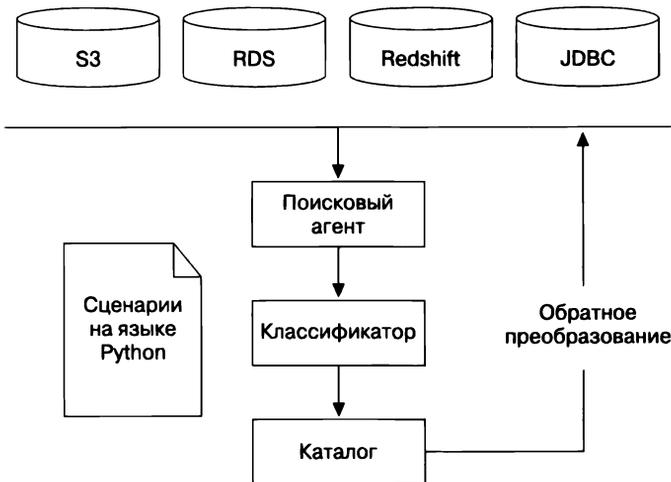


Рис. 3.4. Конвейер ETL AWS Glue

Вот один из примеров работы AWS Glue. Допустим, у вас есть устаревшая база данных PostgreSQL, в которой хранятся данные о пользователях вашего стартапа. Достаточно подключить AWS Glue к этой базе данных, и он определит ее схему (рис. 3.5).

Схема			
	Имя столбца	Тип данных	Ключ
1	updated_at	timestamp	
2	name	string	
3	created_at	timestamp	
4	id	int	
5	locale	string	

Рис. 3.5. Представление AWS Glue

Далее создается задание — сценарий на языке Python или Scala, переводящий данную схему в другой формат и меняющий место назначения. Выглядят эти сценарии примерно следующим образом (сокращено ради экономии места). Вы можете или воспользоваться данными сценариями, хранящимися в S3 по умолчанию, или подправить их.

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
## @params: [JOB_NAME]
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
####сокращено
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
####сокращено
## @inputs: [frame = applymapping1]
datasink2 = glueContext.write_dynamic_frame.\
    from_options(frame = applymapping1,
        connection_type = "s3",
        connection_options = \
            {"path": "s3://dev-spot-etl-pg/tables/scrummaster"},
        format = "csv", transformation_ctx = "datasink2")
job.commit()
```

После этого можно запланировать выполнение данных заданий при возникновении определенных событий или по заданию планировщика. Конечно, подобное можно сделать и посредством Python с помощью Boto. Самое приятное в данном сервисе — непрерывность бизнес-процессов. Если разработчик увольняется или его увольняют, следующий разработчик сможет без проблем сопровождать сервис. Это пример надежной петли обратной связи, не зависящей от личных качеств конкретного ключевого работника.

AWS Glue также хорошо подходит для более крупных конвейеров обработки данных. Помимо подключения к реляционным базам данных, AWS Glue может выполнять ETL-обработку хранимых в S3 данных. Один из потенциальных источников последних — сервис Amazon Kinesis, который можно использовать для дампа потока данных в корзину S3. Вот пример конвейера с отправкой в S3 событий Firehose. Сначала создаются соединение с клиентом Firehose Boto3 и событие библиотеки asyncio:

```
import asyncio
import time
import datetime
import uuid
import boto3
import json

LOG = get_logger(__name__)

def firehose_client(region_name="us-east-1"):
    """Клиент Kinesis Firehose"""

    firehose_conn = boto3.client("firehose", region_name=region_name)
    extra_msg = {"region_name": region_name, \
                "aws_service": "firehose"}
    LOG.info("firehose connection initiated", extra=extra_msg)
    return firehose_conn

async def put_record(data,
                    client,
                    delivery_stream_name="test-firehose-nomad-no-lambda"):
    """
    См.:
```

```

    http://boto3.readthedocs.io/en/latest/reference/services/
    firehose.html#Firehose.Client.put_record
    """
    extra_msg = {"aws_service": "firehose"}
    LOG.info(f"Pushing record to firehose: {data}", extra=extra_msg)
    response = client.put_record(
        DeliveryStreamName=delivery_stream_name,
        Record={
            'Data': data
        }
    )
    return response

```

Далее для использования в отправляемых в асинхронном потоке событиях создается уникальный идентификатор пользователя (UUID):

```

def gen_uuid_events():
    """Создает событие на основе UUID с меткой даты/времени"""

    current_time = 'test-{'date:%Y-%m-%d %H:%M:%S'}'.\
format(date=datetime.datetime.now())
    event_id = str(uuid.uuid4())
    event = {'event_id':current_time}
    return json.dumps(event)

```

Наконец, цикл ожидания асинхронных событий отправляет эти сообщения в сервис Kinesis, который постепенно помещает их в S3 для преобразования с помощью AWS Glue. Для завершения этого цикла необходимо далее подключить поисковый агент (crawler) Glue для S3, как показано на рис. 3.6, который после этого изучит схему и создаст таблицы, передаваемые затем выполняемым заданиям ETL.

```

def send_async_firehose_events(count=100):
    """Отправка асинхронных событий в Firehose"""

    start = time.time()
    client = firehose_client()
    extra_msg = {"aws_service": "firehose"}
    loop = asyncio.get_event_loop()
    tasks = []
    LOG.info(f"sending async events TOTAL {count}", extra=extra_msg)
    num = 0
    for _ in range(count):
        tasks.append(asyncio.ensure_future(

```

```

        put_record(gen_uuid_events(), client)))
    LOG.info(f"sending aysnc events: COUNT {num}/{count}")
    num +=1
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
end = time.time()
LOG.info("Total time: {}".format(end - start))

```

The screenshot shows the 'Add crawler' wizard in the AWS Glue console. The 'Add a data store' step is selected. The 'Data store' dropdown is set to 'S3'. Under 'Crawl data in', the 'Specified path in my account' radio button is selected. The 'Include path' text box contains 's3://bucket-name/folder-name/file-name'. Below the text box, there is a note: 'All folders and files contained in the include path are crawled. For example, type s3://MyBucket/MyFolder/ to crawl all objects in MyFolder within MyBucket.' and a link for 'Exclude patterns (optional)'. At the bottom, there are 'Back' and 'Next' buttons.

Рис. 3.6. Поискный агент AWS Glue для S3

AWS Batch

Модуль пакетной обработки AWS (AWS Batch) — еще один сервис, освобождающий компании и команды исследователей данных от написания бессмысленного кода. Необходимость запуска пакетных заданий, выполняющих, например, кластеризацию методом *k*-средних или предварительную часть обработки в конвейере данных, возникает очень часто. Опять же нередко это представляет собой нарушенную петлю обратной связи, балансирующую на грани краха в случае увольнения ключевых сотрудников.

Пример конвейера пакетной обработки AWS, приведенный на рис. 3.7, демонстрирует, как из отдельных заранее собранных сервисов можно создать очень надежный сервис на основе «готового механизма классификации изображений» Amazon Rekognition и «готовых» утилит AWS для обработки событий и пакетов.

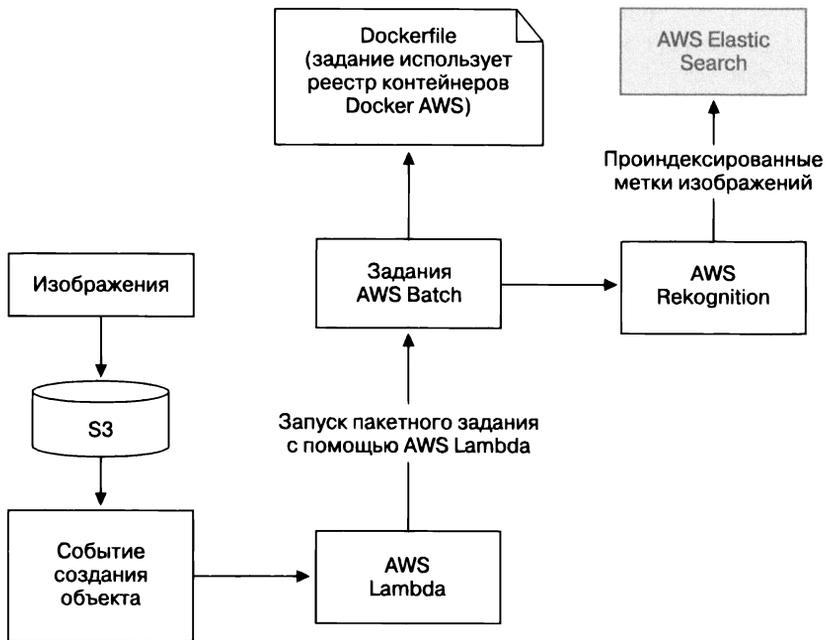


Рис. 3.7. Конвейер машинного обучения для классификации изображений на основе AWS Batch

Как и другие части этого конвейера, AWS Batch можно вызвать с помощью Python и Boto; в случае AWS Lambda можно воспользоваться фреймворком AWS Chalice. AWS Batch решает весьма серьезную задачу. Для нее ранее требовался или исключительно запутанный «доморощенный» код, или сочетание низкоуровневых сервисов AWS, таких как сервис запросов Simple Queue Service (SQS) и сервис оповещений Simple Notification Service (SNS), которые обладают большими возможностями, но могут также привести к запутанной архитектуре.

```

def batch_client():
    """Создаем клиент AWS Batch
    {"message": "Create AWS Batch Connection"}
    {"message": "Found credentials in shared credentials file:
    ~/.aws/credentials"}
    """

    log.info(f"Create AWS Batch Connection")
    client = boto3.client("batch")
    
```

```

return client

def submit_job(job_name="1", job_queue="first-run-job-queue",
              job_definition="Rekognition",
              command="uname -a"):
    """Передаем задание AWS Batch на выполнение"""

    client = batch_client()
    extra_data = {"jobName":job_name,
                  "jobQueue":job_queue,
                  "jobDefinition":job_definition,
                  "command":command}
    log.info("Submitting AWS Batch Job", extra=extra_data)
    submit_job_response = client.submit_job(
        jobName=job_name,
        jobQueue=job_queue,
        jobDefinition=job_definition,
        containerOverrides={'command': command}
    )
    log.info(f"Job Response: {submit_job_response}",
            extra=extra_data)
    return submit_job_response

```

Петли обратной связи на основе Docker

В основе множества описанных в данной книге методик лежат файлы Docker. Это миниатюрная петля обратной связи с исключительными возможностями. В случае как AWS, так и облачной платформы Google (GCP) можно компоновать свои собственные контейнеры Docker и выдавать их из соответствующих сервисов-реестров. До недавних пор Docker был не слишком надежен, но сегодня это, несомненно, революционная технология.

Есть множество причин, чтобы использовать Docker для машинного обучения. Начинать с самого низкого уровня и мучиться с настоящим компоновочным адом на своей машине — пустая трата времени, если можно взять файл, вызвать его в чистой песочнице и получить чистую среду.

Зачем теряться в командах `pip install` и утилитах управления пакетами `conda`, конфликтующих друг с другом, когда можно объявить `Dockerfile` с той средой, которая точно будет работать как в промышленной эксплуатации, так и на машинах с операционными системами OS X, Linux и Windows?

Кроме того, взгляните на следующий пример Dockerfile, предназначенного для тестирования базирующегося на AWS Lambda приложения. Он и достаточно короткий (поскольку собирается на основе Amazon Dockerfile для Linux), и наглядный:

```
FROM amazonlinux:2017.09

RUN yum -y install python36 python36-devel gcc \
    procps libcurl-devel mod_nss crypto-utils \
    unzip

RUN python3 --version

# Создаем каталог app и добавляем приложение
ENV APP_HOME /app
ENV APP_SRC $APP_HOME/src
RUN mkdir "$APP_HOME"
RUN mkdir -p /artifacts/lambda
RUN python3 -m venv --without-pip ~/.env && \
    curl https://bootstrap.pypa.io/get-pip.py | \
    ~/.env/bin/python3

#Копируем все файлы требований
COPY requirements-testing.txt requirements.txt ./

#Устанавливаем оба [этих файла] с помощью pip
RUN source ~/.env/bin/activate && \
    pip install --install-option="--with-nss" pycurl && \
    pip install -r requirements-testing.txt && \
    source ~/.env/bin/activate && \
    pip install -r requirements.txt
COPY . $APP_HOME
```

Для интеграции с реестром контейнеров AWS необходимо авторизоваться в AWS:

```
AWS_PROFILE=metamachine
AWS_DEFAULT_REGION=us-east-1
export AWS_PROFILE
export AWS_DEFAULT_REGION
```

```
aws ecr get-login --no-include-email --region us-east
```

Теперь собираем образ локально:

```
docker build -t metamachine/lambda-tester .
```

Создаем для него тег:

```
docker tag metamachine/lambda-tester:latest\
  907136348507.dkr.ecr.us-east-1.amazonaws.com\
  /metamachine/myorg/name:latest
```

Отправляем в реестр AWS:

```
docker push 907136348507.dkr.ecr.us-east-1.amazonaws.com\
  /metamachine/lambda-tester:latest
```

Теперь другие сотрудники компании могут запустить этот образ, скачав его на локальную машину:

```
docker pull 907136348507.dkr.ecr.us-east-1.amazonaws.com\
  /metamachine/lambda-tester:latest
```

Дальнейший запуск образа не вызывает сложностей. Вот пример запуска образа и монтирования локальной файловой системы:

```
docker run -i -t -v `pwd`:/project 907136348507.\
  dkr.ecr.us-east-1.amazonaws.com/ \
  metamachine/lambda-tester /bin/bash
```

Это лишь один из примеров использования Docker, но, по сути, любой сервис в данной книге может как-либо взаимодействовать с Docker — от запуска пакетных заданий и до выполнения технологических процессов исследования данных на основе пользовательских блокнотов Jupiter.

Резюме

Петли обратной связи очень важны для преодоления в компаниях «лабораторного» менталитета в сфере науки о данных. Возможно, «наука о данных» — не лучший термин для описания решения в компании задач машинного обучения. Прагматичное решение задач ИИ требует уделять больше внимания достижению результата, чем методике. В конце концов, тратить месяцы на выбор оптимального алгоритма машинного обучения для

чего-то, что никогда не попадет в промышленную эксплуатацию, — яркий пример бессмысленности и выбрасывания денег на ветер.

Один из способов довести больше алгоритмов машинного обучения до промышленной эксплуатации — перестать работать на износ. Для этого прекрасно подойдут готовые решения от провайдеров облачных сервисов. Переход от «героической разработки» к стилю работы, поощряющему непрерывность бизнес-процессов и отправку решений в промышленную эксплуатацию, несет благо всем: отдельным разработчикам, бизнесу в целом и глобальному развитию ИИ.

Часть II

ИИ в облаке

4

Разработка ИИ в облачной среде с помощью облачной платформы Google

Не существует простых методов
«построения команды» каждый сезон.
Фундамент строится по кирпичику.

*Билл Беличик
(Bill Belichick)*

Как для разработчиков, так и для исследователей данных в GCP есть много приятного. Большое количество сервисов GCP нацелено на упрощение разработки и расширение ее возможностей. В известной мере Google давно был лидером в отдельных аспектах облачных сервисов, которые AWS лишь недавно стал воспринимать всерьез. Намного опережала свое время, в частности, служба хостинга App Engine, выпущенная в 2008 г. на основе модели PaaS («платформа как сервис») с доступом к облачному хранилищу Google (Google Cloud Datastore) — сервису базы данных NoSQL с широкими возможностями управления.

С другой стороны, Amazon существенно обогнал конкурентов в своей приоритетной нацеленности на непосредственное удовлетворение потребностей пользователей. Один из основополагающих принципов Amazon, наряду с экономичностью, — максимальная клиентоориентированность. Это сочетание привело к лавине дешевых облачных сервисов и возможностей, с которыми Google поначалу не хотел или не мог соревноваться. Многие годы невозможно было даже найти телефон любого сервиса Google, и их первые новаторские решения, вроде Google App Engine, были заброшены

и едва не завяли на корню. Основным источником доходов Google всегда была реклама, а в Amazon — товары. В результате AWS лидирует на мировом рынке облачных сервисов с подавляющим отрывом, удерживая от 30 до 35 % рынка.

Взрыв интереса к ИИ и «прикладным» большим данным дал шанс Google Cloud, которым Google немедленно воспользовался. Google, вероятно, был компанией машинного обучения и больших данных с самого начала, что дало ему в руки оружие для атаки на лидера рынка облачных сервисов, AWS. Хотя по некоторым из их облачных предложений Google отставал, зато по другим, в сфере машинного обучения и ИИ, был впереди. Возникла новая арена борьбы — облачные сервисы машинного обучения и ИИ, и GCP готова всерьез на ней сражаться.

Обзор GCP

Google заработал в 2017 г. около \$74 млрд чистыми на продажах цифровой рекламы, но лишь \$4 млрд — от облачных сервисов. Положительная сторона этой диспропорции — в наличии больших денежных ресурсов на финансирование исследований и разработку инноваций в сфере облачных сервисов. Хороший пример — TPU, превышающие современные GPU и CPU по производительности в 15–30 раз (<https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu>).

Помимо создания специализированных микросхем для ИИ, Google также занимался разработкой полезных сервисов ИИ с заранее обученными моделями, например Cloud Vision API (облачный API компьютерного зрения), Cloud Speech API (облачный API распознавания речи) и Cloud Translation API (облачный API машинного перевода). Одна из тем, которой посвящена данная книга, — принцип прагматизма. В Области залива Сан-Франциско нет недостатка в компаниях, битком набитых инженерами и исследователями данных, работающими над совершенно бесполезными вещами. Они усердно, до потери пульса трудятся над блокнотами Jupiter, которые, возможно, никогда не принесут их компании никакого реального дохода.

Это ничуть не меньшая проблема, чем разработчик клиентской части, переписывающий веб-сайт с Backbone на Angular, а с Angular на Vue.js

с периодичностью в полгода. Для разработчика да, некий смысл здесь есть, а именно — возможность внести затем в свое резюме соответствующий фреймворк. Но при этом они вредят себе, поскольку упускают возможность научиться создавать полезные промышленные решения на основе имеющегося потенциала.

Именно тут могут пригодиться заранее обученные модели и высокоуровневые утилиты из GCP. Обращение к этим API могло бы оказаться выгодным многим компаниям, вместо того чтобы делать эквивалент портирования клиентской части на самый популярный в текущем месяце JavaScript-фреймворк, только в сфере науки о данных. Это помогло бы также сгладить углы для некоторых команд исследователей данных, позволяя быстро выдавать на-гора результаты, одновременно работая над более сложными задачами.

Другие высокоуровневые сервисы, предоставляемые GCP, могут также сильно повысить ценность команды исследователей данных. Один из них — Datalab, для которого существует связанный с ним «бесплатный» продукт — Colaboratory (<https://colab.research.google.com/>). Эти сервисы с самого начала приносили колоссальную пользу за счет решения безумных проблем с управлением. Кроме того, благодаря удобной интеграции с платформой GCP они очень сильно упростили тестирование сервисов и прототипов решений. Более того, с приобретением сервиса Kaggle (<http://kaggle.com/>) интеграция с инструментарием Google намного углубилась, что упрощает наем исследователей данных, умеющих использовать, например, BigQuery. Это был ловкий ход со стороны Google.

Одно из отличий платформы GCP от AWS — предоставление высокоуровневых PaaS-сервисов, например Firebase (<https://firebase.google.com/>).

Colaboratory

Colaboratory — исследовательский проект Google, не требующий установки и работающий в облаке (<https://colab.research.google.com/>). Он основан на блокнотах Jupiter, бесплатен и содержит множество заранее установленных пакетов, например Pandas, matplotlib и TensorFlow. В нем есть несколько очень удобных готовых возможностей, благодаря чему будет очень полезным во многих сценариях использования.

Вот список наиболее, как мне кажется, интересных возможностей.

- ❑ Удобная интеграция с электронными таблицами Google, облачным хранилищем Google и локальной файловой системой с возможностью преобразования их в тип `DataFrame` библиотеки `Pandas`.
- ❑ Поддержка как Python 2, так и Python 3.
- ❑ Возможность загрузки блокнотов.
- ❑ Блокноты хранятся в Google Drive и могут эксплуатировать те возможности совместного применения, благодаря которым так удобно работать с документами Google Drive.
- ❑ Возможность одновременного редактирования блокнота двумя пользователями.

Одна из наиболее впечатляющих возможностей Colaboratory — создание совместной среды обучения для проектов на основе блокнотов Jupiter. В нем решено несколько чрезвычайно каверзных задач: совместное применение, работа с наборами данных и предварительная установка библиотек. Дополнительный сценарий использования — программное создание блокнотов Colaboratory, подключенных в конвейер ИИ вашей компании. Можно заранее заполнить каталог блокнотами с доступом к запросам BigQuery или модели машинного обучения в качестве шаблонов для будущих проектов.

Приведу пример простейшего технологического процесса. Этот блокнот можно найти на GitHub по адресу https://github.com/noahgift/pragmaticai-gcp/blob/master/notebooks/dataflow_sheets_to_pandas.ipynb. На рис. 4.1 создается новый блокнот.

Далее устанавливаем библиотеку `gsread`:

```
!pip install --upgrade -q gsread
```

Для записи в электронную таблицу необходима аутентификация, как показано ниже. В результате создается объект `gc`:

```
from google.colab import auth
auth.authenticate_user()
```

```
import gsread
from oauth2client.client import GoogleCredentials
```

```
gc = gsread.authorize(GoogleCredentials.get_application_default())
```

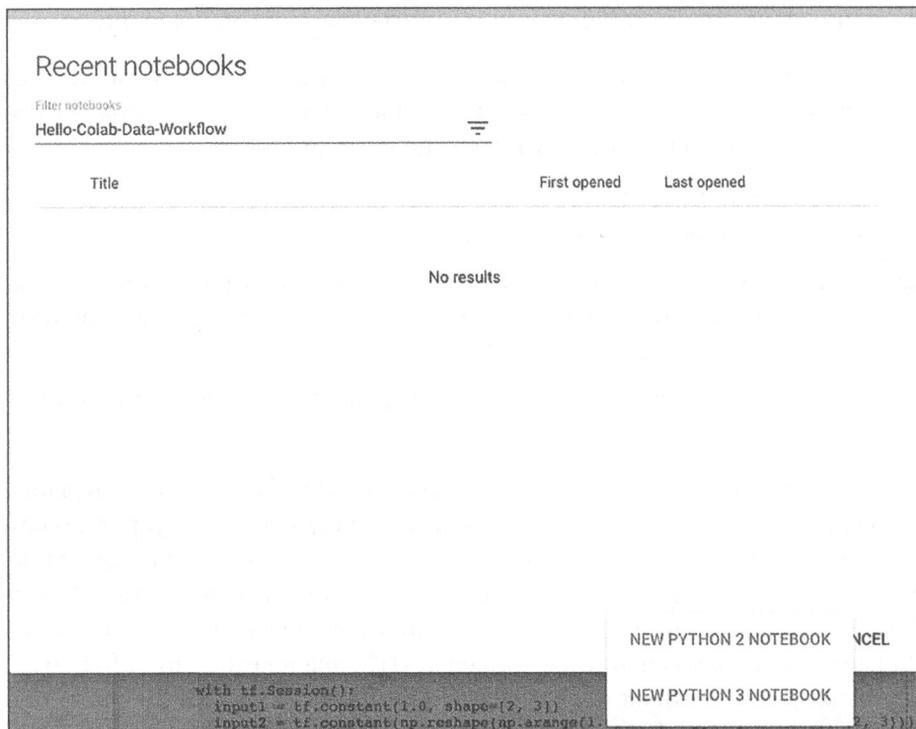


Рис. 4.1. Создание блокнота Colaboratory

Далее используем объект `gc` для создания электронной таблицы из одного столбца, заполненного значениями от 1 до 10:

```
sh = gc.create('pramaticai-test')
worksheet = gc.open('pramaticai-test').sheet1
cell_list = worksheet.range('A1:A10')
```

```
import random
count = 0
for cell in cell_list:
    count +=1
    cell.value = count
worksheet.update_cells(cell_list)
```

Наконец, преобразуем электронную таблицу в объект `DataFrame` библиотеки `Pandas`:

```
worksheet = gc.open('pramaticai-test').sheet1
rows = worksheet.get_all_values()
import pandas as pd
df = pd.DataFrame.from_records(rows)
```

Datalab

Следующая остановка в нашем туре по GCP — Datalab (<https://cloud.google.com/datalab/docs/quickstart>). Вся экосистема gcloud требует установки набора инструментов разработчика (software development kit, SDK) из <https://cloud.google.com/sdk/downloads>. Другой вариант установки — с помощью терминала следующим образом:

```
curl https://sdk.cloud.google.com | bash
exec -l $SHELL
gcloud init
gcloud components install datalab
```

После инициализации среды gcloud можно запустить экземпляр Datalab. Следует отметить несколько интересных нюансов. Docker — потрясающая технология для запуска изолированных версий Linux, которые будут работать на вашем ноутбуке точно так же, как и в ЦОД или на другом ноутбуке какого-нибудь будущего сотрудника.

Расширяем возможности Datalab с помощью Docker и реестра контейнеров Google

Datalab можно запустить на локальной машине, как описано в следующем руководстве для новичков: <https://github.com/googledatalab/datalab/wiki/Getting-Started>. Локальная, бесплатная версия Datalab очень удобна, но еще больше возможностей можно получить за счет расширения исходного образа Datalab, сохранения его в собственном реестре контейнеров Google и последующего запуска его на виртуальном узле, более мощном, чем ваша рабочая станция или ноутбук, например, n1-highmem-32 с 16-процессорными ядрами и 104 Гбайт оперативной памяти.

Решение ранее недоступных для вашего локального ноутбука задач внезапно становится элементарным. Технологический процесс расширения

базового образа Docker для Datalab описан в упомянутом выше руководстве. Основной вывод: после клонирования репозитория необходимо внести изменения в Dockerfile.in.

Запуск полнофункциональных машин с помощью Datalab

Запуск полнофункционального экземпляра блокнота Jupiter выглядит так, как показано на рис. 4.2.

```
→ pragmaticai-gcp git:(master) datalab create\  
  --machine-type n1-highmem-16 pragai-big-instance  
Creating the instance pragai-big-instance  
Created [https://www.googleapis.com/compute/v1  
/projects/cloudai-194723/zones/us-central1-f/  
instances/pragai-big-instance].  
Connecting to pragai-big-instance.  
This will create an SSH tunnel and may prompt you  
to create an rsa key pair. To manage these keys, see  
https://cloud.google.com/compute/docs/instances/  
adding-removing-ssh-keys  
Waiting for Datalab to be reachable at http://localhost:8081/  
Updating project ssh metadata...-
```

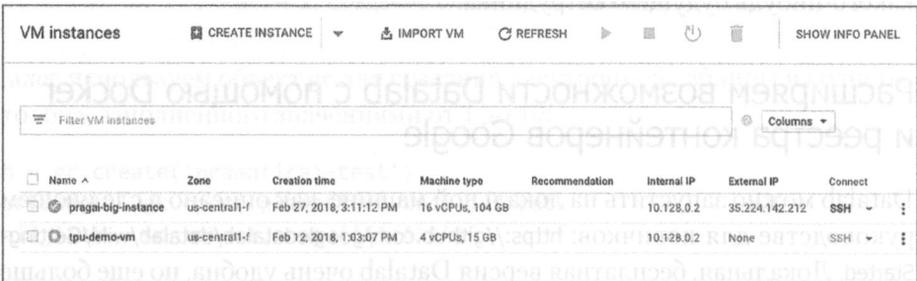


Рис. 4.2. Запущенный в консоли GCP экземпляр Datalab

Для демонстрации выполнения этим экземпляром реальной задачи мы синхронизировали с корзиной GCP найденные на сайте data.world (<https://data.world/dataquest/mlb-game-logs>) отчеты об играх Главной лиги бейсбола с 1871 по 2016 г. На рис. 4.3 с помощью команды describe показано, что в объект DataFrame библиотеки Pandas было загружено 171 000 строк.

```
gsutil cp game_logs.csv gs://pragai-datalab-test
Copying file://game_logs.csv [Content-Type=text/csv]...
- [1 files][129.8 MiB/129.8 MiB]
  628.9 KiB/s
Operation completed over 1 objects/129.8 MiB.
```

```
In [19]: df.describe()
```

	date	number_of_game	v_game_number	h_game_number	v_score	h_score	length_outs	attendanc
count	171907.000	171907.000	171907.000	171907.000	171907.000	171907.000	140841.000	118877.000
mean	19534616.307	0.261	76.930	76.954	4.421	4.701	53.620	20184.247
std	414932.618	0.606	45.178	45.163	3.278	3.356	5.572	14257.382
min	18710504.000	0.000	1.000	1.000	0.000	0.000	0.000	0.000
25%	19180516.000	0.000	38.000	38.000	2.000	2.000	51.000	7962.000
50%	19530530.000	0.000	76.000	76.000	4.000	4.000	54.000	18639.000
75%	19890512.000	0.000	115.000	115.000	6.000	6.000	54.000	31242.000
max	20161002.000	3.000	165.000	165.000	49.000	38.000	156.000	99027.000

8 rows x 83 columns

Рис. 4.3. 171 000 строк в объекте DataFrame из корзины GCP

Весь блокнот, содержащий нижеприведенные команды, можно найти на GitHub (<https://github.com/noahgift/pragmaticai-gcp/blob/master/notebooks/pragai-big-instance.ipynb>). Он начинается с нескольких импортов:

```
Import pandas as pd
pd.set_option('display.float_format', lambda x: '%.3f' % x)
import seaborn as sns
from io import BytesIO
```

Далее с помощью специальной команды Datalab выходные результаты присваиваются переменной `game_logs`:

```
%gcs read --object gs://pragai-datalab-test/game_logs.csv\
  --variable game_logs
```

Создается новый объект DataFrame:

```
df = pd.read_csv(BytesIO(game_logs))
```

И наконец, этот объект DataFrame выводится на график, как показано на рис. 4.4:

```
%timeit
ax = sns.regplot(x="v_score", y="h_score", data=df)
```

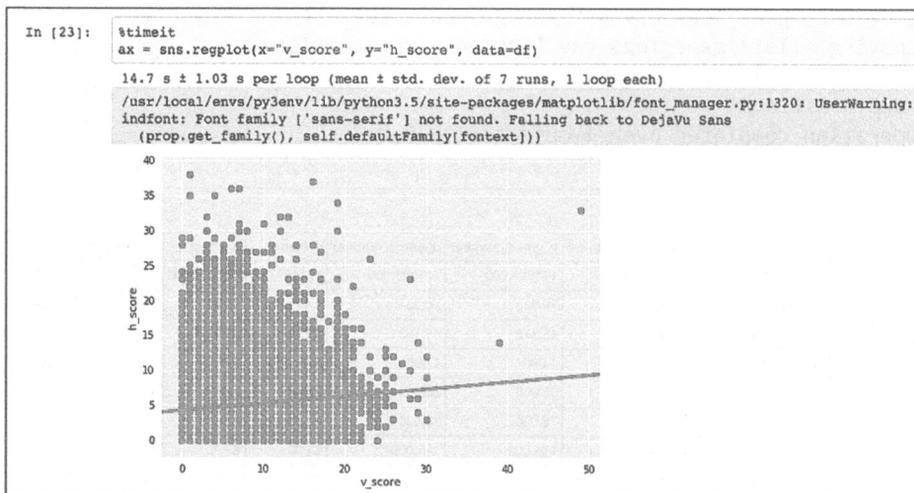


Рис. 4.4. Отрисовка на графике 171 000 строк с помощью библиотеки Seaborn на 32-ядерной машине с 100 Гбайт оперативной памяти занимает 17 с

Вывод из этого упражнения: Datalab имеет смысл использовать для ускорения мощных машин при выполнении разведочного анализа данных. Уже через секунду вы получаете результаты, на которые иначе пришлось бы потратить часы тяжелых вычислений и разведочного анализа данных. Еще один вывод: GCP выигрывает соревнование с Amazon в этом сегменте облачных сервисов, поскольку для разработчиков привычнее перемещать большие наборы данных в блокноты и использовать привычные утилиты «малых данных» вроде Seaborn и Pandas.

И еще один вывод: Datalab отлично подходит в качестве фундамента для создания конвейеров машинного обучения промышленного класса. В них можно исследовать корзины GCP, без проблем интегрировать BigQuery, да и другие части экосистемы GCP: механизм машинного обучения, TPU и реестр контейнеров.

BigQuery

BigQuery — одна из жемчужин экосистемы GCP и прекрасный сервис для построения вокруг него прагматичных конвейеров машинного обучения и ИИ для промышленной эксплуатации. Во многих аспектах это решение

превосходит AWS в смысле удобства для разработчика. Завершенные комплексные решения — более сильная сторона AWS, а GCP ориентируется на утилиты и уже привычные для разработчиков парадигмы. Перемещать данные в/из BigQuery очень легко, и можно делать это множеством различных способов, начиная от командной строки на локальной машине и до корзин GCP и вызовов API.

Перемещение данных в BigQuery из командной строки. Простейший способ переместить данные в BigQuery — воспользоваться утилитой командной строки bq. Рекомендую следующий подход.

Во-первых, проверьте, есть ли в проекте по умолчанию наборы данных:

```
→ pragmaticai-gcp git:(master) bq ls
```

В данном случае существующих наборов данных в проекте по умолчанию нет, так что будет создан новый набор:

```
→ pragmaticai-gcp git:(master) bq mk gamelogs
Dataset 'cloudai:gamelogs' successfully created.
```

Далее созданный набор данных проверяется с помощью команды bq ls:

```
→ pragmaticai-gcp git:(master) bq ls
datasetId
-----
gamelogs
```

Затем на сервер загружается содержащий 171 000 записей локальный CSV-файл размером 134 Мбайт с указанием флага --autodetect. Этот флаг делает возможным отложенную загрузку на сервер наборов данных из нескольких столбцов, благодаря тому что не требует заранее описывать схему.

```
→ pragmaticai-gcp git:(master) X bq load\
  --autodetect gamelogs.records game_logs.csv
Upload complete.
Waiting on bqjob_r3f28bca3b4c7599e_00000161daddc035_1
... (86s) Current status: DONE
→ pragmaticai-gcp git:(master) X du -sh game_logs.csv
134M  game_logs.csv
```

Теперь, после загрузки на сервер данных, можно выполнять к ним запросы из Datalab, как показано на рис. 4.5.

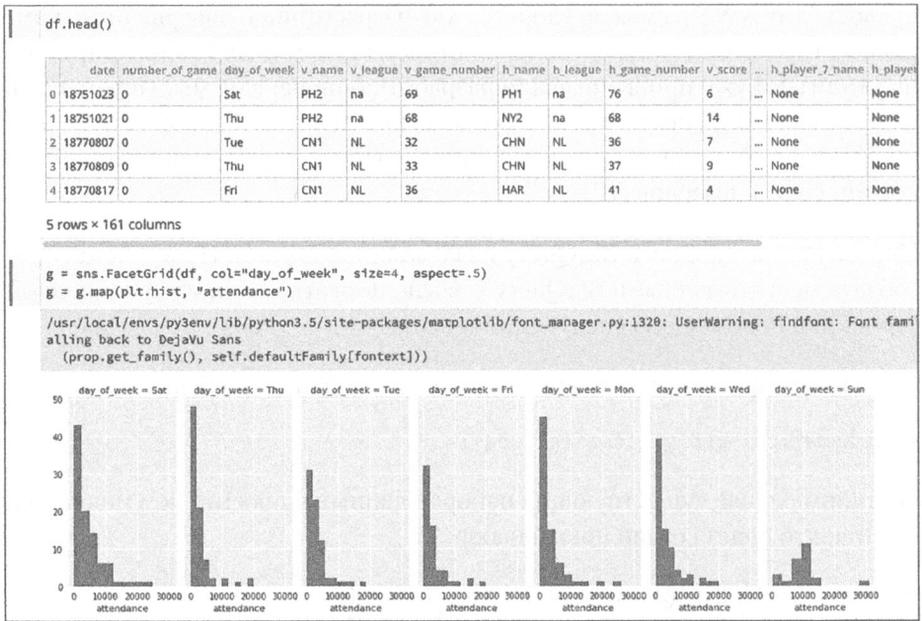


Рис. 4.5. Из BigQuery в Pandas и далее в конвейер Seaborn

Сначала выполняем импорты. Обратите внимание на импорт модуля `google.datalab.bigquery`, упрощающего обращение к BigQuery:

```
import pandas as pd
import google.datalab.bigquery as bq
pd.set_option('display.float_format', lambda x: '%.3f' % x)
import seaborn as sns
from io import BytesIO
```

Далее результат запроса преобразуется в `DataFrame`. Всего, как вы помните, у нас 171 000 строк, но для данного запроса мы ограничимся 10 000 строк:

```
some_games = bq.Query('SELECT * FROM `gamelogs.records` LIMIT 10000')
df = some_games.execute(output_options=
    bq.QueryOutput.dataframe()).result()
```

Наконец, этот объект `DataFrame` преобразуется в визуализацию Seaborn с графиками по дням недели:

```
g = sns.FacetGrid(df, col="day_of_week", size=4, aspect=.5)
g = g.map(plt.hist, "attendance")
```

Из данного примера конвейера можно сделать вывод, что BigQuery и Datalab производят сильное впечатление и способствуют удобному созданию конвейеров машинного обучения. В BigQuery можно за считанные минуты загрузить громадные наборы данных, запустить выполнение разведочного анализа данных на рабочей станции Jupiter, после чего преобразовать результаты в выполняемые блокноты.

Этот набор программных средств можно подключить прямо к сервисам машинного обучения Google или к обучению пользовательских моделей классификации с помощью TPU. Как уже упоминалось ранее, одно из преимуществ платформы AWS — технологический процесс, стимулирующий использование распространенных библиотек с открытым исходным кодом, таких как Seaborn и Pandas. С одной стороны, эти утилиты при достаточно больших наборах данных в конце концов отказывают, но они представляют собой дополнительный слой, благодаря которому становится возможным обращаться к привычным инструментам.

Облачные сервисы ИИ компании Google

Прагматизм в ИИ крайне приветствуется. Почему бы тогда по возможности не использовать готовые инструменты? К счастью, GCP включает несколько удобных сервисов, начиная от чисто экспериментальных и заканчивая вполне серьезными, на основе которых можно строить свою компанию. Вот краткий список основных таких сервисов.

- ❑ Облачное автоматическое машинное обучение (Cloud AutoML, <https://cloud.google.com/automl/>).
- ❑ Облачные тензорные процессоры (Cloud TPU, <https://cloud.google.com/tpu/>).
- ❑ Механизм облачного машинного обучения (Cloud Machine Learning Engine, <https://cloud.google.com/ml-engine/>).
- ❑ Облачный поиск вакансий (Cloud Job Discovery¹, <https://cloud.google.com/job-discovery/>).

¹ Сейчас носит название Cloud Talent Solution (<https://cloud.google.com/solutions/talent-solution/>).

- ❑ Облачный диалоговый интерфейс, корпоративная версия (Dialogflow Enterprise Edition, <https://cloud.google.com/dialogflow-enterprise/>).
- ❑ Облачный анализ текстов на естественном языке (Cloud Natural Language, <https://cloud.google.com/natural-language/>).
- ❑ Облачный API распознавания речи (Cloud Speech-to API, <https://cloud.google.com/speech/>).
- ❑ Облачный API машинного перевода (Cloud Translation API, <https://cloud.google.com/translate/>).
- ❑ Облачный API компьютерного зрения (Cloud Vision API, <https://cloud.google.com/vision/>).
- ❑ Облачное распознавание изображений (Cloud Video Intelligence, <https://cloud.google.com/video-intelligence/>).

ИИ-сервисы также могут служить дополнением к существующему ЦОД или облаку. Почему бы не попробовать работу сервиса облачного анализа текстов на естественном языке от Google на своих наборах данных из AWS и сравнить работу обоих сервисов, вместо того чтобы — упаси боже — обучать свою собственную модель этому? Прагматичным командам разработчиков ИИ обычно хватает ума для выбора подобных готовых решений, промышленного их применения, с тем чтобы сфокусировать свои усилия на создании алгоритмов машинного обучения под конкретную задачу в тех областях, где это действительно нужно.

Рекомендуемый технологический процесс использования указанных сервисов не слишком отличается от других приведенных в данной главе примеров: запуска Datalab, загрузки в него данных и экспериментов с API. Можно исследовать API, однако, и другим способом: просто загружать данные в анализатор API для каждого из сервисов. Во многих случаях будет оптимально начать с этого.

Классифицируем мою собаку с помощью API компьютерного зрения Google. По адресу <https://cloud.google.com/vision/docs/quickstart> можно найти пример использования API компьютерного зрения с помощью анализатора API. Для его тестирования я загрузил фото моей собаки `titan_small.jpg` в корзину `pragai-cloud-vision`.

На рис. 4.6 показан запрос API к этой корзине/файлу. На рис. 4.7 изображен Титан — моя собака.

Request body

```
{
  "requests": [
    {
      "features": [
        {
          "type": "LABEL_DETECTION"
        }
      ],
      "image": {
        "source": {
          "imageUri": "gs://pragai-cloud-vision/titan_small.jpg"
        }
      }
    }
  ]
}
```

Press ctrl+space or click one of the hint bubbles for suggestions.

Рис. 4.6. Запрос API компьютерного зрения Google в браузере



Рис. 4.7. Как бы хитро Титан ни смотрел, ему не обмануть ИИ Google

Так что же система классификации изображений выяснила про нашу собаку? Оказалось, что она с вероятностью более 50 % — далматинец. Скорее всего, она — помесь, а не чистокровная.

```
{
  "responses": [
    {
      "labelAnnotations": [
        {
          "mid": "/m/0bt91r",
          "description": "dog",
          "score": 0.94724846,
          "topicality": 0.94724846
        },
        {
          "mid": "/m/0kpmf",
          "description": "dog breed",
          "score": 0.91325045,
          "topicality": 0.91325045
        },
        {
          "mid": "/m/05mqq3",
          "description": "snout",
          "score": 0.75345945,
          "topicality": 0.75345945
        },
        {
          "mid": "/m/01z5f",
          "description": "dog like mammal",
          "score": 0.7018985,
          "topicality": 0.7018985
        },
        {
          "mid": "/m/02rjc05",
          "description": "dalmatian",
          "score": 0.6340561,
          "topicality": 0.6340561
        },
        {
          "mid": "/m/02xl47d",
          "description": "dog breed group",
          "score": 0.6023531,
          "topicality": 0.6023531
        },
      ]
    }
  ]
}
```

```

    {
      "mid": "/m/03f5jh",
      "description": "dog crossbreeds",
      "score": 0.51500386,
      "topicality": 0.51500386
    }
  ]
}
]
}

```

Тензорные процессоры Google и TensorFlow

В 2018 г. наметилась тенденция появления оригинальных ускорителей машинного обучения. В феврале 2018 г. Google выпустил бета-версию своих TPU, хотя они уже давно использовались внутри компании для таких продуктов, как поиск изображений Google (Google Image Search), фотографии Google (Google Photos) и облачный API компьютерного зрения (Cloud Vision API). Техническую информацию о TPU можно найти в документе «Анализ производительности тензорных процессоров в ЦОД» (*In-Datacenter Performance of a Tensor Processing Unit*, <https://drive.google.com/file/d/0Bx4hafXDDq2EMzRncy1vSUxtcEk/view>). В ней упоминается, в частности, о «Следствии про рог изобилия» из закона Амдала (Amdahl): «Добиться высокой производительности при разумных затратах можно и при низком коэффициенте использования огромного дешевого ресурса».

TPU, подобно некоторым из других сервисов ИИ Google, могут оказаться революционным технологическим прорывом. В частности, TPU — заманчивый вариант вложения средств. Если Google удастся добиться простоты обучения моделей глубокого обучения с помощью SDK TensorFlow, а также высокого КПД использования специализированных аппаратных ускорителей ИИ, то существенное преимущество над другими провайдерами облачных сервисов им гарантировано.

По иронии судьбы, однако, хотя Google движется семимильными шагами к максимальному удобству использования разработчиками различных частей своей облачной экосистемы, SDK TensorFlow все еще далек в этом

смысле от совершенства. Он имеет очень низкий уровень, сложен и, похоже, написан в расчете на аспирантов-математиков, которые предпочитают писать на ассемблере и языке C++. Впрочем, есть несколько программных решений, таких как PyTorch, смягчающих такое положение дел. И как бы я лично ни восхищался TPU, мне кажется, что Google должен в какой-то момент прозреть и осознать сложность TensorFlow. Вероятно, подобное когда-нибудь случится, хотя, стоит отметить, это еще terra incognita.

Запуск MNIST на облачных тензорных процессорах. Данное руководство задумано как дополнение к уже существующему руководству по TPU, находящемуся на предварительной стадии на момент написания этой книги. Найти последнее можно по адресу <https://cloud.google.com/tpu/docs/tutorials/mnist>. Для начала необходимо установить не только SDK gcloud, но и бета-компоненты:

```
gcloud components install beta
```

Далее нам понадобится виртуальная машина (VM), которая будет служить контроллером заданий. Для создания четырехъядерной виртуальной машины в центральном регионе США мы воспользуемся интерфейсом командной строки gcloud:

```
(.tpu) ➔ google-cloud-sdk/bin/gcloud compute instances \
  create tpu-demo-vm \
  --machine-type=n1-standard-4 \
  --image-project=ml-images \
  --image-family=tf-1-6 \
  --scopes=cloud-platform
```

```
Created [https://www.googleapis.com/compute/v1/\
  projects/cloudai-194723/zones/us-central1-f/\
  instances/tpu-demo-vm].
```

NAME	ZONE	MACHINE_TYPE
tpu-demo-vm	us-central1-f	n1-standard-4
RUNNING		

После создания VM необходимо создать экземпляр TPU:

```
google-cloud-sdk/bin/gcloud beta compute tpus create demo-tpu \
  --range=10.240.1.0/29 --version=1.6
Waiting for [projects/cloudai-194723/locations/us-central1-f/\
  operations/operation-1518469664565-5650a44f569ac-9495efa7-903
  9887d] to finish...done.
```

Created [demo-tpu].

```
google-cloud-sdk/bin/gcloud compute ssh tpu-demo-vm -- -L \  
6006:localhost:6006
```

Далее мы скачиваем данные для проекта, после чего загружаем их в облачное хранилище. Обратите внимание, что моя корзина называется `tpu-research`, а у вас будет корзина с другим названием:

```
python /usr/share/tensorflow/tensorflow/examples/how_tos/  
reading_data/convert_to_records.py --directory=./data  
gunzip ./data/*.gz  
export GCS_BUCKET=gs://tpu-research  
gsutil cp -r ./data ${STORAGE_BUCKET}
```

Наконец, значение переменной `TPU_NAME` должно соответствовать имени созданного ранее экземпляра TPU:

```
export TPU_NAME='demo-tpu'
```

Последний этап — обучение модели. В этом примере итераций немного, так что, поскольку производительность TPU очень высока, имеет смысл для эксперимента добавить несколько нулей:

```
python /usr/share/models/official/mnist/mnist_tpu.py \  
--tpu_name=${TPU_NAME} \  
--data_dir=${STORAGE_BUCKET}/data \  
--model_dir=${STORAGE_BUCKET}/output \  
--use_tpu=True \  
--iterations=500 \  
--train_steps=1000
```

После этого модель выводит значение функции потерь. Можно также взглянуть на инструментальную панель TPU, где есть множество интересной графической информации. Необходим еще последний штрих — очистка. Очистить используемые TPU, как показано ниже, необходимо во избежание дальнейших издержек:

```
noahgift@tpu-demo-vm:~$ gcloud beta compute tpus delete demo-tpu  
Your TPU [demo-tpu] will be deleted.  
Do you want to continue (Y/n)? y  
Waiting for [projects/cloudai-194723/locations/us-central1-f/  
operations/operation-1519805921265-566416410875e-018b840b  
-1fd71d53] to finish...done.  
Deleted [demo-tpu].
```

Резюме

GCP — достойный конкурент AWS в плане построения прагматичных решений ИИ. У него есть некоторые преимущества над AWS и уникальные предложения, в основном касающиеся ориентированности на удобство разработчиков и готовые высокоуровневые сервисы ИИ.

Любознательным разработчикам, практикующим ИИ, рекомендуется взглянуть на различные API ИИ и выяснить для себя, как наладить их взаимодействие с целью создания простых и понятных решений. Одна из возможностей (а равно и вызовов для разработчика), созданных компанией Google, — TPU с экосистемой TensorFlow. С одной стороны, она исключительно сложна для начинающих, с другой — возможности ее очень соблазнительны. Экспертные знания в сфере TPU как минимум не мешают любой компании, претендующей на лидерство в области ИИ.

5

Разработка ИИ в облачной среде с помощью веб-сервисов Amazon

Твоя любовь делает меня сильнее.
Твоя ненависть делает меня
неудержимым.

*Роналдо
(Ronaldo)*

Акции FANG (Facebook, Amazon, Netflix и Google) неудержимо растут в цене в последние несколько лет. Акции одного только Amazon выросли на 300 % за период с марта 2015-го по март 2018 г. Netflix также работает на основе AWS. С точки зрения карьерного роста динамика облака AWS очень неплоха и там крутятся огромные деньги. Понимание специфики работы этой платформы и предоставляемых ею сервисов критически важно для успеха в ближайшем будущем многих приложений ИИ.

Из такого массового перемещения денежных масс можно сделать вывод, что облака не только воцарились надолго, но и меняют базовую парадигму разработки программного обеспечения. В частности, AWS сделал ставку на бессерверные технологии. Жемчужина этого стека — технология AWS Lambda, позволяющая выполнять функции на множестве языков — Go, Python, Java, C# и Node — в виде событий внутри большей экосистемы. А облако можно рассматривать как новую операционную систему.

Язык Python известен жесткими ограничениями по масштабируемости в силу самой своей природы. Несмотря на вдохновенные ошибочные, но *почти убедительные* аргументы в пользу того, что глобальная блокировка интерпретатора (global interpreter lock, GIL), как и производительность

Python, неважны, на самом деле на практике при больших масштабах они имеют значение. Именно те черты Python, которые упрощают его использование, исторически оказываются проклятием для его производительности. В частности, GIL, по сути, блокирует возможность эффективного распараллеливания при масштабных вычислениях, по сравнению с другими языками программирования, такими как Java. Конечно, существуют способы обхода подобных ограничений на целевой Linux-машине, но это часто приводит к колоссальным тратам времени на неуклюжее переписывание принципов конкурентности языка Erlang на Python или к простаиванию ядер процессора.

При использовании технологии AWS Lambda этот недостаток Python теряет значение, поскольку роль операционной системы играет сам AWS. Вместо того чтобы применять потоки выполнения или процессы для распараллеливания кода, разработчик облачных сервисов может воспользоваться SNS, SQS, Lambda и другими технологиями на основе стандартных блоков. Эти примитивы далее занимают место потоков выполнения, процессов и других парадигм традиционных операционных систем. Дальнейшее подтверждение проблем с масштабированием Python в операционной системе Linux связано с углубленными исследованиями предполагаемых широких возможностей масштабирования проектов Python.

Если копнуть поглубже, вы обнаружите, что фактически всю работу выполняет, скажем, RabbitMQ, и/или Celery (написанные на Erlang), или Nginx, написанный на высокооптимизированном C. Но не слишком увлекайтесь Erlang (я руководил компанией, интенсивно его использовавшей) — нанять кого-то, кто пишет на этом языке, практически нереально. Язык Go понемногу начинает решать те же проблемы с масштабированием, а нанять разработчиков на Go намного проще. С другой стороны, возможно, лучше всего перенести реализацию конкурентности вглубь облачной операционной системы, сняв ее со своих плеч. Тогда увольнение вашего бесценного разработчика на Erlang или Go не приведет к банкротству компании.

К счастью, однако, при использовании бессерверной технологии недостатки Python при работе в операционной системе Linux внезапно оказываются неважны. Я столкнулся с хорошим примером этого во время работы инженером в занимавшейся большими данными компании Loggly. Мы пытались написать высокопроизводительную асинхронную систему ввода и обработки журналов на Python, и да, при работе на одном ядре она демонстрировала

отличные результаты, от шести до восьми тысяч записей в секунду. Но проблема заключалась в простое остальных ядер, и масштабирование этого асинхронного средства сбора данных на несколько ядер не оправдывало затрат. С помощью создания бессерверных компонентов из AWS, однако, написание всей системы на Python — прекрасная идея в силу естественной масштабируемости данной платформы.

Но не только эти новые облачные операционные системы имеет смысл учитывать. Многие такие технологии, как веб-фреймворки, представляют собой абстракции, построенные на абстракциях из далекого прошлого. Реляционные базы данных были придуманы в 1970-х, и это добротная технология, но в начале 2000-х разработчики веб-фреймворков нагромоздили поверх этой, возникшей в эру персональных компьютеров и ЦОД, технологии множество веб-фреймворков с помощью средств объектно-реляционного отображения и утилит генерации кода. Создание веб-фреймворков становится, чуть ли не преднамеренно, инвестицией в устаревший образ мышления. Безусловно, они способны решать свои задачи и обладают большими возможностями, но за ними ли будущее, особенно с учетом масштабных проектов ИИ? Не думаю.

Бессерверные технологии представляют совершенно иной стиль мышления. Они предлагают автоматическое масштабирование баз данных наряду с гибкостью и эффективностью управления схемами. Вместо запуска клиентской веб-части, например Apache или Nginx, передающих запросы прикладному коду, здесь применяются серверы приложений без сохранения состояния, запускаемые только при поступлении событий.

Возникает, однако, проблема сложности. Поскольку сложность приложений машинного обучения и ИИ растет, приходится чем-то жертвовать ради ее снижения. Один из способов уменьшения сложности — отказ от серверов, а значит, и затрат на их сопровождение. Подобное может также означать, что пришла пора сказать *rm -rf*¹ традиционным веб-фреймворкам. Однако это дело не ближайших дней, так что в данной главе мы рассмотрим традиционное веб-приложение, Flask, но с «привкусом» облачной операционной системы. В других главах вы найдете примеры чисто бессерверной архитектуры, а именно с использованием AWS Chalice.

¹ Команда полного удаления каталога вместе с подкаталогами, без запроса подтверждения.

Создание решений дополненной и виртуальной реальностей на основе AWS

Работая в киноиндустрии и в Caltech, я начал высоко ценить мощные файловые серверы на Linux, установленные на всех рабочих станциях компании. Единая центральная точка подключения для настройки операционной системы, распределения данных и совместного использования дисковых операций ввода/вывода для тысяч машин и пользователей предоставляет большие возможности.

Мало кому известно, что множество кинокомпаний включено в список обладателей 500 самых мощных суперкомпьютеров, причем уже многие годы (<https://www.top500.org/news/new-zealand-to-join-petaflop-club/>). Причина состоит в том, что системы визуализации, связанные с высокопроизводительными централизованными файловыми серверами, используют колоссальные вычислительные ресурсы и ресурсы дискового ввода/вывода. В кинокомпании Weta Digital, когда я работал там над фильмом «Аватар» в Новой Зеландии в 2009 г. (<https://www.geek.com/chips/the-computing-power-that-created-avatar-1031232/>), было более 40 000 процессоров с 104 Тбайт памяти. Они обрабатывали ежедневно до 1,4 млн задач, так что многие ветераны кинопромышленности снисходительно улыбаются, слыша про нынешние рабочие нагрузки Spark и Hadoop.

Весь смысл упоминания об этом (помимо того, чтобы сказать новичкам в больших данных: «Прочь с моей лужайки») — показать, насколько прекрасен централизованный файловый сервер при масштабных вычислениях. Исторически, однако, так сложилось, что эти «Феррари» среди файловых серверов требуют целой команды «механиков» для поддержания работоспособности. А с наступлением эры облачных вычислений внезапно для получения такого файлового сервера оказывается достаточно выбрать его и щелкнуть кнопкой мыши.

Компьютерное зрение: создание конвейеров AR/VR с помощью EFS и Flask

В AWS есть адаптивная файловая система (Elastic File System, EFS) — сервис, служащий именно для получения превосходного файлового сервера буквально за один щелчок кнопкой мыши. В прошлом мне случалось использовать этот сервис, в частности, для создания в AWS централизованного

файлового сервера для конвейера компьютерного зрения на основе виртуальной реальности (VR). Все ресурсы, код и созданные артефакты хранились в EFS. На рис. 5.1 показано, как может выглядеть конвейер VR в AWS при применении EFS. Блоки камер могут насчитывать 48 или 72 камеры, каждая из которых генерирует крупные кадры, поступающие в дальнейшем для обработки в алгоритм компоновки сцен виртуальной реальности.

Высокоуровневый обзор конвейера виртуальной реальности

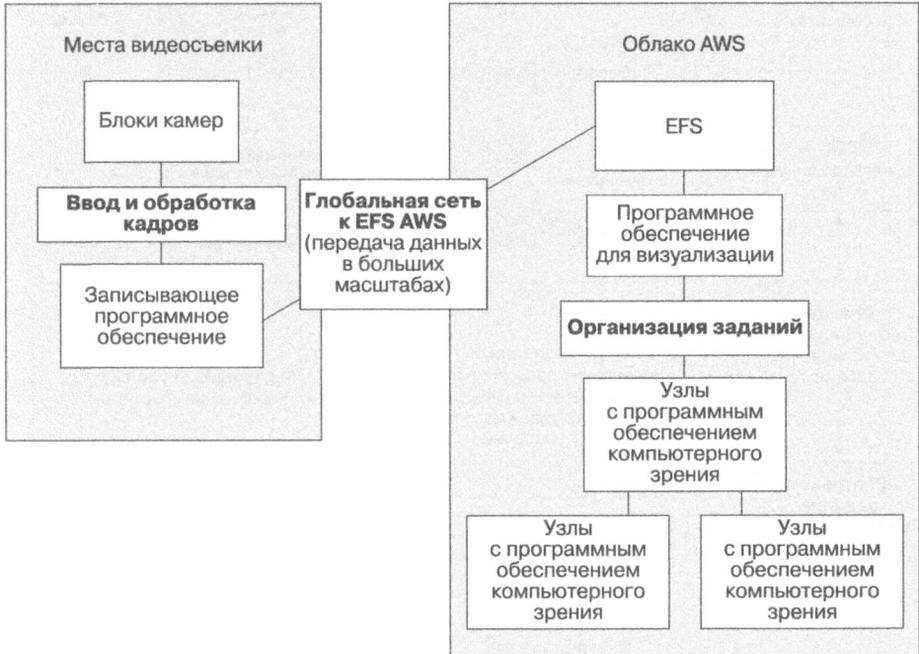


Рис. 5.1. Конвейер виртуальной реальности в AWS с использованием EFS

Один незаметный, но существенный нюанс состоит в серьезном упрощении развертывания кода приложений Python, поскольку можно создать точки подключения EFS для каждой из сред, скажем, разработки (DEV), предэксплуатационного тестирования (STAGE) и промышленной эксплуатации (PRODUCTION). После этого для развертывания достаточно будет выполнить удаленную синхронизацию (rsync) кода между сервером сборки и точкой подключения EFS, что можно сделать за доли секунды в зависимости от ветки: допустим, точка подключения DEV представляет собой основную ветку, а точка подключения STAGE — ветку предэксплуатационного

тестирования и т. д. При этом фреймворк Flask обеспечивает наличие на диске свежей версии кода, благодаря чему развертывание становится тривиальной задачей. Пример можно найти на рис. 5.2.

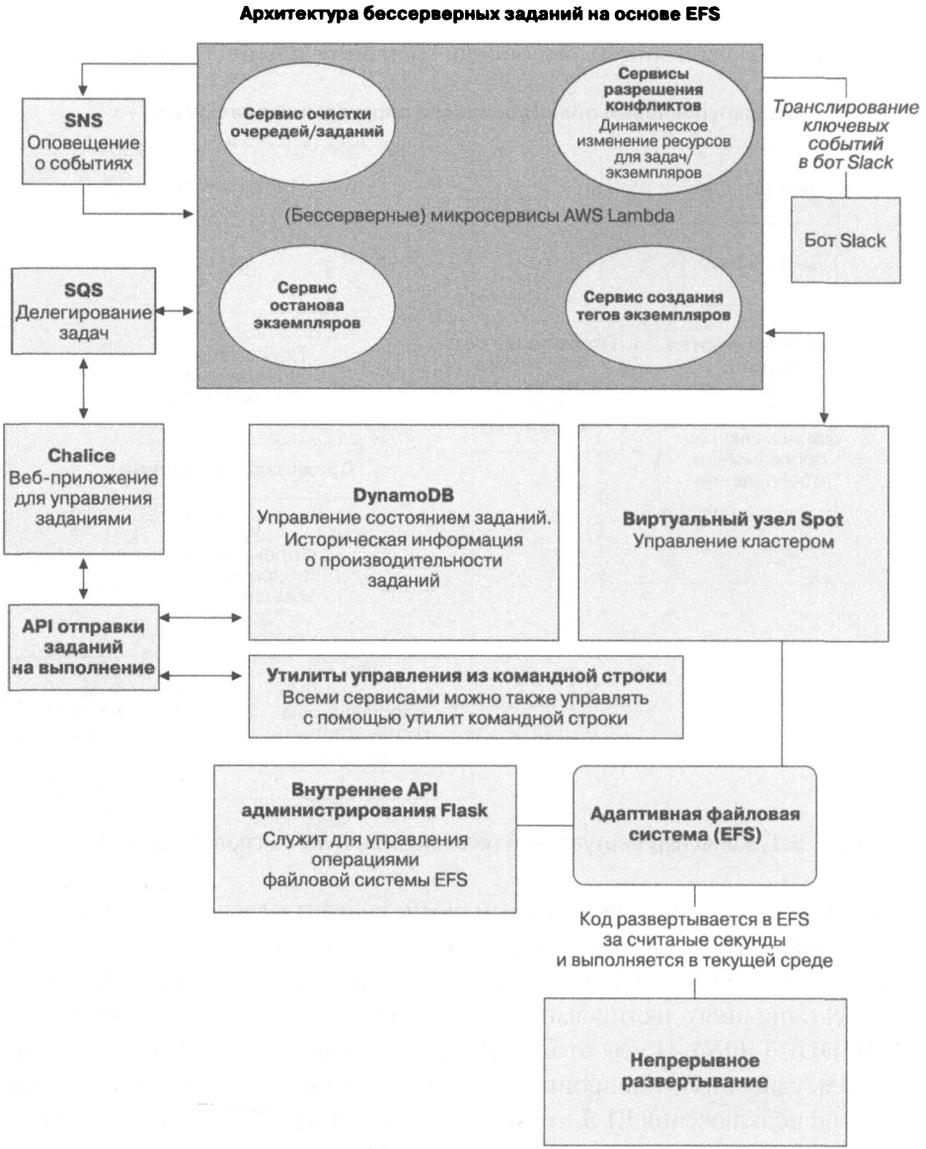


Рис. 5.2. Подробная схема бессерверной архитектуры и архитектуры Flask

Возможность использования при необходимости бессерверной технологии весьма упрощает задачу, кроме того, EFS вместе с Flask — мощный инструмент для создания ИИ продуктов всех мастей. В приведенных примерах речь идет о конвейерах компьютерного зрения/VR/AR, но EFS может пригодиться и при инженерии данных для обычного машинного обучения.

Наконец, я уверен в работоспособности этой архитектуры еще и потому, что создавал ее с нуля для компании, занимающейся AR/VR, и она оказалась настолько популярна, что мы всего за несколько месяцев израсходовали \$100 000 кредита от AWS, запуская гигантские задания на многих сотнях узлов. Иногда успех означает изрядную дыру в кармане.

Создание конвейера инженерии данных с помощью EFS, Flask и Pandas

При создании конвейера ИИ промышленного класса основная трудность часто заключается в инженерии данных. В следующем подразделе будет подробно описано, как можно заложить начала API для промышленной эксплуатации в компаниях вроде Netflix, AWS или стартапах с миллиардной капитализацией. Командам исследователей данных часто приходится создавать библиотеки и сервисы для упрощения работы с данными на используемой их компанией платформе.

В этом примере мы выполним в качестве подтверждения работоспособности концепции агрегирование данных в формате CSV. Воплощающий REST API будет принимать на входе файл в формате CSV, столбец, по которому необходимо группировать данные, и столбец, в котором будет возвращаться результат. Отмечу также реалистичность этого примера: он включает реализацию всех мелких нюансов, таких как документирование API, тестирование, непрерывная поставка, плагины, а также оценку производительности.

Входные данные для нашей задачи выглядят следующим образом:

```
first_name,last_name,count
chuck,norris,10
kristen,norris,17
john,lee,3
sam,mcgregor,15
john,mcgregor,19
```

После обработки с помощью API они приобретут такой вид:

```
norris,27  
lee,3  
mcgregor,34
```

Код проекта можно найти здесь: <https://github.com/noahgift/pai-aws>. А поскольку сборочные файлы и виртуальные среды мы подробно рассматриваем в других главах, перейдем прямо к коду. Данный проект состоит из пяти основных частей: приложения Flask, библиотеки `nlib`, блокнотов, тестов и утилиты командной строки.

Приложение Flask

Приложение Flask состоит из трех компонентов: каталога `static`, содержащего файл `favicon.ico`; каталога `templates`, содержащего файл `index.html`; и основного веб-приложения, насчитывающего около 150 строк кода. Вот пошаговый разбор кода основного приложения Flask.

В первом разделе файла производится импорт Flask и его расширения `flasgger` (генератора документации на основе API фреймворка Swagger; <https://github.com/rochacbruno/flasgger>), а также описываются компоненты журналирования и приложение Flask:

```
import os  
import base64  
import sys  
from io import BytesIO  
  
from flask import Flask  
from flask import send_from_directory  
from flask import request  
from flask_api import status  
from flasgger import Swagger  
from flask import redirect  
from flask import jsonify  
  
from sensible.loginit import logger  
from nlib import csvops  
from nlib import utils  
  
log = logger(__name__)  
app = Flask(__name__)  
Swagger(app)
```

Создаем вспомогательную функцию для декодирования данных, находящегося в формате Base64:

```
def _b64decode_helper(request_object):
    """Возвращает декодированные из формата Base64 данные и размер
        исходных данных"""

    size=sys.getsizeof(request_object.data)
    decode_msg = "Decoding data of size: {size}".format(size=size)
    log.info(decode_msg)
    decoded_data = BytesIO(base64.b64decode(request.data))
    return decoded_data, size
```

Далее мы создаем несколько, по сути, шаблонных маршрутов, предназначенных для выдачи файла favicon.ico и перенаправления к основной документации:

```
@app.route("/")
def home():
    """Маршрут / будет перенаправлять пользователя на документацию
        API: /apidocs"""

    return redirect("/apidocs")
```

```
@app.route("/favicon.ico")
def favicon():
    """Favicon"""

    return send_from_directory(os.path.join(app.root_path, 'static'),
                              'favicon.ico',
                              mimetype='image/vnd.microsoft.icon')
```

Ситуация оказывается интереснее в случае маршрута /api/funcs. Там перечисляются устанавливаемые плагины, которые могут представлять собой пользовательские алгоритмы. Я опишу их подробнее в подразделе, посвященном библиотеке.

```
@app.route('/api/funcs', methods = ['GET'])
def list_apply_funcs():
    """Возвращает список допустимых функций

        GET /api/funcs
        ---
```

```
ответы:  
200:  
    описание: возвращает список допустимых функций.
```

"""

```
appliable_list = utils.appliable_functions()  
return jsonify({"funcs":appliable_list})
```

В этом разделе создается маршрут для группировки, она содержит подробную документацию в виде docstring, что позволяет динамически создать документацию на основе API swagger:

```
@app.route('/api/<groupbyop>', methods = ['PUT'])  
def csv_aggregate_columns(groupbyop):  
    """Агрегирует столбцы в загружаемом CSV-файле  
  
    ---  
    потребляет: application/json  
    параметры:  
        - [находится] в: путь  
          название: допустимая функция (то есть prsum, prmedian)  
          тип: строка  
          обязательный: да  
          описание: допустимая функция,  
          требующая регистрации (см. /api/funcs)  
        - [находится] в: запрос  
          название: столбец  
          тип: строка  
          описание: столбец для обработки при агрегировании  
          обязательный: да  
        - [находится] в: запрос  
          название: группировка  
          тип: строка  
          описание: \столбец, по которому требуется  
          группировать при агрегировании  
          обязательный: да  
        - [находится] в: заголовок  
          название: тип содержимого  
          тип: строка  
          описание: \  
            Требуется указание "Content-Type:application/json"  
          обязательный: да  
        - [находится] в: тело
```

```

название: содержимое
тип: строка
описание: CSV-файл в кодировке base64
обязательный: да

```

```
ответы:
```

```
200:
```

```
описание: возвращает агрегированный CSV-файл.
```

```
"""
```

Наконец, далее создается ядро вызова API. Обратите внимание, что здесь решается множество запутанных практических задач, например проверка правильного типа содержимого, поиск конкретного метода HTTP, журналирование динамически загружаемых плагинов, а также возвращение JSON-ответа с кодом состояния 200 в случае, если все в порядке, и других кодов состояния, если нет.

```

content_type = request.headers.get('Content-Type')
content_type_log_msg = \
    "Content-Type is set to: {content_type}".\
    format(content_type=content_type)
log.info(content_type_log_msg)
if not content_type == "application/json":
    wrong_method_log_msg = \
        "Wrong Content-Type in request:\
        {content_type} sent, but requires application/json".\
        format(content_type=content_type)
    log.info(wrong_method_log_msg)
    return jsonify({"content_type": content_type,
                    "error_msg": wrong_method_log_msg}),
status.HTTP_415_UNSUPPORTED_MEDIA_TYPE

#Синтаксический разбор параметров запроса и извлечение значений
query_string = request.query_string
query_string_msg = "Request Query String:
{query_string}".format(query_string=query_string)
log.info(query_string_msg)
column = request.args.get("column")
group_by = request.args.get("group_by")

#Журналирование и обработка параметров запроса
query_parameters_log_msg = \
    "column: [{column}] and group_by:\

```

```

    [{group_by}] Query Parameter values".\
    format(column=column, group_by=group_by)
log.info(query_parameters_log_msg)
if not column or not group_by:
    error_msg = "Query Parameter column or group_by not set"
    log.info(error_msg)
    return jsonify({"column": column, "group_by": group_by,
                    "error_msg": error_msg}), status.HTTP_400_BAD_REQUEST

#Загрузка плагинов и выбор нужного
plugins = utils.plugins_map()
appliable_func = plugins[groupbyopor]

#Распаковка данных и выполнение над ними нужных действий
data,_ = _b64decode_helper(request)
#Returns Pandas Series
res = csvops.group_by_operations(data,
    groupby_column_name=group_by, \
    apply_column_name=column, func=appliable_func)
log.info(res)
return res.to_json(), status.HTTP_200_OK

```

Следующий фрагмент кода устанавливает такие флаги, как `debug`, и включает шаблонный код для запуска приложения Flask в виде сценария:

```

if __name__ == "__main__": # pragma: no cover
    log.info("START Flask")
    app.debug = True
    app.run(host='0.0.0.0', port=5001)
    log.info("SHUTDOWN Flask")

```

Далее я создал следующий сборочный файл для запуска приложения:

```

(.pia-aws) → pai-aws git:(master) make start-api
#Устанавливает переменную среды PYTHONPATH так, чтобы она указывала
#на каталог, расположенный на один уровень выше
#В промышленной эксплуатации будет иначе
cd flask_app && PYTHONPATH=".." python web.py
2018-03-17 19:14:59,807 - __main__ - INFO - START Flask
* Running on http://0.0.0.0:5001/ (Press CTRL+C to quit)
* Restarting with stat
2018-03-17 19:15:00,475 - __main__ - INFO - START Flask
* Debugger is active!
* Debugger PIN: 171-594-84

```

На рис. 5.3 документация на основе swagger предоставляет пользователю список доступных функций, то есть плагинов из `nlib`. Как видим из выводимых результатов, загружены функции `nmedian`, `npsum`, `numpy` и `tanimoto`. На рис. 5.4 показана удобная веб-форма, благодаря которой разработчик может полностью осуществить вызов API без применения `curl` или какого-либо языка программирования. Самая сильная сторона этой системы — то, что основное веб-приложение состоит всего лишь из 150 строк кода, однако выглядит вполне настоящим и подходит для промышленной эксплуатации!

The screenshot shows a Swagger API documentation interface. At the top, it says "default" and has links for "Show/Hide", "List Operations", and "Expand Operations". The main section is for the endpoint "GET /api/funcs" with the description "Return a list of applicable functions". Below this, there are sections for "Implementation Notes", "GET /api/funcs", "Response Messages", "Curl", "Request URL", "Response Body", "Response Code", and "Response Headers".

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Returns list of applicable functions.		

Curl

```
curl -X GET --header 'Accept: application/json' 'http://0.0.0.0:5001/api/funcs'
```

Request URL

```
http://0.0.0.0:5001/api/funcs
```

Response Body

```
{
  "funcs": [
    "nmedian",
    "npsum",
    "numpy",
    "tanimoto"
  ]
}
```

Response Code

```
200
```

Response Headers

```
{
  "date": "Sun, 18 Mar 2018 02:23:17 GMT",
  "server": "werkzeug/0.14.1 Python/3.6.4",
  "content-length": "81",
  "content-type": "application/json"
}
```

Рис. 5.3. Список доступных плагинов

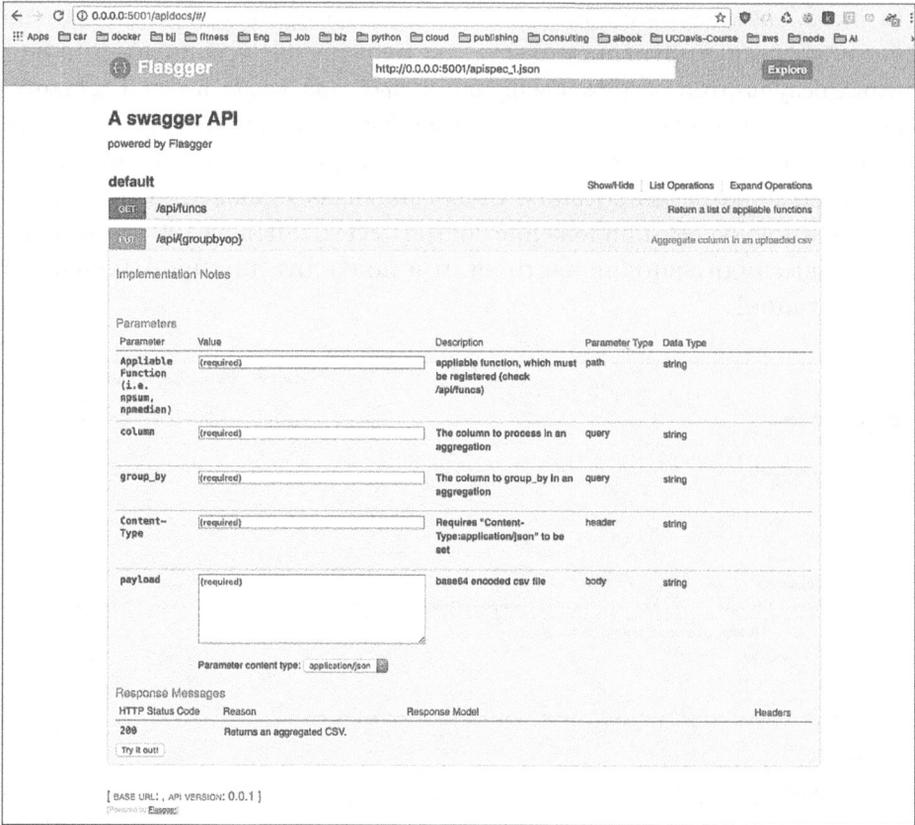


Рис. 5.4. Использование API

Библиотека и плагины

Внутри каталога `nlib` находится четыре файла: `__init__.py`, `applicable.py`, `csvops.py` и `utils.py`. Вот построчный разбор каждого из них.

Файл `__init__.py` очень прост, он содержит переменную для версии:

```
__version__ = 0.1
```

Далее идет файл `utils.py` — загрузчик плагинов, выбирающий «допустимую» функцию из файла `applicable.py`:

```
"""Утилиты
```

```
Основное назначение – утилита для работы с плагинами, обеспечивающая:
```

- * регистрацию

```

* обнаружение
* документирование
функций
"""

import importlib

from sensible.loginit import logger

log = logger(__name__)

def applicable_functions():
    """Возвращает список функций, допустимых для использования
    в операциях группировки"""

    from . import applicable
    module_items = dir(applicable)
    #Filter out special items __
    func_list = list(
        filter(lambda x: not x.startswith("__"),
              module_items))
    return func_list

def plugins_map():
    """Создает словарь доступных для вызова функций

    In [2]: plugins = utils.plugins_map()
    Loading applicable functions/plugins: npmedian
    Loading applicable functions/plugins: npsum
    Loading applicable functions/plugins: numpy
    Loading applicable functions/plugins: tanimoto

    In [3]: plugins
    Out[3]:
    {'npmedian': <function nlib.applicable.npmedian>,
    'npsum': <function nlib.applicable.npsum>,
    'numpy': <module 'numpy' from site-packages...>,
    'tanimoto': <function nlib.applicable.tanimoto>}

    In [4]: plugins['npmedian']([1,3])
    Out[4]: 2.0
    """

    plugins = {}
    funcs = applicable_functions()

```

```

for func in funcs:
    plugin_load_msg =\
        "Loading applicable functions/plugins:\
        {func}".format(func=func)
    log.info(plugin_load_msg)
    plugins[func] = getattr(
        importlib.import_module("nlib.applicable"), func
    )
return plugins

```

В файле `applicable.py` можно создавать пользовательские функции. Эти функции применяются к столбцу объекта `DataFrame` библиотеки `Pandas`, их можно адаптировать для любых действий, которые только выполняются над столбцом:

```

"""Функции, допустимые для операции группировки Pandas
(то есть плагины)"""

```

```
import numpy
```

```
def tanimoto(list1, list2):
    """Коэффициент Танимото
```

```

In [2]: list2=['39229', '31995', '32015']
In [3]: list1=['31936', '35989', '27489',
              '39229', '15468', '31993', '26478']
In [4]: tanimoto(list1,list2)
Out[4]: 0.1111111111111111

```

Определяет степень схожести двух множеств в числовом выражении с помощью операции пересечения

```
"""
```

```

intersection = set(list1).intersection(set(list2))
return float(len(intersection))\
        /(len(list1) + len(list2) - len(intersection))

```

```
def npsum(x):
    """Операция суммирования из библиотеки Numpy"""
```

```
    return numpy.sum(x)
```

```
def npmedian(x):
```

```
"""Операция вычисления медианы из библиотеки Numpy"""
```

```
return numpy.median(x)
```

Наконец, модуль `csvops` отвечает за ввод CSV-файла и операции над ним, как показано ниже:

```
"""
```

```
CSV:
```

Пояснения относительно производительности операций ввода/вывода в Pandas вы можете найти здесь:

```
http://pandas.pydata.org/pandas-docs/stable/io.html#io-perf
"""
```

```
from sensible.loginit import logger
import pandas as pd
```

```
log = logger(__name__)
log.debug("imported csvops module")
```

```
def ingest_csv(data):
    """Выполняет ввод CSV-файла с помощью возможностей библиотеки
    Pandas"""
```

```
    df = pd.read_csv(data)
    return df
```

```
def list_csv_column_names(data):
    """Возвращает список названий столбцов из CSV-файла"""
```

```
    df = ingest_csv(data)
    colnames = list(df.columns.values)
    colnames_msg = "Column Names: {colnames}."\
        format(colnames=colnames)
    log.info(colnames_msg)
    return colnames
```

```
def aggregate_column_name(data,
    groupby_column_name, apply_column_name):
    """Возвращает агрегированные по имени столбца результаты обработки
    CSV-файла в формате JSON"""
```

```
    df = ingest_csv(data)
```

```
res = df.groupby(groupby_column_name)[apply_column_name].sum()
return res
```

```
def group_by_operations(data,
    groupby_column_name, apply_column_name, func):
    """
```

Дает возможность использовать в операции группировки произвольные функции

```
In [14]: res_sum = group_by_operations(data=data,
    groupby_column_name="last_name", columns="count",
    func=np.sum)
```

```
In [15]: res_sum
```

```
Out[15]:
```

```
last_name
```

```
eagle    34
```

```
lee       3
```

```
smith    27
```

```
Name: count, dtype: int64
```

```
"""
```

```
df = ingest_csv(data)
grouped = df.groupby(groupby_column_name)[apply_column_name]
#Группировка с фильтрацией по конкретным столбцам
applied_data = grouped.apply(func)
return applied_data
```

Утилита командной строки

Возвращаясь к идее «прочь с моей лужайки», я создам утилиту командной строки, которая пригодится практически в любом проекте. Каковы бы ни были возможности блокнота Jupiter, существуют вещи, для которых просто лучше подходит утилита командной строки.

Файл `cvsccli.py` выглядит следующим образом. Для начала создаем шаблонную документацию и импорты:

```
#!/usr/bin/env python
"""
```

Утилита командной строки для операций с CSV:

- * Агрегирование

* Будущие доработки

```
"""
```

```
import sys

import click
from sensible.loginit import logger
```

```
import nlib
from nlib import csvops
from nlib import utils
```

```
log = logger(__name__)
```

Ядро утилиты командной строки делает то же, что HTTP API. В нее включены документация и пример входного файла в `ext/input.csv` для тестирования утилиты. Для удобства пользователей утилиты в `docstring` включен пример результата ее запуска:

```
@click.version_option(nlib.__version__)
@click.group()
def cli():
    """Утилита для операций над CSV-файлами

    """

    @cli.command("cvsops")
    @click.option('--file', help='Name of csv file')
    @click.option('--groupby', help='GroupBy Column Name')
    @click.option('--aplyname', help='Apply Column Name')
    @click.option('--func', help='Appliable Function')
    def agg(file,groupby, aplyname, func):
        """Применяет функцию к столбцу группировки в CSV-файле
```

Пример использования:

```
./cvscli.py cvsops --file ext/input.csv -groupby\
last_name --aplyname count --func npmedian
Processing csvfile: ext/input.csv and groupby name:\
last_name and aplyname: count
2017-06-22 14:07:52,532 - nlib.utils - INFO - \
Loading appliable functions/plugins: npmedian
2017-06-22 14:07:52,533 - nlib.utils - INFO - \
Loading appliable functions/plugins: npsum
```

```

2017-06-22 14:07:52,533 - nlib.utils - INFO - \
Loading applicable functions/plugins: numpy
2017-06-22 14:07:52,533 - nlib.utils - INFO - \
Loading applicable functions/plugins: tanimoto
last_name
eagle    17.0
lee      3.0
smith    13.5
Name: count, dtype: float64

"""
Если не file, и не groupby, и не applyname, и не func:
click.echo("--file and --column and --applyname\
--func are required")
sys.exit(1)

click.echo("Processing csvfile: {file} and groupby name:\
{groupby} and applyname: {applyname}").\
format(file=file, groupby=groupby, applyname=applyname))
#Загрузка плагинов и выбор нужного
plugins = utils.plugins_map()
applicable_func = plugins[func]
res = csvops.group_by_operations(data=file,
groupby_column_name=groupby, apply_column_name=applyname,
func=applicable_func)
click.echo(res)

```

Наконец, подобно веб-API, утилита командной строки предоставляет возможность вывода списка доступных плагинов:

```

@cli.command("listfuncs")
def listfuncs():
    """Выводит список функций, которые можно использовать
    для операции группировки
    Пример использования:

    ./csvcli.py listfuncs
    Доступные функции: ['nmedian', 'npsum', 'numpy', 'tanimoto']
    """

    funcs = utils.applicable_functions()
    click.echo("Applicable Functions: {funcs}".format(funcs=funcs))

if __name__ == "__main__":
    cli()

```

Оценка производительности и тестирование API

При создании API для реальной промышленной эксплуатации было бы в корне неверно не оценить его производительность до начала работы. Вот так это можно сделать с помощью команды сборочного файла:

```
→ pai-aws git:(master) make benchmark-web-sum
#Простейшая оценка производительности API для операции суммирования
ab -n 1000 -c 100 -T 'application/json' -u ext/input_base64.txt\
http://0.0.0.0:5001/api/npsum?column=count&group_by=last_name
This is ApacheBench, Version 2.3 <$Revision: 1757674 $>
```

```
.....
Benchmarking 0.0.0.0 (be patient)
Completed 100 requests
Finished 1000 requests
```

```
Server Software: Werkzeug/0.14.1
Server Hostname: 0.0.0.0
Server Port: 5001
```

```
Document Path: /api/npsum?column=count&group_by=last_name
Document Length: 31 bytes
```

```
Concurrency Level: 100
Time taken for tests: 4.105 seconds
Complete requests: 1000
Failed requests: 0
Total transferred: 185000 bytes
Total body sent: 304000
HTML transferred: 31000 bytes
Requests per second: 243.60 [#/sec] (mean)
Time per request: 410.510 [ms] (mean)
```

В данном случае производительность приложения вполне удовлетворительна, как и его масштабируемость при использовании адаптивного балансировщика нагрузки (elastic load balancer, ELB) с несколькими узлами Nginx. Однако это все же пример того, что Python, будучи прекрасным языком написания кода с большими возможностями, все же сильно уступает в смысле производительности таким языкам, как C++, Java, C# и Go. Приложения, написанные на Erlang или Go, зачастую выполняют подобные функции, получая тысячи запросов в секунду.

В данном случае, однако, скорость работы созданного приложения и конкретный сценарий использования соотносятся вполне оптимально. В числе идей

возможных доработок в версии 2 — переключение на AWS Chalice и применение Spark и/или Redis для кэширования запросов и сохранения результатов в оперативной памяти. Обратите внимание, что в AWS Chalice есть возможность кэширования запросов API по умолчанию, благодаря чему добавление дополнительных слоев кэширования не представляет сложности.

Идеи относительно развертывания в EFS

Для развертывания созданного приложения в среде промышленной эксплуатации нам нужен только сервер сборки, а для организации нескольких точек подключения EFS одна — для среды разработки, другая — для промышленной эксплуатации и т. д. При выпуске версии кода в определенную ветку задание удаленно синхронизирует (rsync) его с нужной точкой подключения. В качестве трюка, благодаря которому код будет «знать» местоположение нужной среды, можно воспользоваться именем EFS как способом маршрутизации. Ниже показано, как реализация этого могла бы выглядеть в файле `env.py`.

С помощью нестандартного использования команды `df` операционной системы Linux код может удостовериться, что запущен в нужном месте. В качестве дальнейшего усовершенствования можно хранить данные о среде в хранилище параметров сервиса администрирования AWS (<https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html>).

"""

Код переключения между средами:

Предполагается, что EFS, по сути, является ключом отображения

"""

```
from subprocess import Popen, PIPE

ENV = {
    "local": {"file_system_id": "fs-999BOGUS", \
             "tools_path": "."}, #used for testing
    "dev": {"file_system_id": "fs-203cc189"},
    "prod": {"file_system_id": "fs-75bc4edc"}
}

def df():
    """Получаем результаты команды df"""

    p = Popen('df', stdin=PIPE, stdout=PIPE, stderr=PIPE)
```

```

output, err = p.communicate()
rc = p.returncode
if rc == 0:
    return output
return rc, err

def get_amazon_path(dfout):
    """Извлекаем путь Amazon из точки подключения"""

    for line in dfout.split():
        if "amazonaws" in line:
            return line
    return False

def get_env_efs_id(local=False):
    """Выполняем синтаксический разбор результатов df для получения
    среды и идентификатора EFS"""

    if local:
        return ("local", ENV["local"]["file_system_id"])
    dfout = df()
    path = get_amazon_path(dfout)
    for key, value in ENV.items():
        env = key
        efsid = value["file_system_id"]
        if path:
            if efsid in path:
                return (env, efsid)
    return False

def main():
    env, efsid = get_env_efs_id()
    print "ENVIRONMENT: %s | EFS_ID: %s" % (env, efsid)

if __name__ == '__main__':
    main()

```

Резюме

AWS — весьма разумный выбор в качестве фундамента технологических решений компании. Достаточно посмотреть на рыночную капитализацию компании Amazon, чтобы понять, что она еще довольно долго будет вводить новшества и снижать цены. Достижения Amazon в области бессерверных технологий весьма впечатляют.

Не волнуйтесь о возможной подчиненности определенному поставщику сервисов, ведь использование Erlang или DigitalOcean в ЦОД не означает зависимости от поставщика. Истинная зависимость здесь — от маленькой команды разработчиков со своими странностями и системных администраторов.

В данной главе были продемонстрированы вполне реалистичные API и решения, в основе которых лежат задачи, выполненные мной при консультациях по AWS. Многие идеи из других глав этой книги можно связать с идеями настоящей главы, и такое их содружество вполне может превратиться в решение для промышленной эксплуатации.

Часть III

Создание реальных
приложений ИИ
с нуля

6

Прогноз популярности в соцсетях в НБА

Гении выигрывают отдельные игры, но чемпионаты выигрывают командная работа и интеллект.

*Майкл Джордан
(Michael Jordan)*

Спорт — исключительно интересная тема для исследователей данных, поскольку за каждым показателем скрывается своя история. То, что игрок НБА набирает больше очков, чем другой, отнюдь не значит, что его ценность для команды выше. В результате в последние годы наблюдается бурный рост в сфере индивидуальной статистики игроков, направленной на оценку вклада игрока. Канал ESPN создал систему статистики Real Plus-Minus, сайт FiveThirtyEight публикует прогнозы выступлений игроков НБА (CARMELLO NBA Player Projections), а НБА применяет оценку вклада игроков (Player Impact Estimate). Соцсети ничем не различаются, и дело отнюдь не только в большом числе подписчиков.

В этой главе мы выясним, какие сведения скрываются за показателями, с помощью машинного обучения и последующего создания API для выдачи модели машинного обучения. Причем делать это мы будем в духе решения реальных задач реальными способами. А значит, охватим такие нюансы, как настройка среды, развертывание и мониторинг, а не только создание моделей на основе очищенных данных.

Постановка задачи

При первом взгляде на социальные медиа и НБА возникает множество интересных вопросов. Например:

- влияют ли результаты отдельного игрока на победы команды;
- коррелируют ли результаты игрока на игровой площадке с его влиятельностью в соцсетях;
- коррелирует ли присутствие игрока в соцсетях с популярностью в «Википедии»;
- что является лучшим прогностическим параметром популярности в Twitter: число подписчиков или вовлеченность в социальные медиа;
- коррелирует ли заработная плата игрока с его игровыми результатами;
- увеличивают ли победы команды количество болельщиков на трибунах;
- что сильнее влияет на стоимость команды: посещаемость игр или локальный рынок недвижимости?

Для получения ответов на эти и другие вопросы необходимо собрать данные. Здесь работает правило 80/20, когда 80 % решения задачи составляют сбор и преобразование данных, остальные 20 % — задачи, связанные с машинным обучением и исследованием данных: поиск правильной модели, выполнение EDA (exploratory data analysis — разведочный анализ данных) и проектирование признаков.

Сбор данных

На рис. 6.1 представлен список источников данных для извлечения и преобразования.

Сбор подобных данных представляет собой нетривиальную задачу инженерии разработки ПО. Для этого необходимо преодолеть множество препятствий, например найти хороший источник данных, написать код для их извлечения с учетом ограничений API и, наконец, получить данные в нужной форме. Первый шаг при сборе данных — выяснить, с каких источников начать и где их найти.

С учетом нашей конечной цели — сравнения популярности в социальных медиа с результатами игроков НБА — лучше всего начать с реестра игроков НБА за сезон 2016–2017. Теоретически это несложная задача, но при сборе данных НБА можно попасть в несколько ловушек. Интуитивно кажется, что имеет смысл начать с официального сайта НБА — nba.com. По каким-то причинам, однако, спортивные лиги делают затруднительным скачивание необработанных данных со своих сайтов. НБА не исключение, скачивание статистики с их официального веб-сайта — задача выполнимая, но непростая.



Рис. 6.1. Источники данных о социальном авторитете в НБА

Это приводит к интересному соображению по поводу способа сбора данных. Зачастую проще всего собрать их вручную, то есть скачать с сайта и очистить в Excel, блокноте Jupiter или RStudio. Это вполне разумный начальный подход к задаче исследования данных. Однако если сбор данных из одного источника и их очистка занимают несколько часов, то лучше написать код

для решения такой задачи. Жестких правил тут нет, но опытные исследователи знают, как постепенно продвигаться в решении задачи, не застопориваясь по пути.

Первые источники данных

Вместо того чтобы начинать с непростого источника данных вроде официального веб-сайта НБА, активно мешающего скачивать с него данные, мы начнем с относительно удобного источника. Наш первый источник данных по баскетболу вы можете скачать непосредственно из проекта GitHub для этой книги (<https://github.com/noahgift/pragmaticai>) или с сайта Basketball Reference (https://www.basketball-reference.com/leagues/NBA_2017_per_game.html).

Для машинного обучения на практике недостаточно выбора правильной модели для очищенных данных, нужно еще и знать, как настроить локальную среду.

Для запуска кода необходимо выполнить несколько шагов.

1. Создать виртуальную среду (на основе Python 3.6).
2. Установить несколько пакетов, которые понадобятся нам в этой главе: Pandas, Jupiter.
3. Запустить все это с помощью сборочного файла.

В листинге 6.1 приведена команда, создающая виртуальную среду для Python 3.6 и устанавливающая пакеты, перечисленные в файле `requirements.txt` в листинге 6.2. Эти действия можно выполнить за один раз с помощью следующей однострочной команды:

```
make setup && install
```

Листинг 6.1. Содержимое сборочного файла

```
setup:
    python3 -m venv ~/.pragai6
install:
    pip install -r requirements.txt
```

Листинг 6.2. Содержимое файла requirements.txt

```
pytest
nbval
```

ipython
requests
python-twitter
pandas
pylint
sensible
jupyter
matplotlib
seaborn
statsmodels
sklearn
wikipedia
spacy
ggplot



Удобный прием при работе с виртуальными средами Python — создание псевдонима в файле `.bashrc` или `.zshrc`, который за одну операцию автоматически делает активной нужную среду и переходит в соответствующий каталог. Обычно я делаю это с помощью следующего фрагмента кода:

```
alias pragaibtop="cd ~/src/pragai/chapter6\  
&& source ~/. Pragaib /bin/activate"
```

Для работы над проектом из данной главы просто наберите в командной строке `pragaibtop`, и вы попадете в каталог с кодом нужного проекта и запустите виртуальную среду. Вся мощь псевдонимов командной оболочки в действии. Есть и другие утилиты для этой цели, например `pipenv`, имеет смысл взглянуть и на них.

Для просмотра данных запустите блокнот Jupiter с помощью команды `jupyter notebook`. При этом запустится браузер, где вы сможете просматривать существующие блокноты или создавать новые. Если вы заглянете в исходный код проекта GitHub для этой главы, то найдете файл `basketball_reference.ipynb`.

Он представляет собой простой, типа «Hello, world!», блокнот с загруженными в него данными. Загрузка данных в блокнот Jupiter или, в случае языка R, RStudio — обычно наиболее удобный способ первоначальной проверки и просмотра набора данных. Листинг 6.3 демонстрирует также, как просмотреть данные из обычной командной оболочки IPython дополнительно к Jupiter или вместо него.

Листинг 6.3. Изучение данных с сайта Basketball Reference из блокнота Jupiter

```
import pandas as pd
nba = pd.read_csv("data/nba_2017_br.csv")
nba.describe()
```



Еще один способ убедиться в работоспособности блокнотов Jupiter — использовать плагин nbval для pytest. Вы можете добавить команду `test` сборочного файла для запуска всех блокнотов следующим образом:

```
make test
```

В следующем фрагменте кода показано, как это будет выглядеть в сборочном файле:

```
test:
    py.test --nbval notebooks/*.ipynb
```

Загрузка CSV-файла в Pandas не представляет сложности, если в нем присутствуют названия столбцов, а длина строк одинакова. При работе с подготовленными наборами данные почти всегда оказываются в подходящей для загрузки форме. На практике же дела обычно обстоят сложнее и привести данные в подходящую форму — непростая задача, как мы увидим далее в этой главе.

Рисунок 6.2 демонстрирует результаты выполнения команды `describe` в блокноте Jupiter. Функция `describe` объектов `DataFrame` библиотеки Pandas возвращает описательную статистику, включая число столбцов (в нашем случае 27) и медиану (50-й процентиль) для каждого столбца. На этой стадии не помешает поэкспериментировать с созданным блокнотом Jupiter и посмотреть, какая еще информация доступна. Однако в данных отсутствует единый показатель для оценки игровых результатов в обороне и атаке. Для его получения нам нужно будет объединить наш набор данных с другими источниками из ESPN и НБА, в результате чего сложность проекта возрастет с простого использования данных до их поиска с последующим преобразованием. Разумным будет воспользоваться утилитой скрапинга, например Scrapy, но в данной ситуации можно применить более простой и узкоспециализированный метод. Например, зайдя на сайты ESPN и НБА, скопировать данные оттуда и вставить их в Excel. Затем можно будет вручную очистить данные и сохранить их в файле CSV. Для небольшого набора

данных такой способ часто требует меньше времени, чем написание сценария для выполнения тех же задач.

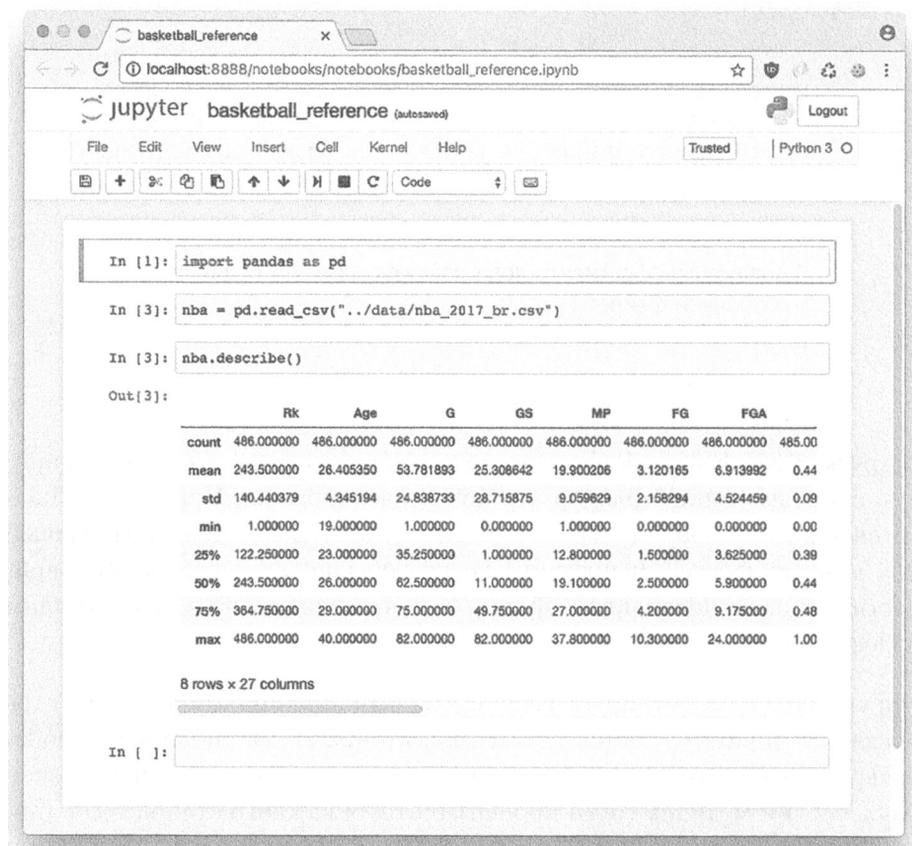


Рис. 6.2. Результаты выполнения команды describe в блокноте Jupyter для объекта DataFrame с данными с сайта Basketball Reference

Но если подобные данные понадобятся в большем проекте, такой подход сыграет нам плохую службу, хотя это один из лучших вариантов для создания прототипа. Ключевая идея для запутанных проектов науки о данных — продвигаться в исследовании, не увязая при этом в слишком большом количестве деталей. Можно потратить уйму времени на автоматизацию запутанного источника данных лишь для того, чтобы позднее понять, что полученная информация бесполезна.

Захват данных с сайта ESPN аналогичен FiveThirtyEight, так что я не стану опять описывать этот процесс здесь. Еще два источника данных — зарплаты и рекламные отчисления. На сайте ESPN есть информация о зарплатах, а «Форбс» предлагает небольшой набор данных о рекламных отчислениях восьми игроков. Таблица 6.1 описывает форму источников данных, а также подытоживает их содержимое. Это весьма впечатляющий список источников данных, полученный в основном вручную.

Таблица 6.1. Источники данных НБА

Источник данных	Имя файла	Строка	Краткое описание
Basketball Reference	nba_2017_attendance.csv	30	Посещаемость стадионов
Forbes	nba_2017_endorsements.csv	8	Игроки с высоким рейтингом
Forbes	nba_2017_team_valuations.csv	30	Все команды
ESPN	nba_2017_salary.csv	450	Большинство игроков
NBA	nba_2017_pie.csv	468	Все игроки
ESPN	nba_2017_real_plus_minus.csv	468	Все игроки
Basketball Reference	nba_2017_br.csv	468	Все игроки
FiveThirtyEight	nba_2017_elo.csv	30	Рейтинг команд
Basketball Reference	nba_2017_attendance.csv	30	Посещаемость стадионов
Forbes	nba_2017_endorsements.csv	8	Игроки с высоким рейтингом
Forbes	nba_2017_team_valuations.csv	30	Все команды
ESPN	nba_2017_salary.csv	450	Большинство игроков

Для получения остальных данных, в основном из Twitter и «Википедии», придется приложить еще немало труда. Среди многообещающих возможностей — изучение рекламных отчислений восьми топ-игроков и исследование стоимости самих команд.

Изучаем наши первые источники данных: команды

Прежде всего нужно создать новый блокнот Jupiter. В репозитории GitHub уже есть готовый, с именем `exploring_team_valuation_nba`. Далее необходимо импортировать набор распространенных библиотек, обычно

используемых при исследовании данных в блокнотах Jupiter. Это показано в листинге 6.4.

Листинг 6.4. Распространенные первоначальные импорты блокнота Jupiter

```
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
%matplotlib inline
```

Далее создадим для каждого источника данных объект DataFrame библиотеки Pandas, как показано в листинге 6.5.

Листинг 6.5. Создание объектов DataFrame для источников

```
attendance_df = pd.read_csv("../data/nba_2017_attendance.csv")
endorsement_df = pd.read_csv("../data/nba_2017_endorsements.csv")
valuations_df = pd.read_csv("../data/nba_2017_team_valuations.csv")
salary_df = pd.read_csv("../data/nba_2017_salary.csv")
pie_df = pd.read_csv("../data/nba_2017_pie.csv")
plus_minus_df = pd.read_csv("../data/nba_2017_real_plus_minus.csv")
br_stats_df = pd.read_csv("../data/nba_2017_br.csv")
elo_df = pd.read_csv("../data/nba_2017_elo.csv")
```

На рис. 6.3 создается цепочка объектов DataFrame — частая практика при сборе данных в реальных условиях.

Объединяем данные о посещаемости с данными о стоимости команд и смотрим на первые несколько строк:

```
In [14]: attendance_valuation_df = \
attendance_df.merge(valuations_df, how="inner", on="TEAM")
```

```
In [15]: attendance_valuation_df.head()
Out[15]:
```

	TEAM	GMS	PCT	TOTAL_MILLIONS	AVG_MILLIONS
0	Chicago Bulls	41	104	0.888882	0.021680
1	Dallas Mavericks	41	103	0.811366	0.019789
2	Sacramento Kings	41	101	0.721928	0.017608
3	Miami Heat	41	100	0.805400	0.019643
4	Toronto Raptors	41	100	0.813050	0.019830

The screenshot shows a Jupyter Notebook interface with the following code and outputs:

```
In [3]: endorsement_df = pd.read_csv("../data/nba_2017_endorsements.csv");endorsement_df.head()
```

```
Out[3]:
```

	NAME	TEAM	SALARY_MILLIONS	ENDORSEMENT_MILLIONS
0	LeBron James	Cleveland Cavaliers	31	55
1	Kevin Durant	Golden State Warriors	27	36
2	Stephen Curry	Golden State Warriors	12	35
3	James Harden	Houston Rockets	27	20
4	Russell Westbrook	Oklahoma City Thunder	27	15

```
In [4]: valuations_df = pd.read_csv("../data/nba_2017_team_valuations.csv");valuations_df.head()
```

```
Out[4]:
```

	TEAM	VALUE MILLIONS
0	New York Knicks	3300.0
1	Los Angeles Lakers	3000.0
2	Golden State Warriors	2800.0
3	Chicago Bulls	2600.0
4	Boston Celtics	2200.0

```
In [5]: salary_df = pd.read_csv("../data/nba_2017_salary.csv");salary_df.head()
```

```
Out[5]:
```

	NAME	POSITION	TEAM	SALARY
0	LeBron James	SF	Cleveland Cavaliers	9083450.0
1	Mike Conley	PG	Memphis Grizzlies	26540100.0
2	Al Horford	C	Boston Celtics	26540100.0
3	Dirk Nowitzki	PF	Dallas Mavericks	25000000.0
4	Carmelo Anthony	SF	New York Knicks	24566980.0

```
In [6]: pie_df = pd.read_csv("../data/nba_2017_pie.csv");pie_df.head()
```

```
Out[6]:
```

	PLAYER	TEAM	AGE	GP	W	L	MIN	OFFRTG	DEFRTG	NETRTG	...	AST	RATIO	ORB%	DREB%	REB%	TO RATIO	EPG%	TS%	USG%	PACE	PIE
0	Russell Westbrook	OKC	28	81	46	25	34.6	107.9	104.6	3.9	...	23.4	5.3	27.9	16.7	12.2	47.6	55.4	40.8	102.31	23.0	
1	Boban Marjanovic	DET	28	35	16	19	8.4	104.3	102.4	1.9	...	5.1	16.6	31.3	23.9	5.7	54.5	60.6	24.8	97.20	19.6	
2	Demetrius Jackson	BOS	22	5	1	4	3.4	124.2	117.3	6.3	...	31.1	9.1	11.8	10.3	0.0	67.5	75.3	17.2	87.46	18.4	
3	Anthony Davis	NOP	24	75	31	44	36.1	104.2	102.5	1.7	...	7.3	6.7	26.9	17.0	8.4	51.8	58.0	32.6	100.19	18.2	
4	James Harden	HOU	27	61	54	27	36.4	113.6	107.3	6.3	...	27.6	3.5	21.2	12.3	14.1	52.5	61.3	34.1	102.98	19.0	

5 rows x 22 columns

Рис. 6.3. Результаты вывода нескольких объектов DataFrame в Jupyter

Воспользуемся функцией `pairplot` библиотеки `Seaborn` для построения показанной на рис. 6.4 матрицы диаграмм рассеяния.

```
In [15]: from IPython.core.display import display, HTML
...: display(HTML("<style>.\
container{ width:100% !important; }</style>"));
sns.pairplot(attendance_valuation_
...: df, hue="TEAM")
```

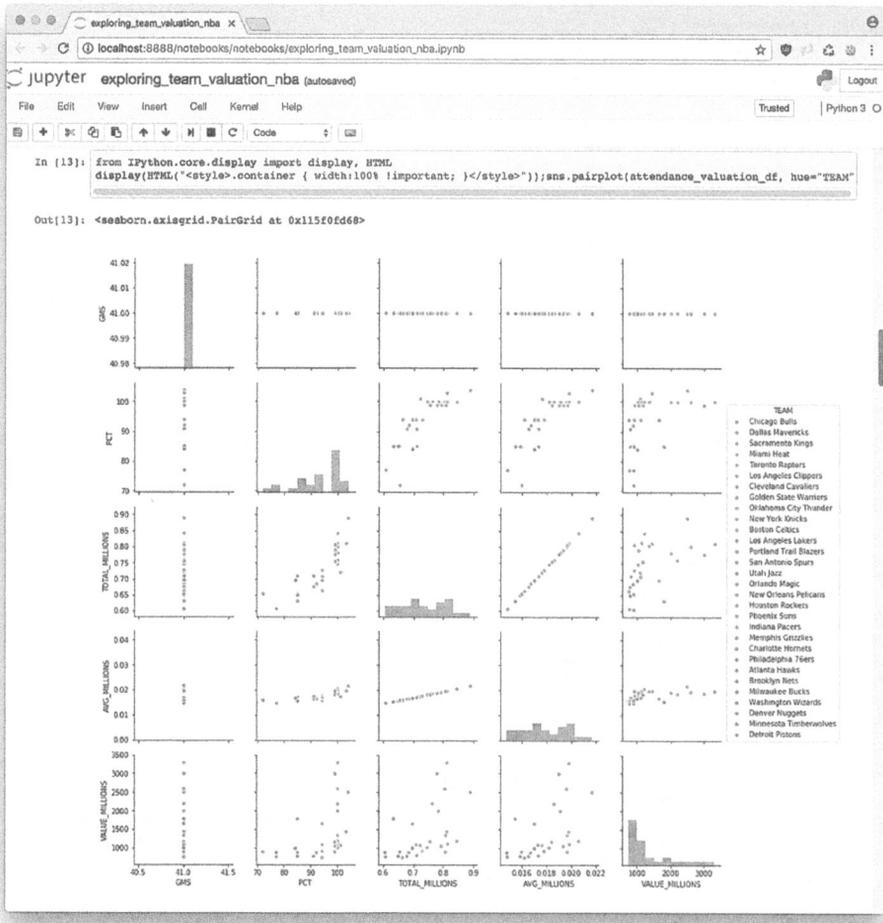


Рис. 6.4. Матрица диаграмм рассеяния посещаемости/стоимости команд

Судя по графикам, между посещаемостью (средней или общей) и стоимостью команды существует взаимосвязь. Чтобы выяснить это, можно воспользоваться картой интенсивности корреляции, показанной на рис. 6.5.

```
In [16]: corr = attendance_valuation_df.corr()
...: sns.heatmap(corr,
...:               xticklabels=corr.columns.values,
...:               yticklabels=corr.columns.values)
...:
```

Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x111007ac8>

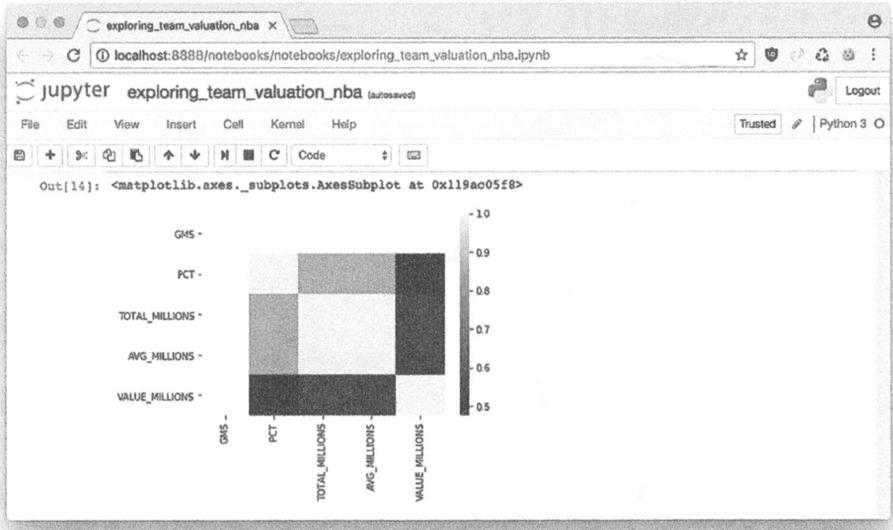


Рис. 6.5. Карта интенсивности корреляции посещаемости/стоимости команд

Взаимосвязь, заметную на матрице диаграмм рассеяния, теперь можно оценить количественно. Карта интенсивности показывает среднюю корреляцию между стоимостью команд и посещаемостью, близкую к 50%. Еще одна карта интенсивности демонстрирует среднюю посещаемость относительно стоимости каждой из команд НБА. Для генерации подобной карты интенсивности в Seaborn необходимо сначала преобразовать данные в сводную таблицу. Полученный в результате график показан на рис. 6.5.

```
In [18]: valuations = attendance_valuation_df.\
pivot("TEAM", "TOTAL_MILLIONS", "VALUE_MILLIONS")
In [19]: plt.subplots(figsize=(20,15))
...: ax = plt.axes()
...: ax.set_title("NBA Team AVG Attendance vs\
Valuation in Millions: 2016-2017 Season")
...: sns.heatmap(valuations,linewidths=.5, annot=True, fmt='g')
...:
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x114d3d080>
```

Карта интенсивности на рис. 6.6 указывает на потенциальное существование интересных паттернов, для изучения которых имеет смысл построить дополнительные графики, возможно трехмерные. В Нью-Йорке и Лос-Анджелесе наблюдаются аномальные значения.

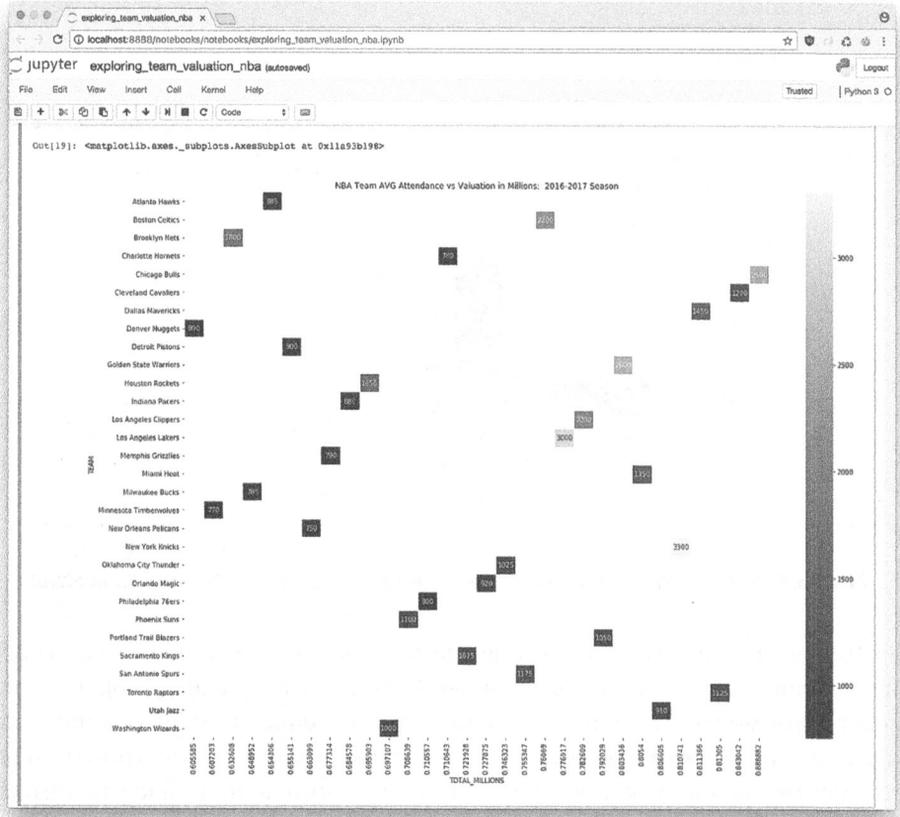


Рис. 6.6. Посещаемость матчей команд НБА относительно карты интенсивности для стоимости команд

Изучаем наши первые источники данных с помощью регрессии

Рисунок 6.6 демонстрирует несколько весьма интригующих аномальных значений, например, клуб «Бруклин Нетс» (Brooklyn Nets) оценивается в \$1,8 млрд, однако посещаемость их матчей — одна из самых низких в НБА. Здесь происходит что-то заслуживающее более пристального внимания. Один из методов дальнейшего исследования — воспользоваться линейной регрессией, чтобы попытаться объяснить эту взаимосвязь. Если рассматривать вариант использования как Python, так и языка R, то сделать это можно несколькими способами. В Python чаще всего применяются два

подхода: с помощью пакета StatsModels и библиотеки scikit-learn. Рассмотрим оба.

В случае пакета StatsModels мы получаем обширные диагностические данные о производимой линейной регрессии, работа с ним похожа на работу с классическим ПО для линейной регрессии вроде Minitab и R.

```
In [24]: results = smf.ols(
        'VALUE_MILLIONS ~TOTAL_MILLIONS',
        data=attendance_valuation_df).fit()
```

```
In [25]: print(results.summary())
```

OLS Regression Results

```
=====
Dep. Variable:          VALUE_MILLIONS    R-squared:          0.282
Model:                  OLS              Adj. R-squared:     0.256
Method:                 Least Squares    F-statistic:        10.98
Date:                   Thu, 10 Aug 2017  Prob (F-statistic):0.00255
Time:                   14:21:16         Log-Likelihood:     -234.04
No. Observations:      30              AIC:                472.1
Df Residuals:           28              BIC:                474.9
Df Model:                1
Covariance Type:       nonrobust
=====
```

```
=====
                    coef    std err          t  P>|t|[0.025 0.975]
-----
```

```
.....
```

Warnings:

```
[1] Standard Errors assume that the covariance matrix of the errors
is correctly specified.
```

Судя по результатам регрессии, переменная TOTAL_MILLIONS (общая посещаемость в миллионах) статистически значима (значение P меньше 0,05) для предсказания изменений посещаемости. Равный 0,282 (то есть 28 %) коэффициент детерминации (R -квадрат) демонстрирует, насколько хорошо произведен подбор параметров, то есть насколько прямая регрессии соответствует данным.

Чтобы выяснить, в какой степени эта модель может предсказывать значения, построим дополнительные графики и проведем дополнительную диагностику. В пакете Seaborn есть встроенная и очень удобная функция `residplot`, предназначенная для построения графиков остаточных погрешностей. Ее использование показано на рис. 6.7. Идеальный вариант: случайно

распределенные остаточные погрешности; наличие на графике каких-либо паттернов может свидетельствовать о проблемах с моделью. Данный пример отнюдь не похож на равномерно случайное распределение:

```
In [88]: sns.residplot(y="VALUE_MILLIONS", x="TOTAL_MILLIONS",
...: data=attendance_valuation_df)
...:
Out[88]: <matplotlib.axes._subplots.AxesSubplot at 0x114d3d080>
```

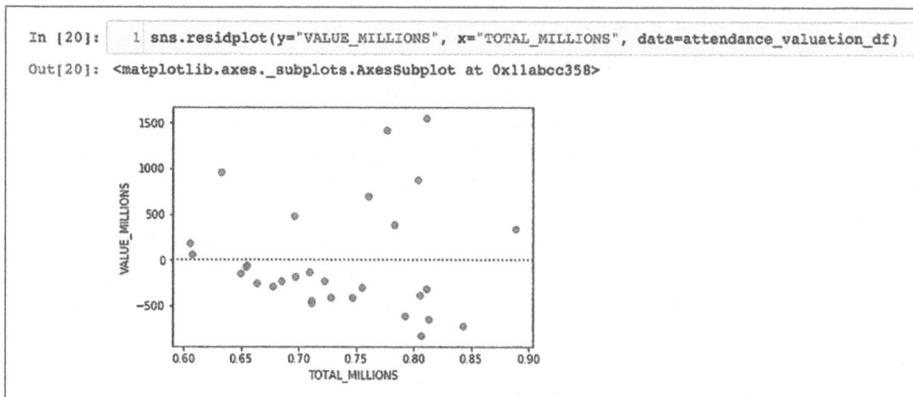


Рис. 6.7. Посещаемость матчей команд НБА и график остаточных погрешностей стоимости команд

Для оценки точности предсказания с применением машинного обучения часто используют среднеквадратическую погрешность (root mean squared error, RMSE). Сделать это с помощью пакета StatsModels можно следующим образом:

```
In [92]: import statsmodels
...: rmse = statsmodels.tools.eval_measures.rmse(
...: attendance_valuation_predictions_df["predicted"],
...: attendance_valuation_predict
...: ions_df["VALUE_MILLIONS"])
...: rmse
...:
Out[92]: 591.33219017442696
```

Чем меньше RMSE, тем лучше предсказание. Для повышения точности предсказания необходимо найти способ снизить RMSE. Кроме того, боль-

ший набор данных, такой, чтобы можно было разделить его на обучающую и контрольную последовательности, позволяет повысить точность и снизить вероятность переобучения. Следующий этап диагностики — построение графика предсказанных линейной регрессией значений относительно настоящих. На рис. 6.8 показаны результаты выполнения функции `lmlplot` для предсказанных и настоящих значений, из которых очевидно, что модель предсказывает значения не так уж хорошо. Однако для начала и это неплохо, именно так часто модели машинного обучения и создаются — посредством обнаружения корреляций и/или статистически значимых взаимосвязей, а затем — принятия решения о том, что имеет смысл собрать дополнительные данные.

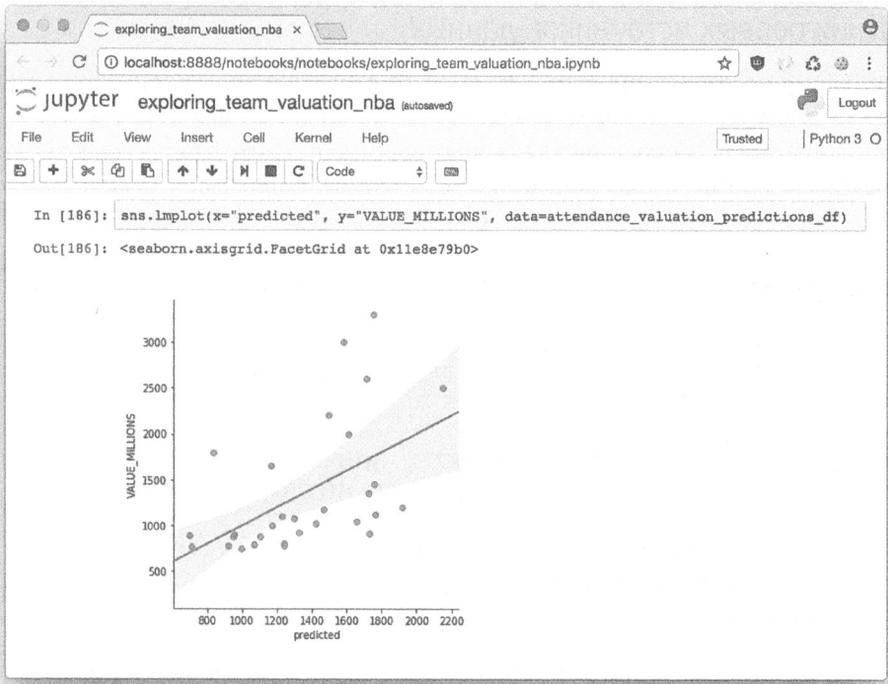


Рис. 6.8. График стоимости команды: предсказанные и настоящие значения

Первоначальный вывод: хотя взаимосвязь между посещаемостью и стоимостью команды НБА существует, здесь есть и отсутствующие (латентные) переменные. Возникает предположение, что здесь могут играть роль

население области, медианные цены на недвижимость и сила команды (рейтинг ЭЛО и процент побед):

```
In [89]: attendance_valuation_predictions_df = \
attendance_valuation_df.copy()
```

```
In [90]: attendance_valuation_predictions_df["predicted"] = \
results.predict()
```

```
In [91]: sns.lmplot(x="predicted", y="VALUE_MILLIONS", \
data=attendance_valuation_predictions_df)
```

```
Out[91]: <seaborn.axisgrid.FacetGrid at 0x1178d2198>
```

Машинное обучение без учителя: кластеризация наших первых источников данных

Чтобы узнать больше про команды НБА, нужно начать с кластеризации данных с помощью машинного обучения без учителя. Мне удалось вручную найти медианные цены на дома для разных округов на сайте <https://www.zillow.com/research/> и данные о населении каждого округа в переписи населения на сайте <https://www.census.gov/data/tables/2016/demo/popest/counties-total.html>.

Загрузим все эти данные в новый объект DataFrame:

```
In [99]: val_housing_win_df =
pd.read_csv("../data/nba_2017_att_val_elo_win_housing.csv")
```

```
In [100]: val_housing_win_df.columns
```

```
Out[100]:
```

```
Index(['TEAM', 'GMS', 'PCT_ATTENDANCE', 'WINNING_SEASON',
       'TOTAL_ATTENDANCE_MILLIONS', 'VALUE_MILLIONS',
       'ELO', 'CONF', 'COUNTY',
       'MEDIAN_HOME_PRICE_COUNTY_MILLIONS',
       'COUNTY_POPULATION_MILLIONS'],
      dtype='object')
```

Кластеризация методом *k*-ближайших соседей производится путем вычисления евклидова расстояния между точками. При этом необходимо нормировать кластеризуемые атрибуты так, чтобы их масштаб не различался, ведь это может исказить кластеризацию. Кроме того, кластеризация — более искусство, чем ремесло, и выбор правильного числа кластеров обычно производится методом проб и ошибок. На практике нормирование осуществляется следующим образом.

```
In [102]: numerical_df = val_housing_win_df.loc[:,\
["TOTAL_ATTENDANCE_MILLIONS", "ELO", "VALUE_MILLIONS",
 "MEDIAN_HOME_PRICE_COUNT
...: Y_MILLIONS"]]
```

```
In [103]: from sklearn.preprocessing import MinMaxScaler
...: scaler = MinMaxScaler()
...: print(scaler.fit(numerical_df))
...: print(scaler.transform(numerical_df))
```

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

```
[[ 1.          0.41898148  0.68627451  0.08776879]
 [ 0.72637903  0.18981481  0.2745098   0.11603661]
 [ 0.41067502  0.12731481  0.12745098  0.13419221]...
```

В этом примере используется функция `MinMaxScaler` из библиотеки `scikit-learn`. Она преобразует все числовые значения к значениям, находящимся между 0 и 1. Далее для кластеризации нормированных данных мы воспользуемся модулем `sklearn.cluster`, после чего присоединим новый столбец с результатами кластеризации:

```
In [104]: from sklearn.cluster import KMeans
...: k_means = KMeans(n_clusters=3)
...: kmeans = k_means.fit(scaler.transform(numerical_df))
...: val_housing_win_df['cluster'] = kmeans.labels_
...: val_housing_win_df.head()
...:
```

```
Out[104]:
```

	TEAM	GMS	PCT_ATTENDANCE	WINNING_SEASON	\
0	Chicago Bulls	41	104	1	
1	Dallas Mavericks	41	103	0	
2	Sacramento Kings	41	101	0	
3	Miami Heat	41	100	1	
4	Toronto Raptors	41	100	1	
	TOTAL_ATTENDANCE_MILLIONS	VALUE_MILLIONS	ELO	CONF	
0	0.888882	2500	1519	East	
1	0.811366	1450	1420	West	
2	0.721928	1075	1393	West	
3	0.805400	1350	1569	East	
4	0.813050	1125	1600	East	
	MEDIAN_HOME_PRICE_COUNTY_MILLIONS	cluster			
0	269900.0	1			
1	314990.0	1			
2	343950.0	0			
3	389000.0	1			
4	390000.0	1			

На этом этапе программное решение уже достаточно развито, чтобы представлять для компании ценность, а конвейер данных начинает формироваться. Далее давайте воспользуемся языком R и пакетом `ggplot` для построения графика кластеров. Чтобы переместить наш набор данных в язык R, запишем его в CSV-файл:

```
In [105]: val_housing_win_df.to_csv(
"../data/nba_2017_att_val_elo_win_housing_cluster.csv"
)
```

Построение 3D-графика кластеризации методом k-ближайших соседей с помощью языка R

Одно из достоинств языка R — возможность построения продвинутых графиков с осмысленным текстом. Умение писать программные решения на языках R и Python открывает массу возможностей машинного обучения. В данной конкретной ситуации мы воспользуемся библиотекой трехмерных диаграмм рассеяния языка R и пакетом RStudio для создания сложного графика взаимосвязей, о которых мы узнали из кластеризации методом k-ближайших соседей. В проекте GitHub для этой главы вы найдете блокнот на языке разметки R, содержащий код и график. Можете также следить за ходом примеров посредством имеющейся у RStudio возможности предварительного просмотра блокнотов.

Для начала работы в консоли в RStudio (или командной оболочке R) импортируем библиотеку `scatterplot3d` и загрузим данные с помощью следующих команд:

```
> library("scatterplot3d",
  lib.loc="/Library/Frameworks/R.framework/\
  Versions/3.4/Resources/library")
> team_cluster <- read_csv("~/src/aibook/src/chapter7/data/\
  nba_2017_att_val_elo_win_housing_cluster.csv",
+   col_types = cols(x1 = col_skip()))
```

Далее создадим функцию для преобразования типов данных в формат, ожидаемый библиотекой `scatterplot3d`:

```
> cluster_to_numeric <- function(column){
+   converted_column <- as.numeric(unlist(column))
+   return(converted_column)
+ }
```

Для хранения данных о цвете кластеров создаем новый столбец:

```
> team_cluster$pcolor[team_cluster$cluster == 0] <- "red"
> team_cluster$pcolor[team_cluster$cluster == 1] <- "blue"
> team_cluster$pcolor[team_cluster$cluster == 2] <- "darkgreen"
```

Строим черновой 3D-график.

```
> s3d <- scatterplot3d(
+   cluster_to_numeric(team_cluster["VALUE_MILLIONS"]),
+   cluster_to_numeric(
+     team_cluster["MEDIAN_HOME_PRICE_COUNTY_MILLIONS"]),
+   cluster_to_numeric(team_cluster["ELO"]),
+   color = team_cluster$pcolor,
+   pch=19,
+   type="h",
+   lty.hplot=2,
+   main="3-D Scatterplot NBA Teams 2016-2017:
+     Value, Performance, Home Prices with kNN Clustering",
+   zlab="Team Performance (ELO)",
+   xlab="Value of Team in Millions",
+   ylab="Median Home Price County Millions"
+ )
>
```

Для вывода текста в нужном месте 3D-пространства придется немного потрудиться:

```
s3d.coords <- s3d$xyz.convert(
  cluster_to_numeric(team_cluster["VALUE_MILLIONS"]),
  cluster_to_numeric(
    team_cluster["MEDIAN_HOME_PRICE_COUNTY_MILLIONS"]),
  cluster_to_numeric(team_cluster["ELO"]))
```

```
#Выводим текст на график
text(s3d.coords$x, s3d.coords$y,      # Координаты x и y
     labels=team_cluster$TEAM,        # текст
     pos=4, cex=.6)                  # сжатие текста
```

В приведенном на рис. 6.9 графике видны некоторые необычные паттерны. «Нью-Йорк Никс» (New York Knicks) и «Лос-Анджелес Лейкерс» (Los Angeles Lakers) — две худших команды в НБА, однако они же и самые дорогие. Кроме того, как можно видеть, они расположены в городах с одними из самых высоких медианных цен на дома, что играет свою роль в их большой стоимости. В силу этих причин они выделены в отдельный кластер.

Кластер 1 представляет собой в основном набор лучших команд НБА. Кроме того, эти команды обычно из городов с более высокими медианными ценами на недвижимость при более широком диапазоне фактической стоимости. Это вызывает подозрения, что цены на недвижимость существенно для стоимости команды, чем ее игровые результаты (что согласуется с итогами проведенной ранее линейной регрессии).

Кластер 0 содержит команды из городов с медианными ценами на недвижимость ниже среднего. Их игровые результаты, а равно и их стоимость, также обычно ниже среднего.

В языке R есть еще один способ многомерной визуализации этих взаимосвязей, так что теперь мы займемся построением графика в языке R с помощью пакета ggplot.

Прежде всего при визуализации взаимосвязи на новом графике необходимо задать подходящие имена для кластеров. Неплохие идеи о том, как назвать кластеры, можно почерпнуть из вышеприведенного 3D-графика. Кластер 0, похоже, представляет собой кластер низкой стоимости/низких игровых результатов, кластер 1 — кластер средней стоимости/высоких игровых результатов, а кластер 2 — кластер высокой стоимости/низких игровых результатов. Следует также отметить, что выбор номера кластера — непростой вопрос (более полную информацию на этот счет можно найти в приложении Б).

```
> team_cluster <- read_csv("nba_cluster.csv",
+                           col_types = cols(x1 = col_skip()))
> library("ggplot2")
>
> #Названия кластеров
> team_cluster$cluster_name[team_cluster$cluster == 0] <- "Low"
Unknown or uninitialised column: 'cluster_name'.
> team_cluster$cluster_name[team_cluster$
+   cluster == 1] <- "Medium Valuation/High Performance"
> team_cluster$cluster_name[team_cluster$
+   cluster == 2] <- "High Valuation/Low Performance"
```

Далее эти названия кластеров можно применять для создания фасетных графиков (по несколько графиков в одном). Кроме того, в библиотеке ggplot есть возможности создания большого количества дополнительных

измерений, и мы воспользуемся ими всеми: цветом для отображения доли побеждающих и проигрывающих в сезоне команд, размером, чтобы показать различия в медианных ценах на недвижимость в округах, и формой в качестве показателя, к какой конференции НБА относится команда (Западной или Восточной):

```
> p <- ggplot(data = team_cluster) +
+   geom_point(mapping = aes(x = ELO,
+                             y = VALUE_MILLIONS,
+                             color =
+                               factor(WINNING_SEASON, labels=
+                                 c("LOSING", "WINNING")),
+                             size = MEDIAN_HOME_PRICE_COUNTY_MILLIONS,
+                             shape = CONF)) +
+   facet_wrap(~ cluster_name) +
+   ggtitle("NBA Teams 2016-2017 Faceted Plot") +
+   ylab("Value NBA Team in Millions") +
+   xlab("Relative Team Performance (ELO)") +
+   geom_text(aes(x = ELO, y = VALUE_MILLIONS,
+                 label=ifelse(VALUE_MILLIONS>1200,
+                               as.character(TEAM), ' ')),hjust=.35,vjust=1)
```

Обратите внимание, что функция `geom_text` выводит имя команды только в том случае, если ее стоимость выше \$1200 млн. Благодаря этому график становится более удобочитаемым и не перегруженным перекрывающимися надписями. В последнем фрагменте кода меняются названия из легенды. Замечу, что цвет также меняется на одно из двух значений вместо четырех по умолчанию (0, 0,25, 0,50, 1). Результат построения графика показан на рис. 6.10. Возможности создания фасетных графиков библиотеки `ggplot` наглядно демонстрируют повышение эффективности исследования данных с помощью кластеризации. Использовать язык R для построения графиков — хорошая идея, даже если вы специализируетесь на другом языке машинного обучения, например Python или Scala. Результаты говорят сами за себя:

#Меняем легенды

```
p +
  guides(color = guide_legend(title = "Winning Season")) +
  guides(size = guide_legend(
+   title = "Median Home Price County in Millions" )) +
  guides(shape = guide_legend(title = "NBA Conference"))
```

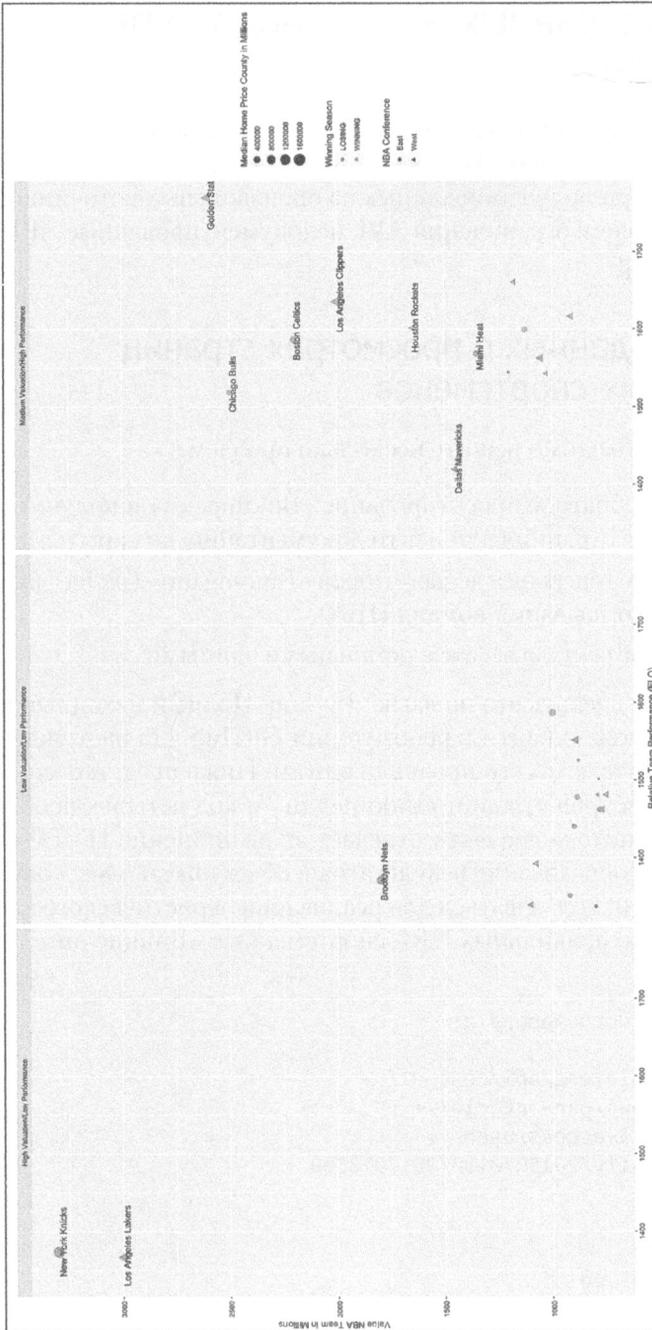


Рис. 6.10. Фасетный график для команд НБА за 2016–2017 гг. на основе кластеризации методом k-ближайших соседей

Получение данных из труднодоступных источников

Собрав основные данные о командах НБА, можно перейти и к более труднодоступным источникам. На этом этапе задача становится более схожей с реальными. При получении данных из произвольных источников возникает масса проблем: ограничения API, недокументированные API, данные с ошибками и др.

Получение данных о просмотрах страниц «Википедии» спортсменов

Для начала необходимо решить несколько проблем.

1. Выполнить обратное проектирование «Википедии» и получить данные о просмотрах страниц (или найти документацию по скрытым API).
2. Найти способ генерации дескрипторов «Википедии» (их названия могут отличаться от названий команд НБА).
3. Соединить объект `DataFrame` с остальными данными.

Расскажем, как сделать это на языке Python. Полный исходный код данного примера можно найти в репозитории GitHub для настоящей книги, а в текущем разделе мы его проанализируем. Ниже представлены пример URL для просмотров страниц «Википедии» и код четырех необходимых модулей. Библиотека `requests` отвечает за выполнение HTTP-вызовов, `Pandas` — за преобразование результатов в объект `DataFrame`, а библиотека `wikipedia` будет использоваться для реализации эвристического алгоритма по обнаружению правильных URL спортсменов в «Википедии».

```
"""
```

Пример формируемого маршрута:

```
https://wikimedia.org/api/rest_v1/ +  
metrics/pageviews/per-article/ +  
en.wikipedia/all-access/user/ +  
LeBron_James/daily/2015070100/2017070500 +
```

```
"""
```

```
import requests  
import pandas as pd  
import time  
import wikipedia
```

```
BASE_URL =\
"https://wikimedia.org/api/rest_v1/\
  metrics/pageviews/per-article/en.wikipedia/all-access/user"
```

Следующий фрагмент кода конструирует URL с именем пользователя и диапазоном дат:

```
def construct_url(handle, period, start, end):
    """Конструирует URL на основе аргументов
```

```

    Должен получиться следующий URL:
    /LeBron_James/daily/2015070100/2017070500
    """
```

```

    urls = [BASE_URL, handle, period, start, end]
    constructed = str.join('/', urls)
    return constructed
```

```
def query_wikipedia_pageviews(url):
```

```

    res = requests.get(url)
    return res.json()
```

```
def wikipedia_pageviews(handle, period, start, end):
    """Возвращает JSON"""
```

```

    constructed_url = construct_url(handle, period, start, end)
    pageviews = query_wikipedia_pageviews(url=constructed_url)
    return pageviews
```

Следующая функция автоматически распространяет запрос на весь 2016 г. В дальнейшем можно сделать код более «абстрактным», но пока что мы делаем просто «халтуру», жестко «зашивая» параметры в код там, где ускорение разработки оправдывает некоторую небрежность. Стоит отметить также, что параметр `sleep` пока обнулен, но, возможно, его нужно будет активизировать в случае, если мы наткнемся на ограничения API. Так часто поступают при первом использовании API, ведь они могут вести себя непредсказуемым образом, а приостановка работы на некоторое время часто помогает решить проблему.

```
def wikipedia_2016(handle, sleep=0):
    """Извлекаем данные о просмотрах страниц за 2016 год"""
```

```

    print("SLEEP: {sleep}".format(sleep=sleep))
    time.sleep(sleep)
```

```

pageviews = wikipedia_pageviews(handle=handle,
    period="daily", start="2016010100", end="2016123100")
if not 'items' in pageviews:
    print("NO PAGEVIEWS: {handle}".format(handle=handle))
    return None
return pageviews

```

Далее преобразуем результаты в объект DataFrame библиотеки Pandas:

```

def create_wikipedia_df(handles):
    """Создаем объект Dataframe для просмотров страниц"""

    pageviews = []
    timestamps = []
    names = []
    wikipedia_handles = []
    for name, handle in handles.items():
        pageviews_record = wikipedia_2016(handle)
        if pageviews_record is None:
            continue
        for record in pageviews_record['items']:
            pageviews.append(record['views'])
            timestamps.append(record['timestamp'])
            names.append(name)
            wikipedia_handles.append(handle)
    data = {
        "names": names,
        "wikipedia_handles": wikipedia_handles,
        "pageviews": pageviews,
        "timestamps": timestamps
    }
    df = pd.DataFrame(data)
    return df

```

Здесь начинается более хитрая часть кода, поскольку для того, чтобы угадать, какой дескриптор нам нужен, понадобится определенная эвристика. На первом шаге мы предполагаем, что большинство дескрипторов представляют собой просто `first_last`. На втором шаге мы добавляем к имени `"(basketball)"`, поскольку именно такую стратегию «Википедия» часто использует для разрешения неоднозначностей.

```

def create_wikipedia_handle(raw_handle):
    """Преобразует первоначальный дескриптор в дескриптор
    «Википедии»"""

    wikipedia_handle = raw_handle.replace(" ", "_")

```

```

return wikipedia_handle

def create_wikipedia_nba_handle(name):
    """Присоединяет к ссылке строку (basketball)"""

    url = " ".join([name, "(basketball)"])
    return url

def wikipedia_current_nba_roster():
    """Получает все ссылки со страницы «Википедии» для текущих составов команд"""

    links = {}
    nba = wikipedia.page("List_of_current_NBA_team_rovers")
    for link in nba.links:
        links[link] = create_wikipedia_handle(link)
    return links

```

Этот код как выполняет эвристический алгоритм, так и возвращает проверенные дескрипторы и предположения:

```

def guess_wikipedia_nba_handle(data="data/nba_2017_br.csv"):
    """Пытается получить правильный дескриптор «Википедии»"""

    links = wikipedia_current_nba_roster()
    nba = pd.read_csv(data)
    count = 0
    verified = {}
    guesses = {}
    for player in nba["Player"].values:
        if player in links:
            print("Player: {player}, Link: {link} ".\
                  format(player=player,
                          link=links[player]))
            print(count)
            count += 1
            verified[player] = links[player] #добавить ссылку «Википедии»
        else:
            print("NO MATCH: {player}".format(player=player))
            guesses[player] = create_wikipedia_handle(player)
    return verified, guesses

```

Далее мы воспользуемся библиотекой Python «Википедии» для преобразования неоправдавшихся первоначальных предположений из имени и фамилии и поиска NBA в кратком описании страницы. Это еще один

вполне уместный трюк, позволяющий получить несколько дополнительных соответствий.

```
def validate_wikipedia_guesses(guesses):
    """Проверяем предположения относительно страниц «Википедии»"""

    verified = {}
    wrong = {}
    for name, link in guesses.items():
        try:
            page = wikipedia.page(link)
        except (wikipedia.DisambiguationError,
                wikipedia.PageError) as error:
            #попробуем добавить суффикс basketball
            nba_handle = create_wikipedia_nba_handle(name)
            try:
                page = wikipedia.page(nba_handle)
                print("Initial wikipedia URL Failed:\
                    {error}".format(error=error))
            except (wikipedia.DisambiguationError,
                    wikipedia.PageError) as error:
                print("Second Match Failure: {error}.\
                    \nformat(error=error))
                wrong[name] = link
                continue
        if "NBA" in page.summary:
            verified[name] = link
        else:
            print("NO GUESS MATCH: {name}".format(name=name))
            wrong[name] = link
    return verified, wrong
```

И в конце сценария выполняем весь код и используем результаты для создания нового CSV-файла:

```
def clean_wikipedia_handles(data="data/nba_2017_br.csv"):
    """'Чистые' дескрипторы"""

    verified, guesses = guess_wikipedia_nba_handle(data=data)
    verified_cleaned, wrong = validate_wikipedia_guesses(guesses)
    print("WRONG Matches: {wrong}".format(wrong=wrong))
    handles = {**verified, **verified_cleaned}
    return handles

def nba_wikipedia_dataframe(data="data/nba_2017_br.csv"):
```

```

handles = clean_wikipedia_handles(data=data)
df = create_wikipedia_df(handles)
return df

def create_wikipedia_csv(data="data/nba_2017_br.csv"):
    df = nba_wikipedia_dataframe(data=data)
    df.to_csv("data/wikipedia_nba.csv")

if __name__ == "__main__":
    create_wikipedia_csv()

```

Все вместе это может занять от нескольких часов до нескольких дней, отражая реальность нелегкого труда по обработке произвольных источников данных с целью решения задачи.

Получение данных о вовлеченности в Twitter спортсменов

Получить данные из Twitter несколько проще. Во-первых, в Python существует замечательная библиотека под названием Twitter. Однако некоторые сложности существуют и здесь. Вот их перечень.

1. Получение сводной информации о вовлеченности с помощью описательной статистики.
2. Поиск нужных дескрипторов Twitter (имена дескрипторов в Twitter найти еще сложнее, чем в «Википедии»).
3. Соединение полученного объекта DataFrame с остальными данными.

Во-первых, создадим файл конфигурации `config.py` и поместим в него учетные данные для API Twitter. Далее создадим пространство имен для использования этих учетных данных с помощью команды `. import config`. Импортируем также библиотеку Twitter для обработки ошибок, наряду с Pandas и NumPy.

```

import time

import twitter
from . import config
import pandas as pd
import numpy as np
from twitter.error import TwitterError

```

Следующий код обращается к Twitter, извлекает оттуда 200 твитов и преобразует их в объект DataFrame библиотеки Pandas. Обратите внимание на то, как часто используется этот паттерн при обращении к различным API: столбцы помещаются в список, после чего на его основе создается DataFrame.

```
def api_handler():
    """Подключается к API Twitter"""

    api = twitter.Api(consumer_key=config.CONSUMER_KEY,
                     consumer_secret=config.CONSUMER_SECRET,
                     access_token_key=config.ACCESS_TOKEN_KEY,
                     access_token_secret=config.ACCESS_TOKEN_SECRET)
    return api

def tweets_by_user(api, user, count=200):
    """Извлекает заданное число твитов. По умолчанию 200"""

    tweets = api.GetUserTimeline(screen_name=user, count=count)
    return tweets

def stats_to_df(tweets):
    """Получает статистические данные из Twitter и преобразует их
    в объект Dataframe"""

    records = []
    for tweet in tweets:
        records.append({"created_at":tweet.created_at,
                       "screen_name":tweet.user.screen_name,
                       "retweet_count":tweet.retweet_count,
                       "favorite_count":tweet.favorite_count})
    df = pd.DataFrame(data=records)
    return df

def stats_df(user):
    """Возвращает объект DataFrame со статистикой"""

    api = api_handler()
    tweets = tweets_by_user(api, user)
    df = stats_to_df(tweets)
    return df
```

Теперь можно использовать последнюю функцию, `stats_df`, для интерактивного исследования данных, полученных в результате обращения к API Twitter. Вот пример описательной статистики для Леброна Джеймса (LeBron James):

```
df = stats_df(user="KingJames")
In [34]: df.describe()
Out[34]:
```

	favorite_count	retweet_count
count	200.000000	200.000000
mean	11680.670000	4970.585000
std	20694.982228	9230.301069
min	0.000000	39.000000
25%	1589.500000	419.750000
50%	4659.500000	1157.500000
75%	13217.750000	4881.000000
max	128614.000000	70601.000000

```
In [35]: df.corr()
Out[35]:
```

	favorite_count	retweet_count
favorite_count	1.000000	0.904623
retweet_count	0.904623	1.000000

В следующем коде API Twitter вызывается с небольшой задержкой — во избежание проблем с притормаживанием выполнения из-за ограничений API. Обратите внимание, что дескрипторы Twitter извлекаются из CSV-файла. На сайте Basketball Reference есть большой выбор учетных записей Twitter. Можно также найти их вручную.

```
def twitter_handles(sleep=.5,data="data/twitter_nba_combined.csv"):
    """Выдает дескрипторы"""

    nba = pd.read_csv(data)
    for handle in nba["twitter_handle"]:
        time.sleep(sleep) #Во избежание притормаживания вследствие
                           #ограничений API Twitter
        try:
            df = stats_df(handle)
        except TwitterError as error:
            print("Error {handle} and error msg {error}".format(
                handle=handle,error=error))
            df = None
        yield df

def median_engagement(data="data/twitter_nba_combined.csv"):
    """Медианная вовлеченность в Twitter"""

    favorite_count = []
    retweet_count = []
```

```

nba = pd.read_csv(data)
for record in twitter_handles(data=data):
    print(record)
    #Отсутствующие записи сохраняются в виде значения Nan
    if record is None:
        print("NO RECORD: {record}".format(record=record))
        favorite_count.append(np.nan)
        retweet_count.append(np.nan)
        continue
    try:
        favorite_count.append(record['favorite_count'].median())
        retweet_count.append(record["retweet_count"].median())
    except KeyError as error:
        print("No values found to append {error}".\
            format(error=error))
        favorite_count.append(np.nan)
        retweet_count.append(np.nan)

print("Creating DF")
nba['twitter_favorite_count'] = favorite_count
nba['twitter_retweet_count'] = retweet_count
return nba

```

И наконец, создается новый CSV-файл:

```

def create_twitter_csv(data="data/nba_2016_2017_wikipedia.csv"):
    nba = median_engagement(data)
    nba.to_csv("data/nba_2016_2017_wikipedia_twitter.csv")

```

Изучаем данные об игроках НБА

Для исследования данных спортсменов мы создадим новый блокнот Jupiter с названием `nba_player_power_influence_performance`. Для начала импортируем несколько часто используемых библиотек:

```

In [106]: import pandas as pd
...: import numpy as np
...: import statsmodels.api as sm
...: import statsmodels.formula.api as smf
...: import matplotlib.pyplot as plt
...: import seaborn as sns
...: from sklearn.cluster import KMeans
...: color = sns.color_palette()
...: from IPython.core.display import display, HTML
...: display(HTML("<style>.container\

```

```
{ width:100% !important; }</style>"))
...: %matplotlib inline
...:
<IPython.core.display.HTML object>
```

Далее загружаем в проект файлы данных и переименовываем столбцы:

```
In [108]: attendance_valuation_elo_df = \
pd.read_csv("../data/nba_2017_att_val_elo.csv")
In [109]: salary_df = pd.read_csv("../data/nba_2017_salary.csv")
In [110]: pie_df = pd.read_csv("../data/nba_2017_pie.csv")
In [111]: plus_minus_df = \
pd.read_csv("../data/nba_2017_real_plus_minus.csv")
In [112]: br_stats_df = pd.read_csv("../data/nba_2017_br.csv")
In [113]: plus_minus_df.rename(
    columns={"NAME": "PLAYER", "WINS": "WINS_RPM"}, inplace=True)
...: players = []
...: for player in plus_minus_df["PLAYER"]:
...:     plyr, _ = player.split(",")
...:     players.append(plyr)
...: plus_minus_df.drop(["PLAYER"], inplace=True, axis=1)
...: plus_minus_df["PLAYER"] = players
...:
```

Некоторые из источников данных дублируются, их можно отбросить:

```
In [114]: nba_players_df = br_stats_df.copy()
...: nba_players_df.rename(
    columns={'Player': 'PLAYER', 'Pos': 'POSITION',
            'Tm': "TEAM", 'Age': 'AGE', "PS/G": "POINTS"}, i
...: nplace=True)
...: nba_players_df.drop(["G", "GS", "TEAM"],
    inplace=True, axis=1)
...: nba_players_df = \
nba_players_df.merge(plus_minus_df, how="inner", on="PLAYER")
...:

In [115]: pie_df_subset = pie_df[["PLAYER", "PIE",
    "PACE", "W"]].copy()
...: nba_players_df = nba_players_df.merge(
    pie_df_subset, how="inner", on="PLAYER")
...:

In [116]: salary_df.rename(columns={'NAME': 'PLAYER'}, inplace=True)
...: salary_df["SALARY_MILLIONS"] = \
    round(salary_df["SALARY"]/1000000, 2)
...: salary_df.drop(["POSITION", "TEAM", "SALARY"],
```

```
inplace=True, axis=1)
...:
```

```
In [117]: salary_df.head()
```

```
Out[117]:
```

	PLAYER	SALARY_MILLIONS
0	LeBron James	30.96
1	Mike Conley	26.54
2	Al Horford	26.54
3	Dirk Nowitzki	25.00
4	Carmelo Anthony	24.56

Для 111 игроков НБА отсутствует информация о зарплате, так что часть игроков при анализе мы также отбросим:

```
In [118]: diff = list(set(
    nba_players_df["PLAYER"].values.tolist() -
    set(salary_df["PLAYER"].values.tolist()))
```

```
In [119]: len(diff)
```

```
Out[119]: 111
```

```
In [120]: nba_players_with_salary_df = \
    nba_players_df.merge(salary_df);
```

У нас остался объект DataFrame Pandas с 38 столбцами:

```
In [121]: nba_players_with_salary_df.columns
```

```
Out[121]:
```

```
Index(['Rk', 'PLAYER', 'POSITION', 'AGE', 'MP',
      'FG', 'FGA', 'FG%', '3P',
      '3PA', '3P%', '2P', '2PA', '2P%', 'eFG%',
      'FT', 'FTA', 'FT%', 'ORB',
      'DRB', 'TRB', 'AST', 'STL', 'BLK', 'TOV',
      'PF', 'POINTS', 'TEAM', 'GP',
      'MPG', 'ORPM', 'DRPM', 'RPM', 'WINS_RPM',
      'PIE', 'PACE', 'W',
      'SALARY_MILLIONS'],
      dtype='object')
```

```
In [122]: len(nba_players_with_salary_df.columns)
```

```
Out[122]: 38
```

Далее этот объект DataFrame можно объединить с данными из «Википедии». Они свернуты в одно поле с медианой, чтобы их можно было представить в виде одной строки в столбце:

```
In [123]: wiki_df = pd.read_csv(
    "../data/nba_2017_player_wikipedia.csv")
In [124]: wiki_df.rename(columns=\
    {'names': 'PLAYER', "pageviews": "PAGEVIEWS"}, inplace=True)
In [125]: median_wiki_df = wiki_df.groupby("PLAYER").median()
In [126]: median_wiki_df_small = median_wiki_df[["PAGEVIEWS"]]
In [127]: median_wiki_df_small.reset_index(
    level=0, inplace=True);median_wiki_df_sm.head()
```

Out[127]:

	PLAYER	PAGEVIEWS
0	A.J. Hammons	1.0
1	Aaron Brooks	10.0
2	Aaron Gordon	666.0
3	Aaron Harrison	487.0
4	Adreian Payne	166.0

```
In [128]: nba_players_with_salary_wiki_df =\
    nba_players_with_salary_df.merge(median_wiki_df_small)
```

Осталось только добавить столбцы с данными из Twitter:

```
In [129]: twitter_df = pd.read_csv(
    "../data/nba_2017_twitter_players.csv")
In [130]: nba_players_with_salary_wiki_twitter_df =\
    nba_players_with_salary_wiki_df.merge(twitter_df)
```

Всего у нас теперь 41 атрибут:

```
In [132]: len(nba_players_with_salary_wiki_twitter_df.columns)
Out[132]: 41
```

Следующий логичный этап исследования данных — создание карты интенсивности корреляции:

```
In [133]: plt.subplots(figsize=(20,15))
    ...: ax = plt.axes()
    ...: ax.set_title("NBA Player Correlation Heatmap")
    ...: corr = nba_players_with_salary_wiki_twitter_df.corr()
    ...: sns.heatmap(corr,
    ...:               xticklabels=corr.columns.values,
    ...:               yticklabels=corr.columns.values)
    ...:
```

```
Out[133]: <matplotlib.axes._subplots.AxesSubplot at 0x111e665c0>
<matplotlib.figure.Figure at 0x111e66780>
```


Машинное обучение без учителя для данных об игроках НБА

При наличии такого разнообразного набора данных и множества полезных атрибутов немало ценной информации может принести выполнение машинного обучения без учителя по игрокам НБА. Первый этап — нормирование данных и выбор атрибутов для кластеризации (отбрасывание строк с отсутствующими значениями):

```
In [135]: numerical_df = \
nba_players_with_salary_wiki_twitter_df.loc[:, \
["AGE", "TRB", "AST", "STL", "TOV", "BLK", "PF", "POINTS", \
"MPG", "WINS_RPM", "W", "SALARY_MILLIONS", "PAGEVIEWS", \
"TWITTER_FAVORITE_COUNT"]].dropna()
In [142]: from sklearn.preprocessing import MinMaxScaler
...: scaler = MinMaxScaler()
...: print(scaler.fit(numerical_df))
...: print(scaler.transform(numerical_df))
...:
MinMaxScaler(copy=True, feature_range=(0, 1))
[[ 4.28571429e-01  8.35937500e-01  9.27927928e-01 ...,
  2.43447079e-01  1.73521746e-01]
 [ 3.80952381e-01  6.32812500e-01  1.00000000e+00 ...,
  1.86527023e-01  7.89216485e-02]
 [ 1.90476190e-01  9.21875000e-01  1.80180180e-01 ...,
  4.58206449e-03  2.99723082e-02]
 ...,
 [ 9.52380952e-02  8.59375000e-02  2.70270270e-02 ...,
  1.52830350e-02  8.95911386e-04]
 [ 2.85714286e-01  8.59375000e-02  3.60360360e-02 ...,
  1.19532117e-03  1.38459032e-03]
 [ 1.42857143e-01  1.09375000e-01  1.80180180e-02 ...,
  7.25730711e-03  0.00000000e+00]]
```

Далее выполняем кластеризацию еще раз и выводим результаты в CSV-файл для построения фасетного графика на языке R:

```
In [149]: from sklearn.cluster import KMeans
...: k_means = KMeans(n_clusters=5)
...: kmeans = k_means.fit(scaler.transform(numerical_df))
...: nba_players_with_salary_wiki_twitter_df['cluster'] =
kmeans.labels_
...:
In [150]: nba_players_with_salary_wiki_twitter_df.to_csv(
"../data/nba_2017_players_social_with_clusters.csv")
```

Построение фасетного графика по игрокам НБА на языке R

Сначала импортируем CSV-файл и воспользуемся библиотекой `ggplot2`:

```
> player_cluster <- read_csv(
+ "nba_2017_players_social_with_clusters.csv",
+   col_types = cols(X1 = col_skip()))
> library("ggplot2")
```

Далее дадим всем четырем кластерам осмысленные имена:

```
> #Задаем имена кластеров
> player_cluster$cluster_name[player_cluster$
+ cluster == 0] <- "Low Pay/Low Performance"
> player_cluster$cluster_name[player_cluster$
+ cluster == 1] <- "High Pay/Above Average Performance"
> player_cluster$cluster_name[player_cluster$
+ cluster == 2] <- "Low Pay/Average Performance"
> player_cluster$cluster_name[player_cluster$
+ cluster == 3] <- "High Pay/High Performance"
> player_cluster$cluster_name[player_cluster$
+ cluster == 4] <- "Medium Pay/Above Average Performance"
```

Создадим фасетный график с именами кластеров:

```
> #Создаем фасетный график
> p <- ggplot(data = player_cluster) +
+   geom_point(mapping = aes(x = WINS_RPM,
+                             y = POINTS,
+                             color = SALARY_MILLIONS,
+                             size = PAGEVIEWS))+
+   facet_wrap(~ cluster_name) +
+   ggtitle("NBA Players Faceted") +
+   ylab("POINTS PER GAME") +
+   xlab("WINS ATTRIBUTABLE TO PLAYER (WINS_RPM)") +
+   geom_text(aes(x = WINS_RPM, y = POINTS,
```

Чтобы выбрать текст графика в каждом фасете, придется потрудиться, но всю эту работу выполняют R и/или нижеприведенные операторы. Для зарплаты также используются три цвета — для более наглядного отображения различий:

```
label=ifelse(
+ PAGEVIEWS>10000|TOV>5|AGE>37|WINS_RPM>15|cluster
+ == 2 & WINS_RPM > 3,
```

```

+
as.character(PLAYER, '')),hjust=.8, check_overlap = FALSE)
>
> #Меняем легенды
> p +
+   guides(color = guide_legend(title = "Salary Millions")) +
+   guides(size = guide_legend(
+     title = "Wikipedia Daily Pageviews" ))+
+   scale_color_gradientn(colours = rainbow(3))
> geom_text(aes(x = ELO, y = VALUE_MILLIONS, label=ifelse(
VALUE_MILLIONS>1200, as.character(TEAM), '')),hjust=.35,vjust=1)

```

В итоге получаем аккуратный фасетный график (рис. 6.12). Основные обнаруженные метки представляют собой различия между известностью, зарплатой и игровыми результатами. «Лучшие из лучших» — в кластере с Леброном Джеймсом и Расселом Вестбруком, но они различаются также и самыми высокими зарплатами.

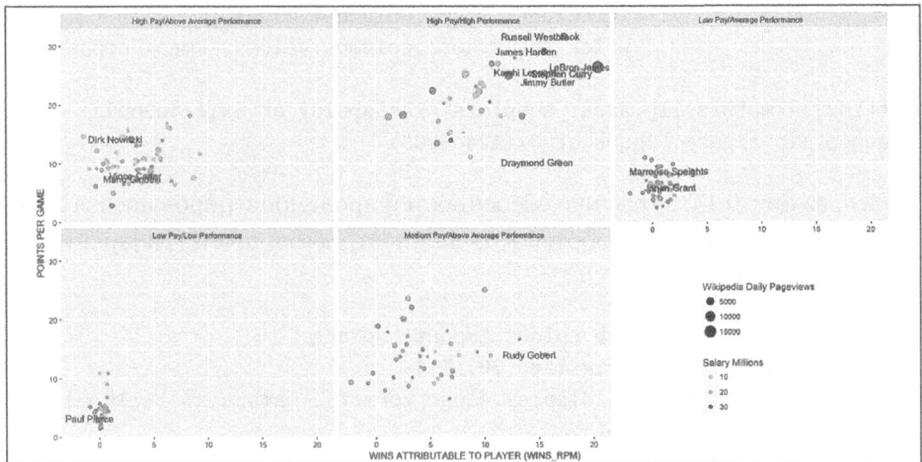


Рис. 6.12. Фасетный график, построенный с помощью библиотеки ggplot, для игроков НБА за 2016–2017 гг. на основе кластеризации методом k-ближайших соседей

Собираем все воедино: команды, игроков, социальный авторитет и рекламные отчисления

После сбора всех данных мы можем попробовать несколько новых интересных графиков. Объединив данные о рекламных отчислениях, командах и игроках, можно создать несколько потрясающих графиков. Во-первых,

показать данные о рекламных отчислениях на карте интенсивности корреляции (рис. 6.13):

```
In [150]: nba_players_with_salary_wiki_twitter_df.to_csv(
"../data/nba_2017_players_social_with_clusters.csv")
```

```
In [151]: endorsements = pd.read_csv(
"../data/nba_2017_endorsement_full_stats.csv")
```

```
In [152]: plt.subplots(figsize=(20,15))
...: ax = plt.axes()
...: ax.set_title("NBA Player Endorsement, \
Social Power, On-Court Performance, \
Team Valuation Correlation Heatmap: 2016-2017
...: Season")
...: corr = endorsements.corr()
...: sns.heatmap(corr,
...:             xticklabels=corr.columns.values,
...:             yticklabels=corr.columns.values, cmap="copper")
...:
```

```
Out[152]: <matplotlib.axes._subplots.AxesSubplot at 0x1124d90b8>
<matplotlib.figure.Figure at 0x1124d9908>
```

Далее, на рис. 6.14, в графике особенностей продемонстрирована вся совокупность проделанной работы. Код для построения этого графика имеет следующий вид:

```
In [153]: from matplotlib.colors import LogNorm
...: plt.subplots(figsize=(20,15))
...: pd.set_option('display.float_format', lambda x: '%.3f' % x)
...: norm = LogNorm()
...: ax = plt.axes()
...: grid = endorsements.select_dtypes([np.number])
...: ax.set_title("NBA Player Endorsement, \
Social Power, On-Court Performance, \
Team Valuation Heatmap: 2016-2017 Season")
...: sns.heatmap(grid,annot=True,
...:             yticklabels=endorsements["PLAYER"],fmt='g',
...:             cmap="Accent", cbar=False, norm=norm)
...:
```

```
Out[153]: <matplotlib.axes._subplots.AxesSubplot at 0x114902cc0>
<matplotlib.figure.Figure at 0x114902048>
```

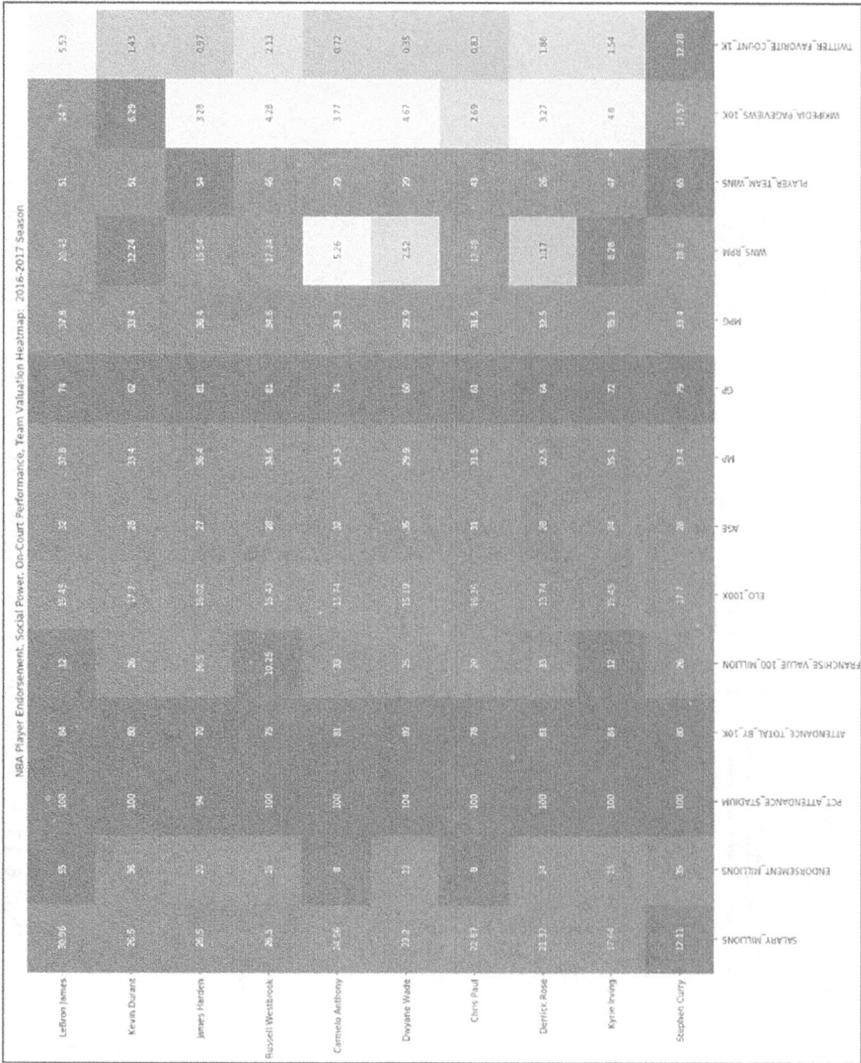



Рис. 6.14. Рекламные отчисления, график особенностей

Обратите внимание, что важную роль в обеспечении удобочитаемости графика особенностей играет преобразование цветов в LogNorm. Благодаря этим относительные изменения приводят к созданию границ между ячейками.

Дальнейшие прагматичные шаги и учебные материалы

Одна из основных причин написания этой книги — желание продемонстрировать создание завершенных работоспособных решений, пригодных для развертывания в промышленной эксплуатации. Чтобы сделать из блокнота подобное решение, нужно изучить некоторые из приведенных в других главах решений, где обсуждаются методы подготовки проектов для промышленной эксплуатации, например создания API прогноза стоимости команд НБА или API для отображения социального авторитета суперзвезд НБА. До презентации на получение инвестиций в фонде Y Combinator (YC) вас может отделять всего несколько строк кода.

Кроме того, существует возможность ветвления блокнота Kaggle (<https://www.kaggle.com/noahgift/social-power-nba>) в качестве исходной точки для дальнейших исследований. Наконец, вы можете найти видео и слайды по данному вопросу на сайте конференции по данным Strata-2018 в Сан-Хосе: <https://conferences.oreilly.com/strata/strata-ca/public/schedule/detail/63606>.

Резюме

В этой главе мы рассмотрели реальную практическую задачу машинного обучения, начав с постановки вопросов и перейдя затем к методикам сбора данных из всего Интернета. Многие из меньших наборов данных были скопированы с сайтов, не препятствующих сбору данных. Для больших источников данных — «Википедии» и Twitter — потребовался иной подход, скорее программный.

Далее мы изучили данные методами как статистики, так и машинного обучения без учителя и визуализации данных.

7 Создание интеллектуального бота Slack в AWS

Победителем становится тот, кто способен двигаться вперед, несмотря на боль.

*Роджер Баннистер
(Roger Banister)*

Люди давно мечтают создать «искусственную жизнь». Чаще всего пока это возможно путем создания ботов. Боты становятся все более неотъемлемой частью нашей повседневной жизни, особенно после появления Siri от компании Apple и Alexa от Amazon. В этой главе мы раскроем все тайны создания ботов.

Создание бота

Для создания бота мы воспользуемся библиотекой Slack для языка Python (<https://github.com/slackapi/python-slackclient>). Для начала работы со Slack необходимо сгенерировать идентификационный маркер. В целом имеет смысл при работе с подобными маркерами экспортировать переменную среды. Я часто делаю это в `virtualenv`, получая, таким образом, автоматически доступ к ней при выполнении в текущей среде. Для этого необходимо немного «взломать» утилиту `virtualenv`, отредактировав сценарий `activate`.

При экспорте переменной Slack в сценарии `~/.env/bin/activate` он будет иметь нижеприведенный вид.

И просто для информации, если вы хотите идти в ногу с последними новинками, рекомендуется использовать появившуюся на рынке новую, офици-

альную утилиту Python для управления средой — `pipenv` (<https://github.com/pypa/pipenv>):

```
_OLD_VIRTUAL_PATH="$PATH"
PATH="$VIRTUAL_ENV/bin:$PATH"
export PATH
SLACK_API_TOKEN=<Your Token Here>
export SLACK_API_TOKEN
```

Для проверки того, задано ли значение переменной среды, удобно использовать команду `printenv` операционных систем OS X и Linux. После этого для проверки отправки сообщения можно воспользоваться следующим коротким сценарием:

```
import os
from slackclient import SlackClient

slack_token = os.environ["SLACK_API_TOKEN"]
sc = SlackClient(slack_token)

sc.api_call(
    "chat.postMessage",
    channel="#general",
    text="Hello from my bot! :tada:"
)
```

Стоит также отметить, что утилита `pipenv` — рекомендуемое решение, объединяющее в одном компоненте возможности утилит `pip` и `virtualenv`. Она стала новым стандартом, так что имеет смысл взглянуть на нее с точки зрения управления пакетами.

Преобразование библиотеки в утилиту командной строки

Как и в других примерах из этой книги, удачной идеей будет преобразовать наш код в утилиту командной строки, чтобы облегчить проверку новых идей. Стоит отметить, что многие разработчики-новички часто отдают предпочтение не утилитам командной строки, а другим решениям, например, просто работают в блокнотах Jupiter. Сыграю ненадолго роль адвоката дьявола и задам вопрос, который вполне может возникнуть у читателей: «А зачем нам утилиты командной строки в проекте, основанном на

блокнотах Jupiter? Разве смысл блокнотов Jupiter состоит не в том, чтобы сделать ненужными командную оболочку и командную строку?» Добавление утилиты командной строки в проект хорошо тем, что позволяет быстро пробовать различные варианты входных данных. Блоки кода блокнотов Jupiter не принимают входные данные, в некотором смысле это сценарии с жестко «защитыми» данными.

Множество утилит командной строки на платформах как GCP, так и AWS существует не случайно: они обеспечивают гибкость и возможности, недоступные для графических интерфейсов. Замечательный сборник эссе на эту тему фантаста Нила Стивенсона (Neal Stephenson) называется «В начале... была командная строка». В нем Стивенсон говорит: «GUI приводят к значительным дополнительным накладным расходам на каждый, даже самый маленький компонент программного обеспечения, которые полностью меняют среду программирования». Он заканчивает сборник словами: «...жизнь — штука очень тяжелая и сложная; никакой интерфейс это не изменит; и всякий, кто считает иначе, — простофиля...» Достаточно жестко, но мой опыт подсказывает, что и достаточно правдиво. Жизнь с командной строкой становится лучше. Попробуйте ее — и вы не захотите возвращаться обратно к GUI.

Для этого мы воспользуемся пакетом `click`, как показано ниже. Отправка сообщений с помощью нового интерфейса оказывается очень простым делом.

```
./clibot.py send --message "from cli"  
sending message from cli to #general
```

Рисунок 7.1 демонстрирует значения по умолчанию, а также настраиваемое сообщение от утилиты `cli`.

```
#!/usr/bin/env python  
import os  
import click  
from slackclient import SlackClient  
  
SLACK_TOKEN = os.environ["SLACK_API_TOKEN"]  
  
def send_message(channel="#general",  
                 message="Hello from my bot!"):  
    """Отправить сообщение на канал"""  
  
    slack_client = SlackClient(SLACK_TOKEN)  
    res = slack_client.api_call(  

```

```

    "chat.postMessage",
    channel=channel,
    text=message
)
return res

@click.group()
@click.version_option("0.1")
def cli():
    """
    Утилита командной строки для слабаков
    """

    @cli.command("send")
    @click.option("--message", default="Hello from my bot!",
                  help="text of message")
    @click.option("--channel", default="#general",
                  help="general channel")
    def send(channel, message):
        click.echo(f"sending message {message} to {channel}")
        send_message(channel, message=message)

if __name__ == '__main__':
    cli()

```

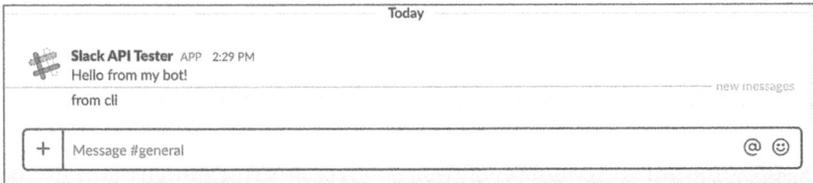


Рис. 7.1. Утилита командной строки бота Slack

Выводим бот на новый уровень с помощью сервиса AWS Step Functions

После создания каналов связи для отправки сообщений в Slack можно усовершенствовать наш код, а именно: запускать его по расписанию и использовать для каких-либо полезных действий. Сервис пошаговых функций AWS (AWS Step Functions) замечательно подходит для этой цели. В следующем разделе наш бот Slack научится производить скрапинг

спортивных страниц Yahoo! игроков НБА, извлекать их места рождения, а затем отправлять эти данные в Slack.

Рисунок 7.2 демонстрирует готовую пошаговую функцию в действии. Первый шаг состоит в извлечении URL профилей игроков НБА, а второй — в использовании библиотеки Beautiful Soup для поиска места рождения каждого из игроков. По завершении выполнения пошаговой функции результаты будут отправлены обратно в Slack.

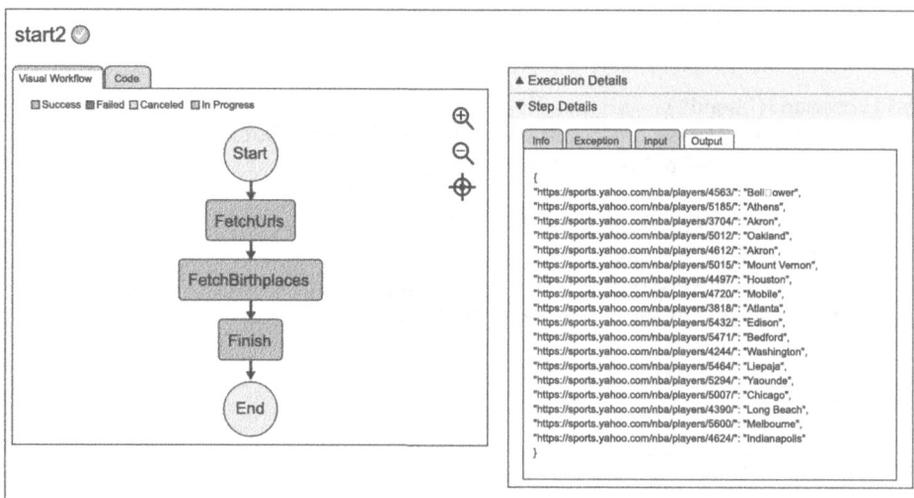


Рис. 7.2. Начальный конвейер утилиты командной строки бота Slack

Для координации отдельных частей работы внутри пошаговой функции можно применить AWS Lambda и Chalice. Lambda (<https://aws.amazon.com/lambda/>) позволяет пользователю выполнять функции в AWS, а фреймворк Chalice (<http://chalice.readthedocs.io/en/latest/>) дает возможность создания бессерверных приложений на языке Python. Вот некоторые предварительные требования:

- ❑ у пользователя должна быть учетная запись AWS;
- ❑ пользователю необходимы учетные данные для использования API;
- ❑ у роли Lambda (создаваемой Chalice) должна быть политика с привилегиями, необходимыми для вызова соответствующих сервисов AWS, например S3.

Настройка учетных данных IAM

Подробные инструкции по настройке учетных данных AWS можно найти по адресу <http://boto3.readthedocs.io/en/latest/guide/configuration.html>. Информацию об экспорте переменных AWS в операционных системах Windows и Linux можно найти здесь: <https://docs.aws.amazon.com/amazonswf/latest/awsrblflowguide/set-up-creds.html>. Существует множество способов настройки учетных данных, но пользователи `virtualenv` могут поместить учетные данные AWS в локальную виртуальную среду, в сценарий `/bin/activate`:

```
#Добавляем ключи AWS
AWS_DEFAULT_REGION=us-east-1
AWS_ACCESS_KEY_ID=xxxxxxxxx
AWS_SESSION_TOKEN=xxxxxxxxx
```

```
#Экспортируем ключи
export AWS_DEFAULT_REGION
export AWS_ACCESS_KEY_ID
export AWS_SESSION_TOKEN
```

Работа с Chalice. У Chalice есть утилита командной строки с множеством доступных команд:

```
Usage: chalice [OPTIONS] COMMAND [ARGS]...
```

Options:

```
--version           Show the version and exit.
--project-dir TEXT  The project directory. Defaults to CWD
--debug / --no-debug Print debug logs to stderr.
--help              Show this message and exit.
```

Commands:

```
delete
deploy
gen-policy
generate-pipeline  Generate a cloudformation template for a...
generate-sdk
local
logs
new-project
package
url
```

Код внутри шаблона `app.py` можно заменить на функции сервиса `Lambda`. В `AWS Chalice` удобно то, что он дает возможность создавать, помимо веб-сервисов, «автономные» функции `Lambda`. Благодаря этой функциональности можно создать несколько функций `Lambda`, связать их с пошаговой функцией и свести воедино, как кубики «Лего».

Например, можно легко создать запускаемую по расписанию функцию `Lambda`, которая будет выполнять какие-либо действия:

```
@app.schedule(Rate(1, unit=Rate.MINUTES))
def every_minute(event):
    """Событие, запланированное для ежеминутного выполнения"""

    #Отправка сообщения боту Slack
```

Для налаживания взаимодействия с ботом для веб-скрапинга необходимо создать несколько функций. В начале файла находятся импорты и объявлено некоторое количество переменных:

```
import logging
import csv
from io import StringIO

import boto3
from bs4 import BeautifulSoup
import requests
from chalice import (Chalice, Rate)
```

```
APP_NAME = 'scrape-yahoo'
app = Chalice(app_name=APP_NAME)
app.log.setLevel(logging.DEBUG)
```

Боту может понадобиться хранить часть данных в `S3`. Следующая функция использует `Boto` для сохранения результатов в `CSV`-файле:

```
def create_s3_file(data, name="birthplaces.csv"):

    csv_buffer = StringIO()
    app.log.info(f"Creating file with {data} for name")
    writer = csv.writer(csv_buffer)
    for key, value in data.items():
        writer.writerow([key,value])
    s3 = boto3.resource('s3')
```

```

res = s3.Bucket('aiwebscraping').\
    put_object(Key=name, Body=csv_buffer.getvalue())
return res

```

Функция `fetch_page` использует библиотеку Beautiful Soup (<https://www.crummy.com/software/BeautifulSoup>) для синтаксического разбора HTML-страницы, расположенной в соответствии с URL статистики НБА, и возвращает объект `soup`:

```

def fetch_page(url="https://sports.yahoo.com/nba/stats/"):
    """Извлекает URL Yahoo"""

    #Скачивает страницу и преобразует ее в объект
    # библиотеки Beautiful Soup
    app.log.info(f"Fetching urls from {url}")
    res = requests.get(url)
    soup = BeautifulSoup(res.content, 'html.parser')
    return soup

```

Функции `get_player_links` и `fetch_player_urls` получают ссылки на URL профилей игроков:

```

def get_player_links(soup):
    """Получает ссылки из URL игроков

    Находит все URL на странице в тегах 'a' и фильтрует их в поисках
    строки 'nba/players'
    """

    nba_player_urls = []
    for link in soup.find_all('a'):
        link_url = link.get('href')
        #Отбрасываем неподходящие
        if link_url:
            if "nba/players" in link_url:
                print(link_url)
                nba_player_urls.append(link_url)
    return nba_player_urls

def fetch_player_urls():
    """Возвращает URL игроков"""

    soup = fetch_page()

```

```
urls = get_player_links(soup)
return urls
```

Далее в функции `find_birthplaces` мы извлекаем с расположенных по этим URL страниц места рождения игроков:

```
def find_birthplaces(urls):
    """Получаем места рождения со страниц профилей игроков NBA
    на Yahoo"""

    birthplaces = {}
    for url in urls:
        profile = requests.get(url)
        profile_url = BeautifulSoup(profile.content, 'html.parser')
        lines = profile_url.text
        res2 = lines.split(",")
        key_line = []
        for line in res2:
            if "Birth" in line:
                #print(line)
                key_line.append(line)
        try:
            birth_place = key_line[0].split(":")[-1].strip()
            app.log.info(f"birth_place: {birth_place}")
        except IndexError:
            app.log.info(f"skipping {url}")
            continue
        birthplaces[url] = birth_place
        app.log.info(birth_place)
    return birthplaces
```

Теперь мы перейдем к функциям Chalice. Обратите внимание: для фреймворка Chalice необходимо, чтобы был создан путь по умолчанию:

```
#Их можно вызвать с помощью HTTP-запросов
@app.route('/')
def index():
    """Корневой URL"""

    app.log.info(f"/ Route: for {APP_NAME}")
    return {'app_name': APP_NAME}
```

Следующая функция `Lambda` представляет собой маршрут, связывающий HTTP URL с написанной ранее функцией:

```
@app.route('/player_urls')
def player_urls():
    """Извлекает URL игроков"""

    app.log.info(f"/player_urls Route: for {APP_NAME}")
    urls = fetch_player_urls()
    return {"nba_player_urls": urls}
```

Следующие функции Lambda — автономные, их можно вызвать внутри пошаговой функции:

```
#Это автономная функция Lambda
@app.lambda_function()
def return_player_urls(event, context):
    """Автономная функция Lambda, возвращающая URL игроков"""

    app.log.info(f"standalone lambda 'return_players_urls'\
        {APP_NAME} with {event} and {context}")
    urls = fetch_player_urls()
    return {"urls": urls}
```

```
#Это автономная функция Lambda
@app.lambda_function()
def birthplace_from_urls(event, context):
    """Находит места рождения игроков"""

    app.log.info(f"standalone lambda 'birthplace_from_urls'\
        {APP_NAME} with {event} and {context}")
    payload = event["urls"]
    birthplaces = find_birthplaces(payload)
    return birthplaces
```

```
#Это автономная функция Lambda
@app.lambda_function()
def create_s3_file_from_json(event, context):
    """Создает файл S3 на основе данных в формате JSON"""

    app.log.info(f"Creating s3 file with event data {event}\
        and context {context}")
    print(type(event))
    res = create_s3_file(data=event)
    app.log.info(f"response of putting file: {res}")
    return True
```

Если запустить получившееся приложение Chalice локально, будут выведены следующие результаты:

```
→ scrape-yahoo git:(master) X chalice local
Serving on 127.0.0.1:8000
scrape-yahoo - INFO - / Route: for scrape-yahoo
127.0.0.1 - - [12/Dec/2017 03:25:42] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2017 03:25:42] "GET /favicon.ico"
scrape-yahoo - INFO - / Route: for scrape-yahoo
127.0.0.1 - - [12/Dec/2017 03:25:45] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2017 03:25:45] "GET /favicon.ico"
scrape-yahoo - INFO - /player_urls Route: for scrape-yahoo
scrape-yahoo - INFO - https://sports.yahoo.com/nba/stats/
https://sports.yahoo.com/nba/players/4563/
https://sports.yahoo.com/nba/players/5185/
https://sports.yahoo.com/nba/players/3704/
https://sports.yahoo.com/nba/players/5012/
https://sports.yahoo.com/nba/players/4612/
https://sports.yahoo.com/nba/players/5015/
https://sports.yahoo.com/nba/players/4497/
https://sports.yahoo.com/nba/players/4720/
https://sports.yahoo.com/nba/players/3818/
https://sports.yahoo.com/nba/players/5432/
https://sports.yahoo.com/nba/players/5471/
https://sports.yahoo.com/nba/players/4244/
https://sports.yahoo.com/nba/players/5464/
https://sports.yahoo.com/nba/players/5294/
https://sports.yahoo.com/nba/players/5336/
https://sports.yahoo.com/nba/players/4390/
https://sports.yahoo.com/nba/players/4563/
https://sports.yahoo.com/nba/players/3704/
https://sports.yahoo.com/nba/players/5600/
https://sports.yahoo.com/nba/players/4624/
127.0.0.1 - - [12/Dec/2017 03:25:53] "GET /player_urls"
127.0.0.1 - - [12/Dec/2017 03:25:53] "GET /favicon.ico"
```

Для развертывания приложения выполните команду chalice deploy:

```
→ scrape-yahoo git:(master) X chalice deploy
Creating role: scrape-yahoo-dev
Creating deployment package.
Creating lambda function: scrape-yahoo-dev
Initiating first time deployment.
Deploying to API Gateway stage: api
https://bt98uzs1cc.execute-api.us-east-1.amazonaws.com/api/
```

Благодаря интерфейсу командной строки для HTTP (<https://github.com/jakubroztocil/httpie>) мы вызываем маршрут HTTP из AWS и извлекаем доступные в `/api/player_urls` ссылки:

```
→ scrape-yahoo git:(master) X http \
https://<a lambda route>.amazonaws.com/api/player_urls
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 941
Content-Type: application/json
Date: Tue, 12 Dec 2017 11:48:41 GMT
Via: 1.1 ba90f9bd20de9ac04075a8309c165ab1.cloudfront.net (CloudFront)
X-Amz-Cf-Id: ViZswjo4UeHYwrc9e-5vMVTdHv_Ic0dhVIG0BrDdtYqD5KWcAuZKKQ==
X-Amzn-Trace-Id: sampled=0;root=1-5a2fc217-07cc12d50a4d38a59a688f5c
X-Cache: Miss from cloudfront
x-amzn-RequestId: 64f24fcd-df32-11e7-a81a-2b511652b4f6
```

```
{
  "nba_player_urls": [
    "https://sports.yahoo.com/nba/players/4563/",
    "https://sports.yahoo.com/nba/players/5185/",
    "https://sports.yahoo.com/nba/players/3704/",
    "https://sports.yahoo.com/nba/players/5012/",
    "https://sports.yahoo.com/nba/players/4612/",
    "https://sports.yahoo.com/nba/players/5015/",
    "https://sports.yahoo.com/nba/players/4497/",
    "https://sports.yahoo.com/nba/players/4720/",
    "https://sports.yahoo.com/nba/players/3818/",
    "https://sports.yahoo.com/nba/players/5432/",
    "https://sports.yahoo.com/nba/players/5471/",
    "https://sports.yahoo.com/nba/players/4244/",
    "https://sports.yahoo.com/nba/players/5464/",
    "https://sports.yahoo.com/nba/players/5294/",
    "https://sports.yahoo.com/nba/players/5336/",
    "https://sports.yahoo.com/nba/players/4390/",
    "https://sports.yahoo.com/nba/players/4563/",
    "https://sports.yahoo.com/nba/players/3704/",
    "https://sports.yahoo.com/nba/players/5600/",
    "https://sports.yahoo.com/nba/players/4624/"
  ]
}
```

Еще один удобный способ работы с функциями Lambda — непосредственный их вызов с помощью пакета `click` и библиотеки `Voto` языка Python.

Мы можем создать новую утилиту командной строки с названием `wsccli.py` (сокращение от *web-scraping command-line interface* — «интерфейс командной строки для веб-скрапинга»). В первой части кода мы настраиваем журналирование и импортируем библиотеки:

```
#!/usr/bin/env python

import logging
import json

import boto3
import click
from pythonjsonlogger import jsonlogger

#Инициализация журналирования
log = logging.getLogger(__name__)
log.setLevel(logging.INFO)
LOGHANDLER = logging.StreamHandler()
FORMMATTER = jsonlogger.JsonFormatter()
LOGHANDLER.setFormatter(FORMMATTER)
log.addHandler(LOGHANDLER)
```

Следующие три функции предназначены для подключения к функции Lambda через `invoke_lambda`:

```
###Вызовы API Boto Lambda
def lambda_connection(region_name="us-east-1"):
    """Создаем подключение к Lambda"""

    lambda_conn = boto3.client("lambda", region_name=region_name)
    extra_msg = {"region_name": region_name, "aws_service": "lambda"}
    log.info("instantiate lambda client", extra=extra_msg)
    return lambda_conn

def parse_lambda_result(response):
    """Получаем результаты из ответа библиотеки Boto в формате JSON"""

    body = response['Payload']
    json_result = body.read()
    lambda_return_value = json.loads(json_result)
    return lambda_return_value

def invoke_lambda(func_name, lambda_conn, payload=None,
                  invocation_type="RequestResponse"):
```

```

"""Вызываем функцию Lambda"""

extra_msg = {"function_name": func_name, "aws_service": "lambda",
             "payload":payload}
log.info("Calling lambda function", extra=extra_msg)
if not payload:
    payload = json.dumps({"payload":"None"})

response = lambda_conn.invoke(FunctionName=func_name,
                              InvocationType=invocation_type,
                              Payload=payload
)
log.info(response, extra=extra_msg)
lambda_return_value = parse_lambda_result(response)
return lambda_return_value

```

Обертываем функцию `invoke_lambda` с помощью пакета Python для создания утилит командной строки Click. Обратите внимание, что мы задали значение по умолчанию для опции `--func`, при котором используется развернутая нами ранее функция `Lambda`:

```

@click.group()
@click.version_option("1.0")
def cli():
    """Вспомогательная утилита командной строки для веб-скрапинга"""

    @cli.command("lambda")
    @click.option("--func",
                  default="scrape-yahoo-dev-return_player_urls",
                  help="name of execution")
    @click.option("--payload", default='{"cli":"invoke"}',
                  help="name of payload")
    def call_lambda(func, payload):
        """Вызывает функцию Lambda

        ./wscli.py lambda
        """
        click.echo(click.style("Lambda Function invoked from cli:",
                               bg='blue', fg='white'))
        conn = lambda_connection()
        lambda_return_value = invoke_lambda(func_name=func,
                                           lambda_conn=conn,
                                           payload=payload)

```

```

formatted_json = json.dumps(lambda_return_value,
                              sort_keys=True, indent=4)
click.echo(click.style(
    "Lambda Return Value Below:", bg='blue', fg='white'))
click.echo(click.style(formatted_json, fg="red"))

if __name__ == "__main__":
    cli()

```

Выводимые этой утилитой результаты аналогичны вызову HTTP-интерфейса:

```

→ X ./wscli.py lambda \
--func=scrape-yahoo-dev-birthplace_from_urls\
--payload '{"url":["https://sports.yahoo.com/nba/players/4624/",\
"https://sports.yahoo.com/nba/players/5185/"]}'
Lambda Function invoked from cli:
{"message": "instantiate lambda client",
 "region_name": "us-east-1", "aws_service": "lambda"}
{"message": "Calling lambda function",
 "function_name": "scrape-yahoo-dev-birthplace_from_urls",
 "aws_service": "lambda", "payload":
"{\"url\": [\"https://sports.yahoo.com/nba/players/4624/\",
 \"https://sports.yahoo.com/nba/players/5185/\"]}"
 "message": null, "ResponseMetadata":
{"RequestId": "a6049115-df59-11e7-935d-bb1de9c0649d",
 "HTTPStatusCode": 200, "HTTPHeaders":
{"date": "Tue, 12 Dec 2017 16:29:43 GMT", "content-type":
 "application/json", "content-length": "118", "connection":
 "keep-alive", "x-amzn-requestid":
 "a6049115-df59-11e7-935d-bb1de9c0649d",
 "x-amzn-remapped-content-length": "0", "x-amz-executed-version":
 "$LATEST", "x-amzn-trace-id":
 "root=1-5a3003f2-2583679b2456022568ed0682;sampled=0"},
 "RetryAttempts": 0}, "StatusCode": 200,
 "ExecutedVersion": "$LATEST", "Payload":
"<botocore.response.StreamingBody object at 0x10ee37dd8>",
 "function_name": "scrape-yahoo-dev-birthplace_from_urls",
 "aws_service": "lambda", "payload":
"{\"url\": [\"https://sports.yahoo.com/nba/players/4624/\",
 \"https://sports.yahoo.com/nba/players/5185/\"]}"
Lambda Return Value Below:
{
  "https://sports.yahoo.com/nba/players/4624/": "Indianapolis",
  "https://sports.yahoo.com/nba/players/5185/": "Athens"
}

```

Завершение создания пошаговой функции

Последний этап создания пошаговой функции, как описывается в документации от AWS (<https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-creating-activity-state-machine.html>), — создание с помощью веб-интерфейса структуры конечного автомата в формате нотации объектов JavaScript (JavaScript Object Notation, JSON). Следующий код демонстрирует этот конвейер, начиная от исходных функций Lambda для скрапинга Yahoo!, сохранения данных в файле S3 и, наконец, отправки содержимого в Slack:

```
{
  "Comment": "Fetch Player Urls",
  "StartAt": "FetchUrls",
  "States": {
    "FetchUrls": {
      "Type": "Task",
      "Resource": \
        "arn:aws:lambda:us-east-1:561744971673:\
        function:scrape-yahoo-dev-return_player_urls",
      "Next": "FetchBirthplaces"
    },
    "FetchBirthplaces": {
      "Type": "Task",
      "Resource": \
        "arn:aws:lambda:us-east-1:561744971673:\
        function:scrape-yahoo-dev-birthplace_from_urls",
      "Next": "WriteToS3"
    },
    "WriteToS3": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:\
        561744971673:function:scrape-yahoo-dev-create_s3_file_from_json",
      "Next": "SendToSlack"
    },
    "SendToSlack": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:561744971673:\
        function:send_message",
      "Next": "Finish"
    },
    "Finish": {
      "Type": "Pass",
```

```

        "Result": "Finished",
        "End": true
    }
}
}

```

На рис. 7.2 было показано выполнение первой части этого конвейера. Чрезвычайно полезна возможность видеть промежуточные результаты работы конечного автомата. Кроме того, возможность мониторинга в режиме реального времени каждой части конечного автомата очень удобна для отладки.

Рисунок 7.3 демонстрирует полный конвейер с добавлением шагов записи в S3-файл и отправки содержимого в Slack. Осталось только решить, как запускать эту утилиту скрапинга — через определенный интервал времени или в ответ на какое-либо событие.

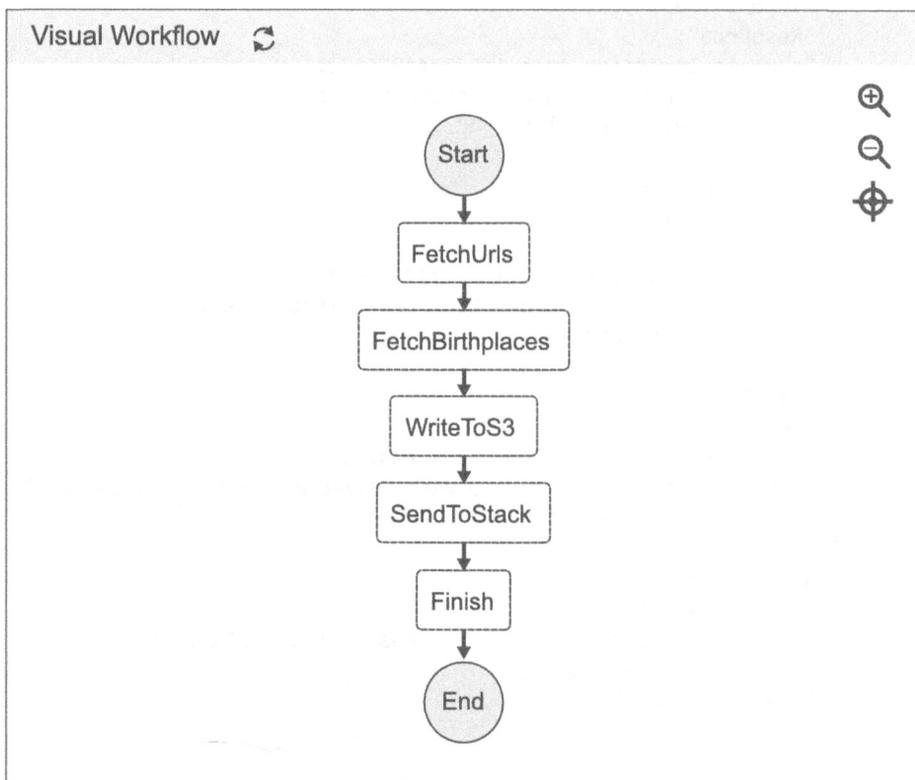


Рис. 7.3. Окончательный конвейер утилиты командной строки бота Slack

Резюме

В этой главе вы познакомились с множеством потрясающих концепций построения приложений ИИ. В ней были созданы бот Slack и утилита веб-скрапинга, соединенные затем с помощью бессерверных сервисов от AWS. В такой начальный каркас можно добавить еще много всего — например, Lambda-функцию обработки написанных на естественных языках текстов для чтения веб-страниц и получения их краткого содержимого или алгоритм кластеризации без учителя, который бы кластеризовал новых игроков НБА по произвольным атрибутам.

8

Извлечение полезной информации об управлении проектами из учетной записи GitHub-организации

Искусство джиу-джитсу совершенно.
Ответственность за ошибки лежит
на людях.

Риксон Грейси
(*Rickson Gracie*)

Эта глава посвящена двум интереснейшим задачам: использованию науки о данных для изучения вопросов управления проектами по разработке программного обеспечения и публикации утилит для исследования данных в каталоге пакетов Python (Python Package Index). Наука о данных как научная дисциплина стремительно развивается, множество статей посвящается тонкостям использования того или иного алгоритма, но лишь немногие посвящены вопросам сбора данных, создания структуры проекта и в конечном счете публикации программного обеспечения в каталоге пакетов Python. В этой главе вы найдете четкое практическое руководство по решению обеих задач. Исходный код можно найти на GitHub: <https://github.com/noahgift/devml>.

Обзор проблем, возникающих при управлении программными проектами

Несмотря на то что индустрия программного обеспечения существует уже несколько десятилетий, ее до сих пор беспокоят проблемы задержки поставки и низкого качества. Есть и другие проблемы, связанные с оценкой производительности команд и отдельных разработчиков. Зачастую предприятия — разработчики ПО находятся на переднем крае прогресса в сфере

трудовых отношений. В настоящее время в этой отрасли промышленности наблюдается тенденция к использованию фрилансеров или работающих по договору команд сотрудников в качестве вспомогательной или замещающей рабочей силы на некоторых этапах разработки программного обеспечения. В результате этого возникает очевидная задача оценки компанией способностей этих сотрудников.

Далее, достаточно мотивированных разработчиков-профессионалов интересуют пути совершенствования своих навыков. Что могут перенять они у лучших разработчиков мира? К счастью, последние генерируют своеобразные «журналы поведения», благодаря которым можно попытаться ответить на эти вопросы. Каждая фиксация изменений кода в репозитории разработчиком подает определенный сигнал.

Хорошему разработчику достаточно лишь нескольких минут обсуждения метаданных исходного кода, чтобы сказать: «А, сущая бессмыслица, я могу обойти эту систему». Можно взглянуть на профиль разработчика на GitHub и увидеть нечто поистине впечатляющее, например 3000 фиксаций изменений в год, то есть примерно десять в день ежедневно. Но если посмотреть внимательнее, может оказаться, что это всего лишь результаты работы автоматизированного сценария. Последний был создан разработчиком для обеспечения видимости активной работы в профиле или же просто «фиктивных» фиксаций изменений, добавляющих на самом деле всего лишь строку в файл README.

Контраргументом к вышеприведенному могла бы быть фраза, которую можно услышать по поводу экзаменов по математике: «Да тут можно просто сжульничать, ничего сложного». При оценке работы студента или сотрудника всегда можно сжульничать, но это не значит, что всем нужно автоматически ставить пять по алгебре или что стоит отказаться от экзаменов и тестов вообще. С точки зрения настоящего исследователя данных, подобное только делает задачу более интересной. Необходимо учитывать способы исключения фиктивных данных и шума, чтобы точно оценить производительность команд и отдельных разработчиков.

Исследовательские вопросы. Вот неполный список вопросов, который имеет смысл обдумать.

- Каковы отличительные признаки хорошего разработчика программного обеспечения?
- Каковы отличительные признаки неопытного или плохого разработчика программного обеспечения?

- ❑ Каковы отличительные признаки хорошей команды разработчиков программного обеспечения?
- ❑ Существуют ли признаки, сигнализирующие о сбойном программном обеспечении?
- ❑ Может ли управляющий программным проектом на основе какого-либо поступившего сигнала немедленно «развернуть проблемный проект на 180 градусов»?
- ❑ Существует ли наглядная разница между проектами с открытым и закрытым исходным кодом?
- ❑ Существуют ли признаки, сигнализирующие о том, что разработчик жульничает?
- ❑ Какие признаки, независимо от языка программирования, указывают на хорошего разработчика?
- ❑ Какие признаки сигнализируют о состоянии проекта на конкретном языке программирования?
- ❑ Как можно сравнить проделываемую лучшими разработчиками работу, если все репозитории находятся в полном беспорядке? Ведь при этом так легко все «скрыть».
- ❑ Как найти разработчиков в своей компании и на GitHub, похожих на ваших лучших разработчиков?
- ❑ Есть ли в вашей компании «шелуха», то есть люди, на которых не стоит полагаться? Один из способов выяснить это — узнать, фиксирует ли конкретный разработчик изменения регулярно с понедельника по пятницу? Найденные мной в последние несколько лет исследования утверждают, что плохие разработчики часто фиксируют изменения с большими и/или частыми перерывами. Лучшие разработчики — скажем, разработчики с 10–20-летним опытом — часто фиксируют изменения в коде примерно 80–90 % времени с понедельника по пятницу (даже если занимаются обучением кого-либо или консультационной поддержкой).
- ❑ Есть ли проблемы с новым менеджером проектов, директором или руководителем компании, который сводит к нулю производительность разработчиков, заставляя их проводить бесконечные часы на совещаниях?
- ❑ Есть ли у вас в компании «плохой» гениальный разработчик? Кто-то генерирующий ненадежный код с исключительной плодотворностью?

Создание исходного каркаса проекта исследования данных

При разработке нового программного решения в сфере науки о данных часто забывают про исходную структуру проекта. Для начала рекомендуется создать макет, который бы способствовал работе в целом и обеспечил высококачественную логическую схему проекта. Существует несколько вариантов организации структуры проекта, из которых мы покажем лишь один. В листинге 8.1 приведен точный результат выполнения команды `ls`, а ниже дано описание назначения каждого из элементов.

- ❑ Каталог `.circleci`. В нем находятся требуемые для сборки проекта (с помощью SaaS-сервиса сборки CircleCI) настройки. Существует множество аналогичных сервисов на основе программного обеспечения с открытым исходным кодом. Можно также воспользоваться утилитой с открытым исходным кодом, например Jenkins.
- ❑ `.gitignore`. Очень важно пропускать файлы, не являющиеся частью проекта. Про это часто забывают.
- ❑ `CODE_OF_CONDUCT.md`. Никогда не помешает включить в проект информацию о том, как, по вашему мнению, должны вести себя участники его разработки.
- ❑ `CONTRIBUTING.MD`. Явные инструкции по включению дополнений в проект очень важны для привлечения помощников, чтобы не отпугнуть потенциально ценных участников проекта.
- ❑ `LICENSE`. Наличие лицензии, например MIT или BSD, не мешает. В некоторых случаях в отсутствие у вас лицензии компании не смогут вносить вклад в ваш проект.
- ❑ Сборочный файл (`Makefile`). Часто используемый стандарт сборки проектов, существующий уже десятки лет. Это отличный инструмент для тестирования, развертывания и настройки среды разработки.
- ❑ Файл `README.md`. Хороший файл `README.md` должен отвечать на основные вопросы о проекте, например: как пользователю собрать проект и что он собой представляет. Кроме того, часто бывает полезно включить в него метки, демонстрирующие его качество, например пройденные сборки — `[!CircleCI]` (<https://circleci.com/gh/noahgift/devml.svg?style=svg>) (см. <https://circleci.com/gh/noahgift/devml>).

- ❑ Утилита командной строки. В данном примере есть утилита командной строки `dml`. Интерфейс командной строки очень удобен как для изучения возможностей библиотеки, так и в качестве интерфейса для тестирования.
- ❑ Каталог для библиотеки с файлом `__init__.py`. В корневом каталоге проекта необходимо создать каталог для библиотеки с файлом `__init__.py` в качестве признака переносимости проекта. В данном примере библиотека называется `devml`.
- ❑ Каталог `ext`. Подходящее место, например, для файлов `config.json` и `config.yml`. Лучше всего размещать не являющиеся кодом части проекта там, где на них можно будет централизованно ссылаться. Может понадобиться также подкаталог `data` для создания локальных, усеченных примеров.
- ❑ Каталог `notebooks`. Специальный каталог для блокнотов Jupiter, предназначенный для централизации разработки связанного с блокнотами кода. Кроме того, такой каталог упрощает настройку автоматического тестирования блокнотов.
- ❑ Файл `requirements.txt`. Содержит список необходимых для проекта пакетов.
- ❑ Файл `setup.py`. Этот файл конфигурации задает способ развертывания пакета Python. Его можно использовать для развертывания пакета в каталоге пакетов Python.
- ❑ Каталог `tests`. Каталог для размещения тестов.

Листинг 8.1. Структура проекта

```
(.devml) → devml git:(master) ✗ ls -la
drwxr-xr-x  3 noahgift staff   96 Oct 14 15:22 .circleci
-rw-r--r--  1 noahgift staff 1241 Oct 21 13:38 .gitignore
-rw-r--r--  1 noahgift staff 3216 Oct 15 11:44 CODE_OF_CONDUCT.md
-rw-r--r--  1 noahgift staff  357 Oct 15 11:44 CONTRIBUTING.md
-rw-r--r--  1 noahgift staff 1066 Oct 14 14:10 LICENSE
-rw-r--r--  1 noahgift staff  464 Oct 21 14:17 Makefile
-rw-r--r--  1 noahgift staff 13015 Oct 21 19:59 README.md
-rwxr-xr-x  1 noahgift staff  9326 Oct 21 11:53 dml
drwxr-xr-x  4 noahgift staff   128 Oct 20 15:20 ext
drwxr-xr-x  7 noahgift staff   224 Oct 22 11:25 notebooks
-rw-r--r--  1 noahgift staff   117 Oct 18 19:16 requirements.txt
-rw-r--r--  1 noahgift staff 1197 Oct 21 14:07 setup.py
drwxr-xr-x 12 noahgift staff   384 Oct 18 10:46 tests
```

Сбор и преобразование данных

Как обычно, самая сложная часть задачи состоит в поиске способа сбора данных и преобразования их в нечто удобное для наших целей. Эта подзадача состоит из нескольких частей. Первая: получение данных из отдельного репозитория и создание на их основе объекта `DataFrame` библиотеки `Pandas`. Для этого мы создадим в каталоге `devml` новый модуль под названием `mkdata.py`, предназначенный для преобразования метаданных репозитория `Git` в объект `DataFrame` библиотеки `Pandas`.

Выбранный фрагмент этого модуля вы можете найти здесь: <https://github.com/noahgift/devml/blob/master/devml/mkdata.py>. Функция `log_to_dict` принимает на входе путь к извлеченному на диск содержимому репозитория `Git` и преобразует результаты выполнения команды `Git`:

```
def log_to_dict(path):
    """Преобразует журнал Git в словарь языка Python"""

    os.chdir(path) #Переходим в другой каталог для обработки журнала Git
    repo_name = generate_repo_name()
    p = Popen(GIT_LOG_CMD, shell=True, stdout=PIPE)
    (git_log, _) = p.communicate()
    try:
        git_log = git_log.decode('utf8').\
            strip('\n\x1e').split("\x1e")
    except UnicodeDecodeError:
        log.exception("utf8 encoding is incorrect,
            trying ISO-8859-1")
        git_log = git_log.decode('ISO-8859-1').\
            strip('\n\x1e').split("\x1e")

    git_log = [row.strip().split("\x1f") for row in git_log]
    git_log = [dict(list(zip(GIT_COMMIT_FIELDS, row)))\
        for row in git_log]
    for dictionary in git_log:
        dictionary["repo"]=repo_name
    repo_msg = "Found %s Messages For Repo: %s" %\
        (len(git_log), repo_name)
    log.info(repo_msg)
    return git_log
```

В следующих двух функциях вышеприведенной функции передается путь на диске. Обратите внимание, что журналы хранятся в виде элементов

списка, который затем используется для создания объекта DataFrame в Pandas:

```
def create_org_df(path):
    """Возвращает объект DataFrame Pandas для GitHub-организации"""

    original_cwd = os.getcwd()
    logs = create_org_logs(path)
    org_df = pd.DataFrame.from_dict(logs)
    #Преобразует дату [Git] в тип datetime [библиотеки Pandas]
    datetime_converted_df = convert_datetime(org_df)
    #Добавляем индекс для даты
    converted_df = date_index(datetime_converted_df)
    new_cwd = os.getcwd()
    cd_msg = "Changing back to original cwd: %s from %s" % \
        (original_cwd, new_cwd)
    log.info(cd_msg)
    os.chdir(original_cwd)
    return converted_df

def create_org_logs(path):
    """Проходим по всем путям текущего рабочего каталога, формируя
    словари на основе журналов"""

    combined_log = []
    for sdir in subdirs(path):
        repo_msg = "Processing Repo: %s" % sdir
        log.info(repo_msg)
        combined_log += log_to_dict(sdir)
    log_entry_msg = "Found a total log entries: %s" % \
        len(combined_log)
    log.info(log_entry_msg)
    return combined_log
```

Код в действии выглядит следующим образом при запуске без сбора данных в объект DataFrame:

```
In [5]: res = create_org_logs("/Users/noahgift/src/flask")
2017-10-22 17:36:02,380 - devml.mkdata - INFO - Found repo:\
/Users/noahgift/src/flask/flask
In [11]: res[0]
Out[11]:
{'author_email': 'rgerganov@gmail.com',
 'author_name': 'Radoslav Gerganov',
 'date': 'Fri Oct 13 04:53:50 2017',
```

```
'id': '9291ead32e2fc8b13cef825186c968944e9ff344',
'message': 'Fix typo in logging.rst (#2492)',
'repo': b'flask'}
```

Вторая часть кода — создание объекта `DataFrame` — имеет такой вид:

```
res = create_org_df("/Users/noahgift/src/flask")
In [14]: res.describe()
Out[14]:
```

```
      commits
count  9552.0
mean     1.0
std      0.0
min     1.0
25%     1.0
50%     1.0
75%     1.0
max     1.0
```

Если не вдаваться в подробности, то именно так и выглядит паттерн для получения специализированных данных из сторонних источников, например журналов Git. Чтобы узнать больше подробностей, не помешает взглянуть на полный исходный код.

Обработка GitHub-организации в целом

Естественный следующий этап после написания кода для преобразования Git-репозитория на диске в объекты `DataFrame` — сбор данных из нескольких репозиториях сразу, то есть целой GitHub-организации. Одна из главных проблем с анализом одного отдельного репозитория — то, что в контексте компании он представляет собой лишь обрывок информации. Один из способов решения этой проблемы — обратиться к API GitHub и извлечь репозитории программным путем. Полный исходный код данного решения приведен на https://github.com/noahgift/devml/blob/master/devml/fetch_repo.py. Вот его избранные фрагменты:

```
def clone_org_repos(oauth_token, org, dest, branch="master"):
    """Клонируем все репозитории GitHub-организаций и возвращаем
    экземпляры репозитория
    """
    if not validate_checkout_root(dest):
```

```

return False

repo_instances = []
repos = org_repo_names(oauth_token, org)
count = 0
for name, url in list(repos.items()):
    count += 1
    log_msg = "Cloning Repo # %s REPO NAME: %s , URL: %s " %\
              (count, name, url)
    log.info(log_msg)
    try:
        repo = clone_remote_repo(name, url, dest, branch=branch)
        repo_instances.append(repo)
    except GitCommandError:
        log.exception("NO MASTER BRANCH...SKIPPING")
return repo_instances

```

Основную работу берут на себя пакеты PyGithub и gitpython. После запуска программа в цикле находит все репозитории из API и клонирует их. С помощью вышеприведенного кода можно затем создать единый объект DataFrame.

Формирование предметно-ориентированной статистики

Все вышеприведенное было сделано с одной целью: исследовать собранные данные и сформировать предметно-ориентированную статистику. Для этого мы создадим файл stats.py, полное его содержимое вы можете найти здесь: <https://github.com/noahgift/devml/blob/master/devml/stats.py>.

Основная часть данного файла — функция author_unique_active_days. Она определяет по записям DataFrame число дней, когда разработчик вел активную деятельность. Это уникальная предметно-ориентированная статистика, редко упоминаемая в обсуждении статистики репозитория в исходном коде.

Ниже показана основная функция:

```

def author_unique_active_days(df, sort_by="active_days"):
    """Объект DataFrame с уникальными днями активности,
       отсортировано по именам разработчика в порядке убывания

    author_name  unique_days
    46  Armin Ronacher  271

```

260 Markus Unterwaditzer 145
 ""

```

author_list = []
count_list = []
duration_active_list = []
ad = author_active_days(df)
for author in ad.index:
    author_list.append(author)
    vals = ad.loc[author]
    vals.dropna(inplace=True)
    vals.drop_duplicates(inplace=True)
    vals.sort_values(axis=0, inplace=True)
    vals.reset_index(drop=True, inplace=True)
    count_list.append(vals.count())
    duration_active_list.append(vals[len(vals)-1]-vals[0])
df_author_ud = DataFrame()
df_author_ud["author_name"] = author_list
df_author_ud["active_days"] = count_list
df_author_ud["active_duration"] = duration_active_list
df_author_ud["active_ratio"] = \
    round(df_author_ud["active_days"]/\
          df_author_ud["active_duration"].dt.days, 2)
df_author_ud = df_author_ud.iloc[1:] #первая строка =
df_author_ud = df_author_ud.sort_values(\
    by=sort_by, ascending=False)
return df_author_ud

```

При вызове из оболочки IPython она возвращает следующие результаты:

```
In [18]: from devml.stats import author_unique_active_days
```

```
In [19]: active_days = author_unique_active_days(df)
```

```
In [20]: active_days.head()
```

```
Out[20]:
```

	author_name	active_days	active_duration	active_ratio
46	Armin Ronacher	241	2490 days	0.10
260	Markus Unterwaditzer	71	1672 days	0.04
119	David Lord	58	710 days	0.08
352	Ron DuPlain	47	785 days	0.06
107	Daniel Neuhäuser	19	435 days	0.04

В этой статистике вы можете видеть коэффициент — `active_ratio`, — отражающий часть времени от начала работы над проектом и до конца, когда

соответствующий разработчик активно фиксировал изменения кода. В подобных показателях интересно, в частности, то, что они отражают степень вовлеченности в проект и можно провести интересные параллели между лучшими разработчиками открытого исходного кода. В следующем разделе мы подключим эти базовые компоненты к утилите командной строки и сравним два различных проекта с открытым исходным кодом с помощью созданного нами кода.

Подключение проекта по исследованию данных к интерфейсу командной строки

В первой половине данной главы мы создали компоненты, необходимые для начала анализа. В этом разделе мы подключим их к гибкой утилите командной строки, использующей фреймворк Click. Полный исходный код для пакета `dml` можно найти здесь: <https://github.com/noahgift/devml/blob/master/dml>. Отдельные существенные его фрагменты показаны ниже.

Сначала импортируем библиотеку и фреймворк Click:

```
#!/usr/bin/env python
import os

import click

from devml import state
from devml import fetch_repo
from devml import __version__
from devml import mkdata
from devml import stats
from devml import org_stats
from devml import post_processing
```

Затем подключаем к утилите вышеприведенный код, для чего требуется лишь несколько его строк:

```
@gstats.command("activity")
@click.option("--path", default=CHECKOUT_DIR, help="path to org")
@click.option("--sort", default="active_days",
              help="can sort by: active_days, active_ratio, active_duration")
```

```
def activity(path, sort):
    """Создает статистику активности

    Пример запускается после извлечения данных из хранилища:
    python dml.py gstats activity --path\
        /Users/noah/src/wulio/checkout
    """

    org_df = mkdata.create_org_df(path)
    activity_counts = stats.author_unique_active_days(\
        org_df, sort_by=sort)
    click.echo(activity_counts)
```

Использование этой утилиты из командной строки выглядит следующим образом:

```
# Коэффициент активности разработчиков Linux
dml gstats activity --path /Users/noahgift/src/linux\
    --sort active_days
```

author_name	active_days	active_duration	active_ratio
Takashi Iwai	1677	4590 days	0.370000
Eric Dumazet	1460	4504 days	0.320000
David S. Miller	1428	4513 days	0.320000
Johannes Berg	1329	4328 days	0.310000
Linus Torvalds	1281	4565 days	0.280000
Al Viro	1249	4562 days	0.270000
Mauro Carvalho Chehab	1227	4464 days	0.270000
Mark Brown	1198	4187 days	0.290000
Dan Carpenter	1158	3972 days	0.290000
Russell King	1141	4602 days	0.250000
Axel Lin	1040	2720 days	0.380000
Alex Deucher	1036	3497 days	0.300000

```
# Коэффициент активности разработчиков CPython
```

	author_name	active_days	active_duration	active_ratio
146	Guido van Rossum	2256	9673 days	0.230000
301	Raymond Hettinger	1361	5635 days	0.240000
128	Fred Drake	1239	5335 days	0.230000
47	Benjamin Peterson	1234	3494 days	0.350000
132	Georg Brandl	1080	4091 days	0.260000
375	Victor Stinner	980	2818 days	0.350000

235	Martin v. Löwis	958	5266 days	0.180000
36	Antoine Pitrou	883	3376 days	0.260000
362	Tim Peters	869	5060 days	0.170000
164	Jack Jansen	800	4998 days	0.160000
24	Andrew M. Kuchling	743	4632 days	0.160000
330	Serhiy Storchaka	720	1759 days	0.410000
44	Barry Warsaw	696	8485 days	0.080000
52	Brett Cannon	681	5278 days	0.130000
262	Neal Norwitz	559	2573 days	0.220000

Согласно этой статистике, вероятность того, что создатель языка Python Гвидо ван Россума будет работать в заданный день, равна 23 %, а для создателя операционной системы Linux Линуса Торвальдса она равна 28 %. В подобной статистике интересно то, что она отражает поведение разработчиков за длительный период времени. В случае проекта CPython многие из разработчиков трудятся еще и полный рабочий день, что еще невероятнее. Интересно было бы также посмотреть историю разработчиков компании (объединив для этого все доступные репозитории). Я обратил внимание, что в некоторых случаях весьма опытные разработчики при полной занятости выдают код с коэффициентом активности около 85 %.

Исследование GitHub-организаций с помощью блокнота Jupiter

Проект Pallets на GitHub. В числе проблем изучения лишь одного репозитория — то, что это всего часть данных. Ранее созданный код дает нам возможность клонировать и изучать GitHub-организацию в целом. Среди популярных GitHub-организаций можно выделить проект Pallets (<https://github.com/pallets>). В его состав входит множество популярных проектов, например Click и Flask. Блокнот Jupiter для этого нашего исследования можно найти здесь: https://github.com/noahgift/devml/blob/master/notebooks/github_data_exploration.ipynb.

Для запуска Jupiter наберите в командной строке команду `jupyter notebook`. Затем импортируйте нужные библиотеки:

```
In [3]: import sys;sys.path.append("../")
...: import pandas as pd
...: from pandas import DataFrame
...: import seaborn as sns
...: import matplotlib.pyplot as plt
...: from sklearn.cluster import KMeans
```

```

...: %matplotlib inline
...: from IPython.core.display import display, HTML
...: display(HTML("<style>.container {\
      width:100% !important; }</style>"))

```

Далее воспользуемся следующим кодом для скачивания GitHub-организации:

```
In [4]: from devml import (mkdata, stats, state, fetch_repo, ts)
```

```
In [5]: dest, token, org = state.get_project_metadata(\
      "../project/config.json")
```

```
In [6]: fetch_repo.clone_org_repos(token, org,
      ...:     dest, branch="master")
```

Out[6]:

```

[<git.Repo "/tmp/checkout/flask/.git">,
 <git.Repo "/tmp/checkout/pallets-sphinx-themes/.git">,
 <git.Repo "/tmp/checkout/markupsafe/.git">,
 <git.Repo "/tmp/checkout/jinja/.git">,
 <git.Repo "/tmp/checkout/werkzeug/.git">,
 <git.Repo "/tmp/checkout/itsdangerous/.git">,
 <git.Repo "/tmp/checkout/flask-website/.git">,
 <git.Repo "/tmp/checkout/click/.git">,
 <git.Repo "/tmp/checkout/flask-snippets/.git">,
 <git.Repo "/tmp/checkout/flask-docs/.git">,
 <git.Repo "/tmp/checkout/flask-ext-migrate/.git">,
 <git.Repo "/tmp/checkout/pocoo-sphinx-themes/.git">,
 <git.Repo "/tmp/checkout/website/.git">,
 <git.Repo "/tmp/checkout/meta/.git">]

```

После скачивания кода на диск можно его преобразовать в объект DataFrame библиотеки Pandas:

```
In [7]: df = mkdata.create_org_df(path="/tmp/checkout")
```

```
In [9]: df.describe()
```

Out[9]:

	commits
count	8315.0
mean	1.0
std	0.0
min	1.0
25%	1.0
50%	1.0
75%	1.0
max	1.0

И подсчитать количество дней, в которые разработчики вели активную деятельность:

```
In [10]: df_author_ud = stats.author_unique_active_days(df)
...:
In [11]: df_author_ud.head(10)
Out[11]:
```

	author_name	active_days	active_duration	active_ratio
86	Armin Ronacher	941	3817 days	0.25
499	Markus Unterwaditzer	238	1767 days	0.13
216	David Lord	94	710 days	0.13
663	Ron DuPlain	56	854 days	0.07
297	Georg Brandl	41	1337 days	0.03
196	Daniel Neuhäuser	36	435 days	0.08
169	Christopher Grebs	27	1515 days	0.02
665	Ronny Pfannschmidt	23	2913 days	0.01
448	Keyan Pishdadian	21	882 days	0.02
712	Simon Sapin	21	793 days	0.03

Наконец, с помощью функции `sns.barplot` можно превратить эти данные в столбчатый график из библиотеки Seaborn, как показано на рис. 8.1, с десятью разработчиками, внесшими основной вклад в GitHub-организацию, по дням их активности в рамках проекта, то есть дням, когда они действительно фиксировали изменения кода. И неудивительно, что основные авторы большинства проектов демонстрируют втрое большую активность, чем большинство других участников.

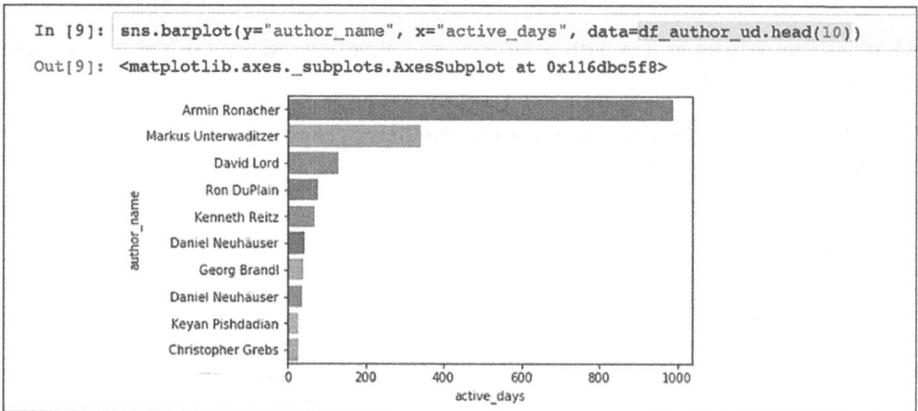


Рис. 8.1. Seaborn-диаграмма дней активности

Вероятно, некоторые из подобных наблюдений можно экстраполировать и на проекты с закрытым исходным кодом из репозитория GitHub-организации. Дни активности — удобный показатель вовлеченности, который можно включить в множество метрик для оценивания эффективности работы команд разработчиков и проектов.

Изучаем метаданные файлов проекта CPython

Следующий блокнот Jupiter, который мы обсудим, будет связан с исследованием метаданных проекта CPython. Его можно найти здесь: https://github.com/noahgift/devml/blob/master/notebooks/repo_file_exploration.ipynb. А проект CPython — тут: <https://github.com/python/cpython>; этот репозиторий используется для разработки языка Python.

Одна из возможных метрик — относительный пересмотр (редактирование) кода (*relative churn*); прочитать о нем подробнее можно в статье, опубликованной исследовательским подразделением компании «Майкрософт»: <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/icse05churn.pdf>. Эта статья утверждает, что «повышение относительного коэффициента пересмотра кода сопровождается повышением концентрации ошибок в системе». Из данного утверждения можно сделать вывод о том, что на основе слишком большого числа изменений файла можно прогнозировать наличие в нем ошибок.

В этом новом блокноте мы опять импортируем необходимые для дальнейшего исследования модули:

```
In [1]: import sys;sys.path.append("../")
...: import pandas as pd
...: from pandas import DataFrame
...: import seaborn as sns
...: import matplotlib.pyplot as plt
...: from sklearn.cluster import KMeans
...: %matplotlib inline
...: from IPython.core.display import display, HTML
...: display(HTML("<style>.container \
    { width:100% !important; }</style>"))
```

Далее генерируем метрики относительного пересмотра кода:

```
In [2]: from devml.post_processing import (
        git_churn_df, file_len, git_populate_file_metadata)

In [3]: df = git_churn_df(path="/Users/noahgift/src/cpython")
2017-10-23 06:51:00,256 - devml.post_processing - INFO -
        Running churn cmd: [git log --name-only
        --pretty=format:] at path [/Users/noahgift/src/cpython]

In [4]: df.head()
Out[4]:
```

	files	churn_count
0	b'Lib/test/test_struct.py'	178
1	b'Lib/test/test_zipimport.py'	78
2	b'Misc/NEWS.d/next/Core'	351
3	b'and'	351
4	b'Builtins/2017-10-13-20-01-47.bpo-31781.cXE9S...	1

Теперь можно воспользоваться несколькими фильтрами из библиотеки Pandas для поиска наиболее часто редактируемых файлов с расширением, соответствующим языку Python. Результат приведен на рис. 8.2.

На рисунке можно, в частности, заметить, что тесты подвергаются активному пересмотру кода. Это стоит исследовать подробнее. Означает ли данный факт, что сами тесты тоже содержат ошибки? У нас может получиться весьма захватывающее исследование. Кроме того, существует несколько модулей Python с исключительно высоким относительным коэффициентом пересмотра кода, например модуль `string.py`: <https://github.com/python/cpython/blob/master/Lib/string.py>. Если просмотреть исходный код этого файла, можно отметить, что он очень сложен для своего размера и содержит метаклассы. Возможно, именно сложность повышает его предрасположенность к ошибкам. Похоже, что имеет смысл продолжить исследование этого модуля посредством науки о данных.

Далее можно сформировать описательную статистику и взглянуть на медианные значения для этого проекта. Статистика показывает, в частности, что за пару десятков лет существования проекта и более 100 000 фиксаций изменений в его коде медианный файл насчитывает около 146 строк и менялся пять раз, а его относительный коэффициент пересмотра кода равен 10 %.

```
In [22]: python_files_df = metadata_df[metadata_df.extension == ".py"]
line_python = python_files_df[python_files_df.line_count > 40]
line_python.sort_values(by="relative_churn", ascending=False).head(15)
```

```
Out[22]:
```

	files	churn_count	line_count	extension	relative_churn
15	b'Lib/test/regtest.py'	627	50.0	.py	12.54
196	b'Lib/test/test_datetime.py'	165	57.0	.py	2.89
197	b'Lib/io.py'	165	99.0	.py	1.67
430	b'Lib/test/test_sundry.py'	91	56.0	.py	1.62
269	b'Lib/test/test__all__.py'	128	109.0	.py	1.17
1120	b'Lib/test/test_userstring.py'	40	44.0	.py	0.91
827	b'Lib/email/__init__.py'	52	62.0	.py	0.84
85	b'Lib/test/test_support.py'	262	461.0	.py	0.57
1006	b'Lib/test/test_select.py'	44	82.0	.py	0.54
1474	b'Lib/lib2to3/fixes/fix_itertools_imports.py'	30	57.0	.py	0.53
346	b'Doc/conf.py'	106	206.0	.py	0.51
222	b'Lib/string.py'	151	305.0	.py	0.50
804	b'Lib/test/test_normalization.py'	53	108.0	.py	0.49
592	b'Lib/test/test_fcntl.py'	68	152.0	.py	0.45
602	b'Lib/test/test_winsound.py'	67	148.0	.py	0.45

Рис. 8.2. Наиболее часто редактируемые файлы с расширением .py в проекте CPython

Следовательно, оптимально будет создавать маленькие файлы и менять их лишь несколько раз за годы их существования:

```
In [16]: metadata_df.median()
Out[16]:
churn_count      5.0
line_count      146.0
relative_churn   0.1
dtype: float64
```

Если сгенерировать Seaborn-график относительного пересмотра кода, то эти паттерны станут еще более ясными:

```
In [18]: import matplotlib.pyplot as plt
...: plt.figure(figsize=(10,10))
...: python_files_df = \
...:     metadata_df[metadata_df.extension == ".py"]
...: line_python = \
```

```
python_files_df[python_files_df.line_count > 40]
...: line_python_sorted = \
    line_python.sort_values(by="relative_churn",
        ascending=False).head(15)
...: sns.barplot(
    y="files", x="relative_churn", data=line_python_sorted)
...: plt.title('Top 15 CPython Absolute and Relative Churn')
...: plt.show()
```

На рис. 8.3 модуль `regtest.py` выделяется числом своих изменений, и опять же вполне понятно, почему он менялся так часто. Хотя файл и невелик, обычно тестирование регрессии представляет собой весьма непростую задачу. Он вполне может оказаться критическим участком кода, который заслуживает пристального внимания: <https://github.com/python/cpython/blob/master/Lib/test/regtest.py>.

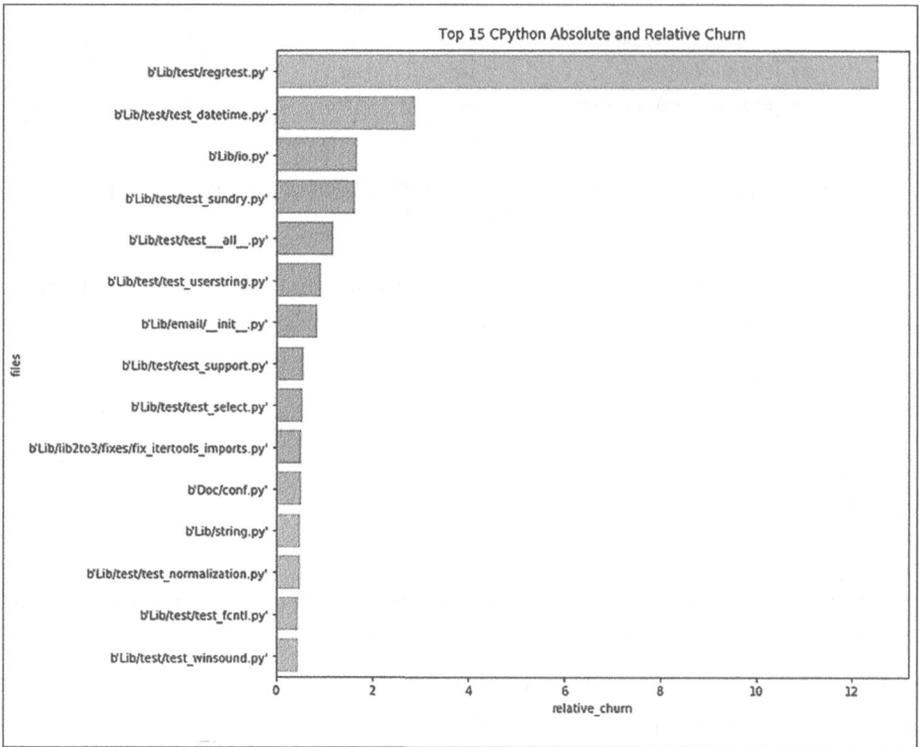


Рис. 8.3. Наиболее часто редактируемые файлы с расширением `.py` в проекте CPython

Изучаем файлы, удаленные из проекта CPython

Еще одна область изучения — файлы, удаленные за время существования проекта. Возможны несколько направлений исследований, основанных на подобном изучении, например предсказание потенциального удаления файла в случае слишком высокого значения относительного коэффициента его пересмотра. Для изучения удаленных файлов мы сначала создадим еще одну функцию в файле `post_processing.py`: https://github.com/noahgift/devml/blob/master/devml/post_processing.py:

```
FILES_DELETED_CMD=\
    'git log --diff-filter=D --summary | grep delete'

def files_deleted_match(output):
    """Извлекаем названия файлов из результатов работы подпроцесса

    то есть:
    wcase/templates/hello.html\n delete mode 100644
    Отбрасываем все, кроме пути к файлу
    """

    files = []
    integers_match_pattern = '^[-+]?[0-9]+$'
    for line in output.split():
        if line == b"delete":
            continue
        elif line == b"mode":
            continue
        elif re.match(integers_match_pattern, line.decode("utf-8")):
            continue
        else:
            files.append(line)
    return files
```

Эта функция ищет сообщения `delete` в журнале Git, сопоставляет с шаблоном и извлекает названия файлов в список с целью создания объекта `DataFrame` библиотеки `Pandas`, которым мы далее сможем воспользоваться в блокноте `Jupyter`:

```
In [19]: from devml.post_processing import git_deleted_files
...: deletion_counts = git_deleted_files(
...:     "/Users/noahgift/src/cpython")
```

Взглянем на несколько последних записей для удаленных файлов:

```
In [21]: deletion_counts.tail()
Out[21]:
```

	files	ext
8812	b'Mac/mwerks/mwerksglue.c'	.c
8813	b'Modules/version.c'	.c
8814	b'Modules/Setup.irix5'	.irix5
8815	b'Modules/Setup.guido'	.guido
8816	b'Modules/Setup.minix'	.minix

Далее выясним, существует ли паттерн, отличающий удаленные файлы от оставшихся в проекте. Для этого необходимо выполнить соединение объекта DataFrame, содержащего информацию об удаленных файлах:

```
In [22]: all_files = metadata_df['files']
...: deleted_files = deletion_counts['files']
...: membership = all_files.isin(deleted_files)
...:
```

```
In [23]: metadata_df["deleted_files"] = membership
```

```
In [24]: metadata_df.loc[metadata_df["deleted_files"] ==\
True].median()
```

```
Out[24]:
```

churn_count	4.000
line_count	91.500
relative_churn	0.145
deleted_files	1.000

dtype: float64

```
In [25]: metadata_df.loc[metadata_df["deleted_files"] ==\
False].median()
```

```
Out[25]:
```

churn_count	9.0
line_count	149.0
relative_churn	0.1
deleted_files	0.0

dtype: float64

Медианные значения для удаленных файлов отличаются от значений для файлов, оставшихся в репозитории, в основном тем, что у удаленных выше относительный коэффициент пересмотра. Возможно, обычно удаляются проблемные файлы? Ответить на данный вопрос без дополнительного его

изучения нельзя. Создадим на основе этого объекта DataFrame карту интенсивности корреляции с помощью библиотеки Seaborn:

```
In [26]: sns.heatmap(metadata_df.corr(), annot=True)
```

Данная карта представлена на рис. 8.4. Она показывает наличие очень небольшой положительной корреляции между относительным коэффициентом пересмотра и удалением файла. Этот сигнал можно включить в модель машинного обучения для предсказания вероятности удаления файла.

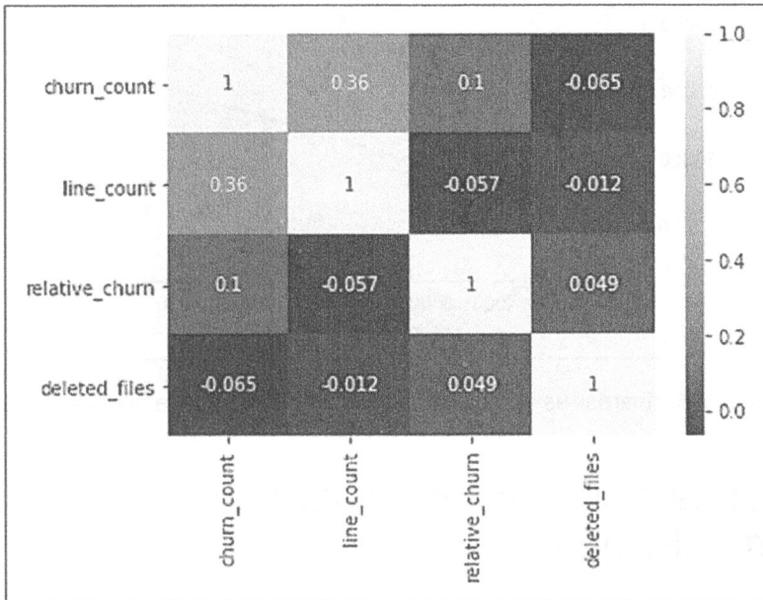


Рис. 8.4. Карта интенсивности корреляции удаления файлов

А теперь нарисуем еще одну, последнюю, диаграмму рассеяния, которая продемонстрирует различия между удаленными и оставшимися в репозитории файлами:

```
In [27]: sns.lmplot(x="churn_count", y="line_count",
                    hue="deleted_files", data=metadata_df)
```

На рис. 8.5 показаны три измерения: количество строк, количество пересмотров и булева категория удаления файла, принимающая значения True/False.

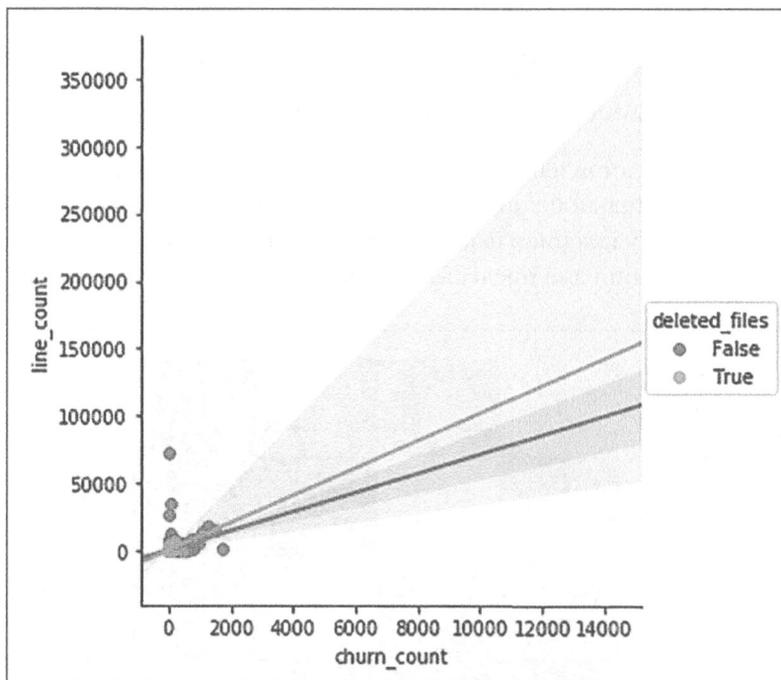


Рис. 8.5. Диаграмма рассеяния для числа строк и числа пересмотров

Развертывание проекта в каталоге пакетов Python

После всех трудов по созданию библиотеки и утилиты командной строки имеет смысл поделиться проектом с другими разработчиками, отправив его в каталог пакетов Python. Для этого потребуется лишь несколько шагов.

1. Создать учетную запись по адресу <https://pypi.python.org/pypi>.
2. Установить утилиту twine с помощью команды `pip install twine`.
3. Создать файл `setup.py`.
4. Добавить в сборочный файл шаг развертывания (`deploy`).

Начнем с шага 3. Ниже приведено содержимое файла `setup.py`. Две важнейшие его части — раздел `packages`, гарантирующий установку библиотеки, и раздел `scripts`. Сценарий целиком вы можете найти здесь: <https://github.com/noahgift/devml/blob/master/setup.py>.

```

import sys
if sys.version_info < (3,6):
    sys.exit('Sorry, Python < 3.6 is not supported')
import os
from setuptools import setup

from devml import __version__

if os.path.exists('README.rst'):
    LONG = open('README.rst').read()

setup(
    name='devml',
    version=__version__,
    url='https://github.com/noahgift/devml',
    license='MIT',
    author='Noah Gift',
    author_email='consulting@noahgift.com',
    description="""Machine Learning, Statistics
    and Utilities around Developer Productivity,
    Company Productivity and Project Productivity""",
    long_description=LONG,
    packages=['devml'],
    include_package_data=True,
    zip_safe=False,
    platforms='any',
    install_requires=[
        'pandas',
        'click',
        'PyGithub',
        'gitpython',
        'sensible',
        'scipy',
        'numpy',
    ],
    classifiers=[
        'Development Status :: 4 - Beta',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python',
        'Programming Language :: Python :: 3.6',
        'Topic :: Software Development \
        :: Libraries :: Python Modules'
    ],
    scripts=["dml"],
)

```

Директива `scripts` далее обеспечит внесение утилиты `dml` в пути всех пользователей, установивших модуль с помощью `pip`.

Нам осталось только внести следующие дополнения в сборочный файл:

`deploy-pypi:`

```
    pandoc --from=markdown --to=rst README.md -o README.rst
    python setup.py check --restructuredtext --strict --metadata
    rm -rf dist
    python setup.py sdist
    twine upload dist/*
    rm -f README.rst
```

Полное содержимое сборочного файла можно найти на GitHub: <https://github.com/noahgift/devml/blob/master/Makefile>.

Наконец, процесс развертывания выглядит следующим образом:

```
(.devml) → devml git:(master) X make deploy-pypi
pandoc --from=markdown --to=rst README.md -o README.rst
python setup.py check --restructuredtext --strict --metadata
running check
rm -rf dist
python setup.py sdist
running sdist
running egg_info
writing devml.egg-info/PKG-INFO
writing dependency_links to devml.egg-info/dependency_links.txt
....
running check
creating devml-0.5.1
creating devml-0.5.1/devml
creating devml-0.5.1/devml.egg-info
copying files to devml-0.5.1...
....
Writing devml-0.5.1/setup.cfg
creating dist
Creating tar archive
removing 'devml-0.5.1' (and everything under it)
twine upload dist/*
Uploading distributions to https://upload.pypi.org/legacy/
Enter your username:
```

Резюме

В первой половине этой главы мы создали базовый каркас приложения для исследования данных и разобрались в его составных частях. Во второй половине с помощью блокнота `Jupyter` мы углубились в исследование проекта `CPython` на `GitHub`. Наконец, было показано, как подготовить проект утилиты командной строки для исследования данных, `DEVML`, к развертыванию в каталоге пакетов `Python`. Эта глава отлично подходит для изучения теми разработчиками утилит для науки о данных, которые хотели бы создавать решения, поставляемые в виде как библиотек `Python`, так и утилит командной строки.

Подобно остальным главам данной книги, это лишь начало для компании или разрабатываемого компанией приложения ИИ. С помощью описанных в остальных главах книги методик можно создать на основе `Flask` или `Chalice` различные API, пригодные для поставки его в промышленную эксплуатацию.

9

Динамическая оптимизация виртуальных узлов EC2 в AWS

Джиу-джитсу — гонка; допустив в бою с превосходящим вас противником ошибку, вы никогда не сможете наверстать упущенное.

*Луис «Лимао» Хередиа
(Luis “Limaо” Heredia),
пятикратный чемпион обеих Америк
по бразильскому джиу-джитсу*

При промышленной эксплуатации приложений машинного обучения часто возникает проблема с управлением заданиями. Примеры заданий могут быть самыми разнообразными, начиная от скрапинга содержимого сайта с целью генерации описательной статистики по большому CSV-файлу и до программного обновления модели машинного обучения с учителем. Управление заданиями — одна из самых сложных задач науки о данных, и существует множество путей ее решения. Кроме того, выполнение заданий может быстро стать весьма дорогостоящим в смысле расхода ресурсов. В этой главе мы рассмотрим несколько различных технологий AWS и я приведу для каждой из них примеры.

Выполнение заданий на платформе AWS

Спотовые виртуальные узлы

Как для промышленных систем машинного обучения, так и для экспериментов необходимо хорошее понимание принципов работы спотовых виртуальных узлов. Полезно будет посмотреть официальное видеоруководство AWS по ним (<https://aws.amazon.com/ec2/spot/spot-tutorials/>), это может помочь

в понимании части изложенного далее материала. Вот некоторые базовые сведения о спотовых виртуальных узлах.

- Обычно они на 50–60 % дешевле, чем зарезервированные виртуальные узлы.
- Полезны во многих отраслях промышленности и сценариях использования:
 - в научных исследованиях;
 - финансовых услугах;
 - компаниях, занимающихся обработкой видео и изображений;
 - сканировании сайтов/обработке данных.
- Насчитывают четыре распространенные архитектуры:
 - Hadoop/сервис Elastic Map Reduce (EMR);
 - запись данных о состоянии в контрольных точках (запись на диск результатов по мере их обработки);
 - кластерную распределенную вычислительную систему (например, StarCluster, <http://star.mit.edu/cluster/docs/latest/index.html>);
 - архитектуру на основе очередей.

Теория спотовых виртуальных узлов и история цен на них

Разобраться с ценообразованием спотовых виртуальных узлов не так уж просто. Одно из первых препятствий состоит в том, что нужно понять, какой именно тип виртуального узла на самом деле требуется для ваших заданий. Даже это нелегко, ведь в зависимости от типа спотовой архитектуры возможны различные узкие места: в некоторых сеть, в других — операции дискового ввода/вывода или процессор. Кроме того, для фреймворка заданий сам способ проектирования кода представляет проблему.

На рис. 9.1 проиллюстрирован закон Амдала, описывающий ограничения параллелизма на практике. Он гласит, что рост производительности ограничивается последовательными элементами программы (формула 9.1). Например, служебные операции при распределении задания могут включать последовательные компоненты. Наверное, лучшим примером будет задание, занимающее 100 с, но включающее 5 с выполнения команды `time.sleep()`, о которой разработчик забыл. При распараллеливании подобного задания теоретически возможный предел роста производительности будет равен 20. В случае скрытой приостановки работы (а такое случается) даже самый быстрый процессор или диск никак не смогут сделать производительность выше соответствующего предела.

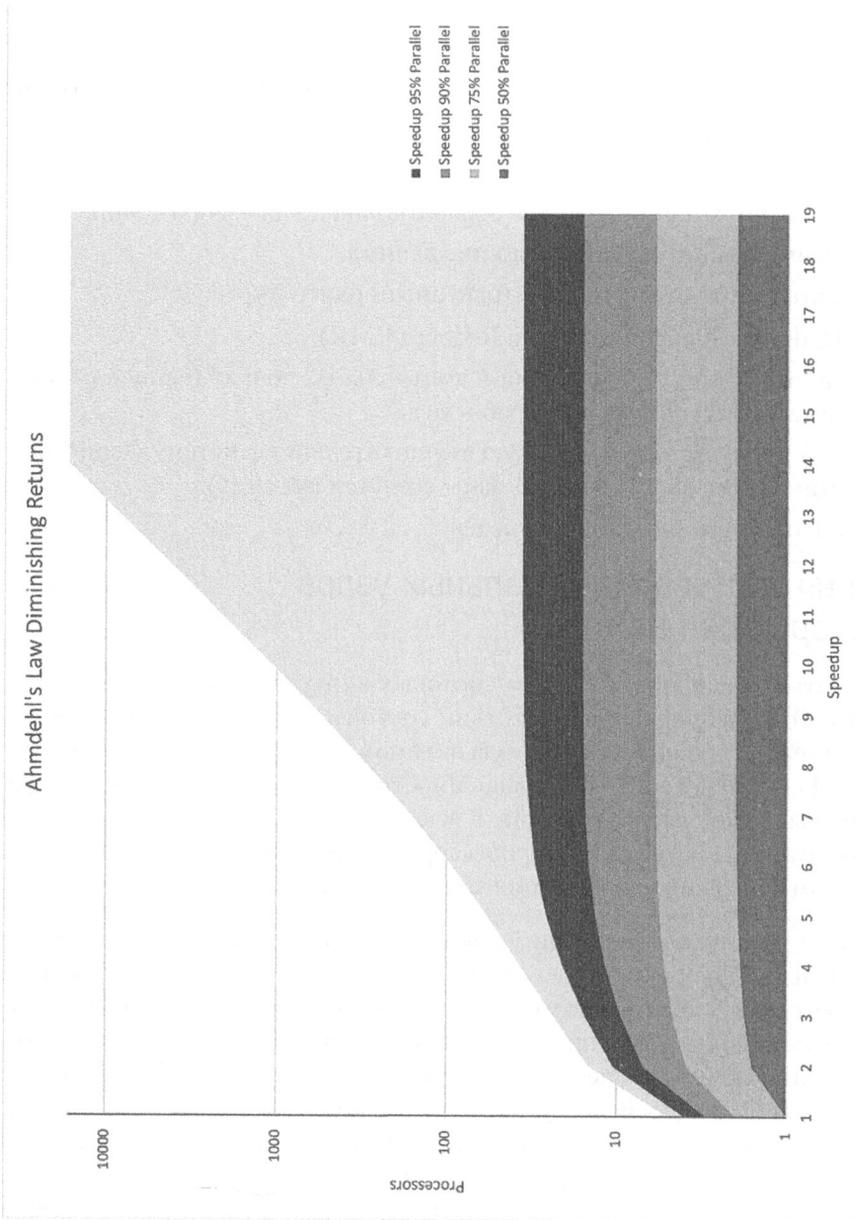


Рис. 9.1. Закон Амдала

Формула 9.1. Уравнение закона Амдала

$$S_{\text{время ожидания}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

где $S_{\text{время ожидания}}(s)$ — суммарный рост производительности; S — рост производительности распараллеленной части; p — доля времени выполнения, требовавшаяся изначально части программы, ускоряемой за счет оптимизации ресурсов.

Если последовательный компонент задания что-то разархивирует, то более быстрые процессор и операции дискового ввода/вывода окажутся полезными. В целом, если не говорить о теории, распределенные задания — вещь непростая, выбор подходящих типов спотовых виртуальных узлов и архитектур требует экспериментов и соответствующего инструментария. Необходимо запустить задание, посмотреть на метрики отдельных узлов, оценить время выполнения, после чего попробовать различные архитектуры и настройки, например сравнить EFS с чередуемыми томами (elastic block storage, EBS), разделяемыми посредством сетевой файловой системы (network file system, NFS).

Я многие годы работал в киноиндустрии, и существует мнение, что это первая отрасль промышленности, имевшая отношение к большим данным. В киноиндустрии работали с фреймворками распределенных заданий задолго до начала обсуждения в их контексте Hadoop, больших данных, машинного обучения и ИИ. Помимо прочего, при работе с большими данными в киноиндустрии я осознал тот факт, что в распределенных системах всегда что-нибудь идет не так, как планировалось. Основной совет: соблюдать строжайшую дисциплину в плане предельного упрощения заданий и работать с наилучшими возможными инструментами.

Возвращаясь к вопросу цен на спотовые виртуальные узлы, следует понимать, что существует множество способов оптимизации производительности распределенных заданий. Конечно, проще всего будет найти дешевые мощные виртуальные узлы, но в долгосрочной перспективе для успешной разработки заданий промышленного уровня критически важно принимать во внимание и другие конфигурации, а также методы их тестирования.

Прекрасный ресурс для сравнения цен на спотовые узлы и оценки возможностей машин — <http://www.ec2instances.info/>. На GitHub также доступен

исходный код этого проекта (<https://github.com/powdahound/ec2instances.info>). Авторы провели скрапинг сайта AWS и показали на удобной веб-странице цены на спотовые виртуальные узлы в сравнении с ценами на зарезервированные. Данные были отформатированы и помещены в CSV-файл для удобства импортирования их в блокнот Jupiter.

Создание утилиты и блокнота для сравнения цен на спотовые виртуальные узлы на основе машинного обучения

Основной предмет данной книги — создание программных решений на основе машинного обучения, пригодных для промышленной эксплуатации. Философия операционной системы Unix включает в себя идею маленьких утилит, которые служат лишь для одного действия, но выполняют это действие хорошо. Промышленные системы часто требуют, помимо основной системы, разработки и множества небольших утилит, благодаря которым подобная система способна функционировать. В духе данной философии мы создадим утилиту, которая ищет цены на спотовые виртуальные узлы в регионе AWS и применяет методы машинного обучения для рекомендации вариантов в этом же кластере. Для начала создадим новый блокнот Jupiter и вставим туда уже привычный шаблонный код:

```
In [1]: import pandas as pd
...: import seaborn as sns
...: import matplotlib.pyplot as plt
...: from sklearn.cluster import KMeans
...: %matplotlib inline
...: from IPython.core.display import display, HTML
...: display(HTML("<style>.container \
...:     { width:100% !important; }</style>"))
...: import boto3
```

Далее мы загрузим исходный CSV-файл с информацией, полученной с сайта <http://www.ec2instances.info/>, немного отформатированный с помощью Excel:

```
In [2]: pricing_df = pd.read_csv("../data/ec2-prices.csv")
...: pricing_df['price_per_ecu_on_demand'] =\
...: pricing_df['linux_on_demand_cost_hourly']/\
...: pricing_df['compute_units_ecu']
```

```

...: pricing_df.head()
...:
Out[2]:

```

	Name	InstanceType	memory_gb	compute_units_ecu	\
R3	High-Memory	Large	r3.large	15.25	6.5
M4		Large	m4.large	8.00	6.5
R4	High-Memory	Large	r4.large	15.25	7.0
C4	High-CPU	Large	c4.large	3.75	8.0
GPU	Extra	Large	p2.xlarge	61.00	12.0

```

vcpu  gpus  fpga  enhanced_networking  linux_on_demand_cost_hourly \
  2    0    0                Yes                        0.17
  2    0    0                Yes                        0.10
  2    0    0                Yes                        0.13
  2    0    0                Yes                        0.10
  4    1    0                Yes                        0.90

price_per_ecu_on_demand
0          0.026154
1          0.015385
2          0.018571
3          0.012500
4          0.075000

```

Передаем имена виртуальных узлов из этого набора данных в API Boto для получения истории цен на спотовые виртуальные узлы:

```

In [3]: names = pricing_df["InstanceType"].to_dict()
In [6]: client = boto3.client('ec2')
...: response = client.describe_spot_price_history(\
...:     InstanceTypes = list(names.values()),
...:     ProductDescriptions = ["Linux/UNIX"])
In [7]: spot_price_history = response['SpotPriceHistory']
...: spot_history_df = pd.DataFrame(spot_price_history)
...: spot_history_df.SpotPrice = \
...:     spot_history_df.SpotPrice.astype(float)
...:

```

Самая полезная для нас информация, возвращаемая API, — значение `SpotPrice` (цена спотового виртуального узла), которое можно использовать как для рекомендации схожих виртуальных узлов, так и для выяснения наилучшей цены в пересчете на адаптивный вычислительный блок (elastic compute unit, ECU) и блок памяти. Кроме того, результаты, возвращаемые в формате JSON, импортируются далее в объект `DataFrame` библиотеки

Pandas. Столбец SpotPrice затем преобразуется в значение с плавающей точкой для дальнейших математических операций над ним:

```
In [8]: spot_history_df.head()
Out[8]:
  AvailabilityZone InstanceType ProductDescription SpotPrice \
0      us-west-2c   r4.8xlarge      Linux/UNIX      0.9000
1      us-west-2c   p2.xlarge      Linux/UNIX      0.2763
2      us-west-2c   m3.2xlarge      Linux/UNIX      0.0948
3      us-west-2c   c4.xlarge      Linux/UNIX      0.0573
4      us-west-2a   m3.xlarge      Linux/UNIX      0.0447

      Timestamp
0 2017-09-11 15:22:59+00:00
1 2017-09-11 15:22:39+00:00
2 2017-09-11 15:22:39+00:00
3 2017-09-11 15:22:38+00:00
4 2017-09-11 15:22:38+00:00
```

Далее сливаются два объекта DataFrame и создаются новые столбцы, соответствующие ценам на спотовые виртуальные узлы в пересчете на вычислительные блоки (ECU) и блоки памяти. Операция describe библиотеки Pandas, примененная к трем столбцам, возвращает статистические показатели только что созданного объекта DataFrame:

```
In [16]: df = spot_history_df.merge(\
        pricing_df, how="inner", on="InstanceType")
...: df['price_memory_spot'] = \
        df['SpotPrice']/df['memory_gb']
...: df['price_ecu_spot'] = \
        df['SpotPrice']/df['compute_units_ecu']
...: df[["price_ecu_spot", "SpotPrice", \
        "price_memory_spot"]].describe()
...:
Out[16]:
   price_ecu_spot  SpotPrice  price_memory_spot
count      1000.000000    1000.000000      1000.000000
mean         0.007443         0.693629         0.005041
std          0.029698         6.369657         0.006676
min          0.002259         0.009300         0.000683
25%         0.003471         0.097900         0.002690
50%         0.004250         0.243800         0.003230
75%         0.006440         0.556300         0.006264
max          0.765957        133.380000         0.147541
```

Для большей ясности визуализируем полученные данные. Сгруппируем их по типу виртуального узла AWS (InstanceType), что позволит вычислить медиану для каждого типа узла:

```
In [17]: df_median = df.groupby("InstanceType").median()
...: df_median["InstanceType"] = df_median.index
...: df_median["price_ecu_spot"] = \
df_median.price_ecu_spot.round(3)
...: df_median["divide_SpotPrice"] = df_median.SpotPrice/100
...: df_median.sort_values("price_ecu_spot", inplace=True)
```

Нарисуем столбчатый график Seaborn, включающий оба графика, друг поверх друга. Это прекрасный способ демонстрации соотношения двух связанных столбцов. Отношение цены на спотовый виртуальный узел к числу вычислительных блоков, price_ecu_spot, сравнивается с исходной ценой на спотовый узел. График приведен на рис. 9.2, где отсортированный объект DataFrame дает возможность увидеть четкий паттерн; тут есть значения, весьма заманчивые для бережливых любителей распределенных вычислений. В данном конкретном регионе, us-west-2, оптимальными являются виртуальные узлы типа r4.large с точки зрения как цены на спотовый виртуальный узел, так и отношения этой цены к числу ECU. Для повышения наглядности разделим цену на спотовый виртуальный узел на 100.

```
...: plt.subplots(figsize=(20,15))
...: ax = plt.axes()
...: sns.set_color_codes("muted")
...: sns.barplot(x="price_ecu_spot", \
...:            y="InstanceType", data=df_median,
...:            label="Spot Price Per ECU", color="b")
...: sns.set_color_codes("pastel")
...: sns.barplot(x="divide_SpotPrice", \
...:            y="InstanceType", data=df_median,
...:            label="Spot Price/100", color="b")
...:
...: # Добавляем легенду и информационную метку оси
...: ax.legend(ncol=2, loc="lower right", frameon=True)
...: ax.set(xlim=(0, .1), ylabel="",
...:        xlabel="AWS Spot Pricing by Compute Units (ECU)")
...: sns.despine(left=True, bottom=True)
...:
<matplotlib.figure.Figure at 0x11383ef98>
```

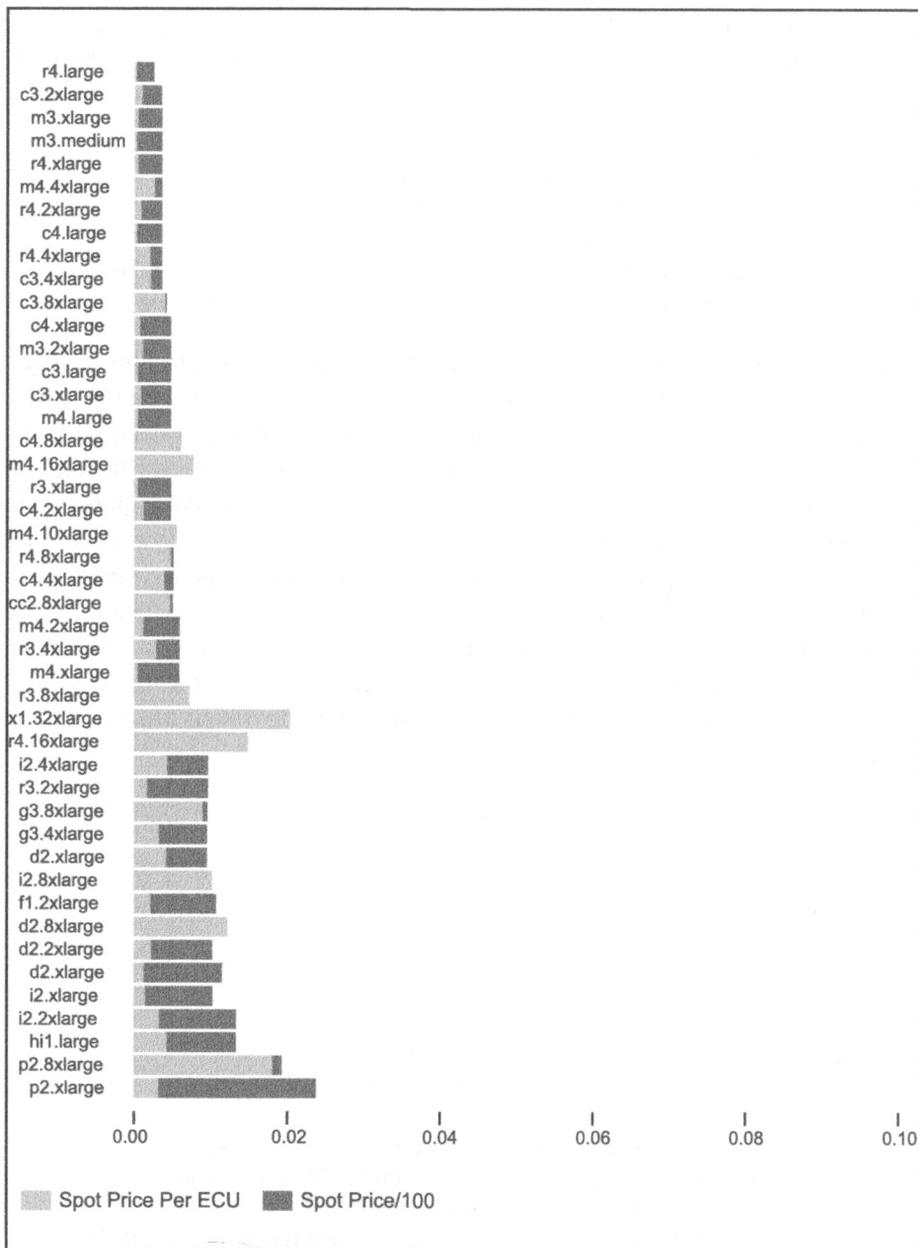


Рис. 9.2. Расценки спотовых виртуальных узлов AWS в пересчете на вычислительные блоки

Здесь достаточно информации для того, чтобы сделать из этого кода утилиту командной строки, удобную при выборе типа спотового виртуального узла. Для создания новой утилиты командной строки мы создадим в каталоге `raws` новый модуль и обернем вышеприведенный код в функции:

```
def cluster(combined_spot_history_df, sort_by="price_ecu_spot"):
    """Кластеризация спотовых виртуальных узлов"""

    df_median = combined_spot_history_df.\
        groupby("InstanceType").median()
    df_median["InstanceType"] = df_median.index
    df_median["price_ecu_spot"] = df_median.price_ecu_spot.round(3)
    df_median.sort_values(sort_by, inplace=True)
    numerical_df = df_median.loc[:,\
        ["price_ecu_spot", "price_memory_spot", "SpotPrice"]]
    scaler = MinMaxScaler()
    scaler.fit(numerical_df)
    scaler.transform(numerical_df)
    k_means = KMeans(n_clusters=3)
    kmeans = k_means.fit(scaler.transform(numerical_df))
    df_median["cluster"]=kmeans.labels_
    return df_median

def recommend_cluster(df_cluster, instance_type):
    """Принимает в качестве параметра (instance_type) тип узла
    и находит рекомендации относительно других похожих узлов"""

    vals = df_cluster.loc[df_cluster['InstanceType'] ==\
        instance_type]
    cluster_res = vals['cluster'].to_dict()
    cluster_num = cluster_res[instance_type]
    cluster_members = df_cluster.loc[df_cluster["cluster"] ==\
        cluster_num]
    return cluster_members
```

Функция `cluster` нормирует данные и на основе значений `price_ecu_spot`, `price_memory_spot` и `SpotPrice` создает три кластера. Функция `recommend_cluster` работает в предположении, что относящиеся к одному кластеру виртуальные узлы с большой вероятностью взаимозаменяемы. Достаточно мельком взглянуть на данные, в `Jupyter`, чтобы увидеть там три отдельных кластера. Кластер 1 отличается практически мизерным объемом памяти и соответствующе высокой ценой на узлы, в него входит только один узел. Кластер 2 насчитывает 11 виртуальных узлов, объемы памяти в нем самые низкие. В кластере 0 больше всего узлов (33) и цены лишь ненамного выше,

зато в среднем вдвое больше памяти. На основе этого можно создать удобную утилиту командной строки для выбора пользователем типа спотового виртуального узла — с малым, средним или большим объемом оперативной памяти — и отображения размера платы за него:

```
In [25]: df_median[["price_ecu_spot", "SpotPrice", \
    "price_memory_spot", "memory_gb", "cluster"]].\
    groupby("cluster").median()
Out[25]:
```

cluster	price_ecu_spot	SpotPrice	price_memory_spot	memory_gb
0	0.005	0.2430	0.002817	61.0
1	0.766	72.0000	0.147541	488.0
2	0.004	0.1741	0.007147	30.0

```
In [27]: df_median[["price_ecu_spot", "SpotPrice", \
    "price_memory_spot", "memory_gb", "cluster"]].\
    groupby("cluster").count()
Out[27]:
```

cluster	price_ecu_spot	SpotPrice	price_memory_spot	memory_gb
0	33	33	33	33
1	1	1	1	1
2	11	11	11	11

Последний этап создания этой утилиты командной строки основан на уже показанной выше в данной главе последовательности действий: импорт библиотеки, управление опциями с помощью фреймворка Click и возврат результатов с помощью функции `click.echo`. У команды `recommend` есть флаг `--instance`, она возвращает результаты для всех членов соответствующего кластера:

```
@cli.command("recommend")
@click.option('--instance', help='Instance Type')
def recommend(instance):
    """Рекомендует схожие спотовые виртуальные узлы на основе
    кластеризации методом k-ближайших соседей
```

Пример использования:

```
./spot-price-ml.py recommend --instance c3.8xlarge
"""
pd.set_option('display.float_format', lambda x: '%.3f' % x)
pricing_df = setup_spot_data("data/ec2-prices.csv")
```

```

names = pricing_df["InstanceType"].to_dict()
spot_history_df = get_spot_pricing_history(names,
    product_description="Linux/UNIX")
df = combined_spot_df(spot_history_df, pricing_df)
df_cluster = cluster(df, sort_by="price_ecu_spot")
df_cluster_members = recommend_cluster(df_cluster, instance)
click.echo(df_cluster_members[["SpotPrice", \
    "price_ecu_spot", "cluster", "price_memory_spot"]])

```

Результаты применения утилиты на практике выглядят следующим образом:

```

→ X ./spot-price-ml.py recommend --instance c3.8xlarge
      SpotPrice  price_ecu_spot  cluster  price_memory_spot
InstanceType
c3.2xlarge      0.098           0.003      0           0.007
c3.4xlarge      0.176           0.003      0           0.006
c3.8xlarge      0.370           0.003      0           0.006
c4.4xlarge      0.265           0.004      0           0.009
cc2.8xlarge     0.356           0.004      0           0.006
c3.large        0.027           0.004      0           0.007
c3.xlarge       0.053           0.004      0           0.007
c4.2xlarge      0.125           0.004      0           0.008
c4.8xlarge      0.557           0.004      0           0.009
c4.xlarge       0.060           0.004      0           0.008
hi1.4xlarge     0.370           0.011      0           0.006

```

Написание модуля запуска спотового виртуального узла

Работать со спотовыми виртуальными узлами можно на множестве уровней. В этом разделе мы обсудим некоторые из них, начиная с простейшего примера. Спотовые виртуальные узлы — главный двигатель машинного обучения на AWS. Понимание, как их правильно использовать, может стать основополагающим или роковым для компании, проекта или хобби. Рекомендуется создавать самоуничтожающиеся спотовые виртуальные узлы, автоматически завершающие свою работу по истечении примерно часа. Это аналог программы Hello, world! в области запуска спотовых виртуальных узлов — по крайней мере рекомендуемый аналог.

В первом разделе мы импортируем библиотеку click, а также boto и библиотеку Base64. Отправляемые в AWS пользовательские данные должны

быть в кодировке Base64, что будет продемонстрировано в приведенном ниже фрагменте кода. Обратите внимание, что если раскомментировать строку кода с `boto.set_stream_logger`, то в журнал запишутся чрезвычайно подробные сообщения (что может оказаться весьма полезным при экспериментах с параметрами).

```
#!/usr/bin/env python
"""Запускаем тестовый спотовый виртуальный узел"""

import click
import boto3
import base64

from sensible.loginit import logger
log = logger(__name__)

#Указываем Boto3 включить журналирование отладочной информации
#boto3.set_stream_logger(name='botocore')
```

В следующем разделе мы создадим утилиту командной строки и зададим пользовательское значение для времени автоматического завершения работы. Это отличный прием, придуманный Эриком Хэммондом (Eric Hammond, <https://www.linkedin.com/in/ehammond/>). По сути, сразу же при запуске машины создается задание с указанием команды `at`, завершающей работу виртуального узла через определенный промежуток времени. В данной утилите командной строки этот прием развивается, и пользователь может задать длительность функционирования, если его не устраивает значение по умолчанию 55 мин.

```
@click.group()
def cli():
    """Средство запуска спотового виртуального узла"""

def user_data_cmds(duration):
    """Изначально выполняемые команды, получает в качестве параметра
    длительность для команды halt"""

    cmds = """
        #cloud-config
        runcmd:
          - echo "halt" | at now + {duration} min
    """.format(duration=duration)
    return cmds
```

В приведенных ниже опциях все значения установлены по умолчанию, пользователю достаточно указать команду запуска. Далее эти опции передаются клиенту boto3 для вызова API запроса спотового виртуального узла:

```
@cli.command("launch")
@click.option('--instance', default="r4.large", help='Instance Type')
@click.option('--duration', default="55", help='Duration')
@click.option('--keyname', default="pragai", help='Key Name')
@click.option('--profile', \
              default="arn:aws:iam::561744971673:instance-profile/admin", \
              help='IamInstanceProfile')
@click.option('--securitygroup', \
              default="sg-61706e07", help='Key Name')
@click.option('--ami', default="ami-6df1e514", help='Key Name')
def request_spot_instance(duration, instance, keyname,
                          profile, securitygroup, ami):
    """Запрашивает спотовый виртуальный узел"""

    user_data = user_data_cmds(duration)
    LaunchSpecifications = {
        "ImageId": ami,
        "InstanceType": instance,
        "KeyName": keyname,
        "IamInstanceProfile": {
            "Arn": profile
        },
        "UserData": base64.b64encode(user_data.encode("ascii")).\
            decode('ascii'),
        "BlockDeviceMappings": [
            {
                "DeviceName": "/dev/xvda",
                "Ebs": {
                    "DeleteOnTermination": True,
                    "VolumeType": "gp2",
                    "VolumeSize": 8,
                }
            }
        ],
        "SecurityGroupIds": [securitygroup]
    }

    run_args = {
        'SpotPrice' : "0.8",
```

```

        'Type'           : "one-time",
        'InstanceCount' : 1,
        'LaunchSpecification' : LaunchSpecifications
    }

    msg_user_data = "SPOT REQUEST DATA: %s" % run_args
    log.info(msg_user_data)

    client = boto3.client('ec2', "us-west-2")
    reservation = client.request_spot_instances(**run_args)
    return reservation

if __name__ == '__main__':
    cli()

```

При вызове утилиты командной строки с указанием параметра `--help` выводится список опций, которые допускают изменение. Обратите внимание, что опций для цены среди них нет, и по нескольким причинам. Во-первых, цена на спотовые виртуальные узлы подчиняется законам рынка, так что в запросе всегда используется минимальная цена. Во-вторых, ее легко можно добавить в утилиту вышеописанным способом.

```

→ ./spot_launcher.py launch --help
Usage: spot_launcher.py launch [OPTIONS]

```

Запрашивает спотовый виртуальный узел

Options:

```

--instance TEXT      Instance Type
--duration TEXT      Duration
--keyname TEXT       Key Name
--profile TEXT       IamInstanceProfile
--securitygroup TEXT Key Name
--ami TEXT           Key Name
--help              Show this message and exit.

```

Запустить спотовый виртуальный узел с новым значением длительности функционирования, допустим 1 ч 55 мин, можно с помощью опции `--duration`:

```

→ X ./spot_launcher.py launch --duration 115
2017-09-20 06:46:53,046 - __main__ - INFO -
SPOT REQUEST DATA: {'SpotPrice': '0.8', 'Type':

```

```
'one-time', 'InstanceCount': 1, 'LaunchSpecification':
  {'ImageId': 'ami-6df1e514', 'InstanceType':
'r4.large', 'KeyName': 'pragai', 'IamInstanceProfile':
  {'Arn': 'arn:aws:iam::561744971673:instance-profile/admin'}},
.....
```

Найти виртуальный узел затем можно с помощью панели инструментов EC2 для региона, в котором он был запущен. Для данного запроса адрес в консоли AWS будет следующим: <https://us-west-2.console.aws.amazon.com/ec2/v2/home?region=us-west-2#Instances:sort=ipv6Ips>, как показано на рис. 9.3, где приведена информация о подключении к машине по протоколу ssh.

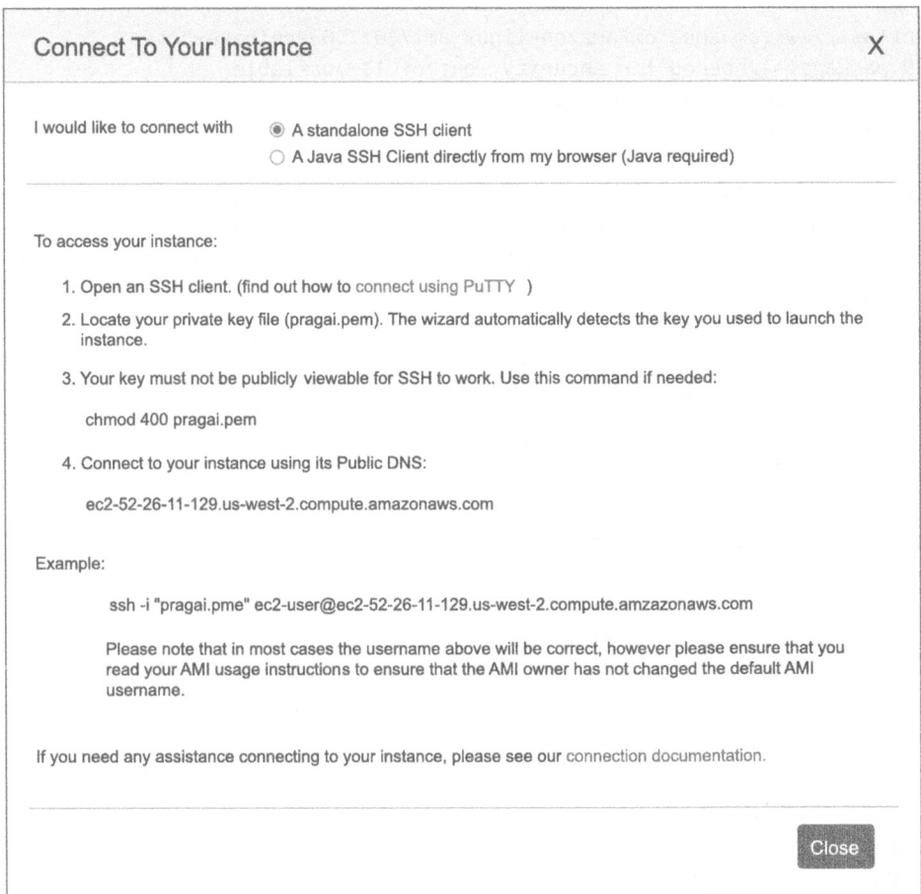


Рис. 9.3. Подключение к спотовому виртуальному узлу AWS

С помощью этой информации можно создать ssh-соединение со спотовым виртуальным узлом:

```
→ X ssh -i "~/.ssh/pragai.pem" ec2-user@52.26.11.129
The authenticity of host '52.26.11.129 (52.26.11.129)'
ECDSA key fingerprint is SHA256:1TaVeVv0L7GE...
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '52.26.11.129'
```

```
  _ |  _ | _ )
  _ | (      /  Amazon Linux AMI
  _ | \ _ | _ |
```

```
https://aws.amazon.com/amazon-linux-ami/2017.03-release-notes/
9 package(s) needed for security, out of 13 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-8-237 ~]$
```

В командной оболочке Amazon Linux с помощью команды `uptime` можно узнать, сколько времени работает виртуальный узел. Данный узел работает уже 1 ч 31 мин и проработает до останова еще немногим более 20 мин:

```
[ec2-user@ip-172-31-8-237 ~]$ uptime
15:18:52 up 1:31, 1 user, load average: 0.00, 0.00, 0.00
```

Если перейти в режим суперпользователя `root`, можно проверить задание останова машины:

```
[ec2-user@ip-172-31-8-237 ~]$ sudo su -
[root@ip-172-31-8-237 ~]# at -l
1 2017-09-20 15:42 a root
```

Далее можно проверить, какая команда будет фактически запущена в этот момент в будущем:

```
#!/bin/sh
# atrun uid=0 gid=0
# mail root 0
umask 22
PATH=/sbin:/usr/sbin:/bin:/usr/bin; export PATH
RUNLEVEL=3; export RUNLEVEL
runlevel=3; export runlevel
```

```

PWD=/; export PWD
LANGSH_SOURCED=1; export LANGSH_SOURCED
LANG=en_US.UTF-8; export LANG
PREVLEVEL=N; export PREVLEVEL
previous=N; export previous
CONSOLETYPE=serial; export CONSOLETYPE
SHLVL=4; export SHLVL
UPSTART_INSTANCE=; export UPSTART_INSTANCE
UPSTART_EVENTS=runlevel; export UPSTART_EVENTS
UPSTART_JOB=rc; export UPSTART_JOB
cd / || {
    echo 'Execution directory inaccessible' >&2
    exit 1
}
${SHELL:-/bin/sh} << 'marcinDELIMITER6382915b'
halt

```

Благодаря внесенным компанией Amazon в свои предложения изменениям подобный подход к исследованиям и разработке становится еще привлекательнее. По состоянию на 3 октября 2017 г. Amazon предлагает посекундную тарификацию с минимумом 1 мин. Это полностью меняет идеологию использования спотовых виртуальных узлов. Одно из наиболее очевидных изменений: теперь имеет смысл использовать спотовые виртуальные узлы просто для выполнения специализированных функций, а по их завершении, скажем, через 30 с останавливать узел.

Во многих реальных проектах сверх подобного простого средства запуска спотовых виртуальных узлов можно перейти к следующему этапу — развертыванию промышленной версии программного обеспечения на запускаемом узле. Сделать это можно несколькими способами.

- ❑ Передать узлу сценарий командной оболочки при запуске, как я только что продемонстрировал. Более сложные примеры вы можете найти в официальной документации AWS (<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html>).
- ❑ Изменить сам образ Linux AMI и воспользоваться им при запуске узла. Сделать это можно двумя способами. Во-первых, просто запустить узел, настроить его, а затем сохранить снимок состояния. Другой метод: применить сборщик образов AMI (AMI Builder Packer, <https://www.packer.io/docs/builders/amazon-eks.html>). При запуске узла на машине уже оказывается

необходимое программное обеспечение. Этот подход можно сочетать с другими, скажем с заранее собранным образом AMI, а также с пользовательским сценарием командной оболочки.

- ❑ Воспользоваться EFS при загрузке, для хранения как данных, так и ПО, а также для подключения к среде библиотек и сценариев. Данный подход очень часто применялся во времена использования в Solaris и других Unix-подобных операционных системах файловой системы NFS. Это замечательный способ настройки среды спотовых виртуальных узлов под свои нужды. Том EFS можно обновить с помощью сервера сборки, посредством копирования или удаленной синхронизации.
- ❑ Неплохим вариантом будет воспользоваться сервисом пакетной обработки AWS с применением контейнеров Docker.

Написание более сложного модуля запуска для спотового виртуального узла

Более сложный модуль запуска для спотового узла может установить в систему какое-либо программное обеспечение, извлекать исходный код из репозитория, выполнять этот исходный код и помещать результаты его выполнения, допустим, в хранилище S3. Для этого нам придется кое-что изменить. Во-первых, необходимо модифицировать файл `buildspec.yml` так, чтобы копировать исходный код в S3. Обратите внимание на удобство команды синхронизации с ключом `--delete` для интеллектуальной синхронизации только тех файлов, которые были изменены, и удаления более не существующих.

```
post_build:
  commands:
    - echo "COPY Code TO S3"
    - rm -rf ~/.aws
    - aws s3 sync $CODEBUILD_SRC_DIR \
s3://pragai-aws/master --delete
```

Обычно при использовании подобных команд сборки я сначала запускаю их на локальной машине, чтобы убедиться, что правильно понимаю суть их работы. Далее необходимо установить Python и виртуальную среду при запуске машины. Для этого можно модифицировать передаваемые вирту-

альному узлу при загрузке команды (`runcmd`). В первом модифицированном нами разделе устанавливается Python вместе с необходимыми для него пакетами. Обратите внимание, что мы воспользовались пакетом `ensurepip`, чтобы упростить выполнение сборочного файла:

```
cmds = """
#cloud-config
runcmd:
- echo "halt" | at now + {duration} min
- wget https://www.python.org/ftp/\
python/3.6.2/Python-3.6.2.tgz
- tar zxvf Python-3.6.2.tgz
- yum install -y gcc readline-devel\
sqlite-devel zlib-devel openssl-devel
- cd Python-3.6.2
- ./configure --with-ensurepip=install && make install
```

Далее производится локальное извлечение синхронизированного с S3 исходного кода — удобный способ развертывания кода на виртуальном узле. Эта операция производится очень быстро с помощью `ssh`-ключей `Git`, без необходимости использования паролей, благодаря привилегиям роли спотового виртуального узла по взаимодействию с S3. После копирования данных из S3 на локальную машину, выполнения команды `source` с использованием виртуальной среды запускается вышеприведенная утилита рекомендации узлов по ценам на основе машинного обучения, и результаты ее работы отправляются в S3:

```
- cd ..
- aws s3 cp s3://pragai-aws/master master\
--recursive && cd master
- make setup
- source ~/.pragia-aws/bin/activate && make install
- ~/.pragia-aws/bin/python spot-price-ml.py\
describe > prices.txt
- aws s3 cp prices.txt s3://spot-jobs-output
""".format(duration=duration)
return cmds
```

Естественно, что следующий шаг — взять наш прототип и разбить его на модули, чтобы можно было выполнять произвольную операцию машинного обучения в виде сценария, а не только как жестко «зашитый» пример.

Приведенный на рис. 9.4 высокоуровневый обзор этого конвейера демонстрирует работу данного процесса на практике.

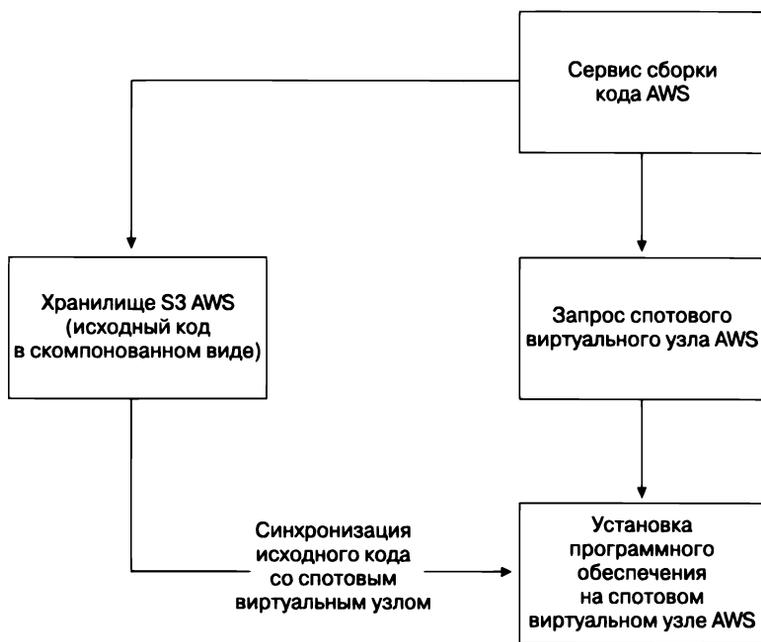


Рис. 9.4. Жизненный цикл временных заданий спотовых узлов AWS

Резюме

В этой главе мы рассмотрели один из часто упускаемых из виду компонентов машинного обучения — сам запуск заданий на AWS. Мы разобрались с несколькими серьезными проблемами, связанными со спотовыми виртуальными узлами: выбором нужного размера узла и наиболее экономически выгодного способа их использования, установкой на них программного обеспечения и развертыванием кода.

Недавние изменения в AWS, связанные с посекундной тарификацией и добавлением таких сервисов, как сервис пакетной обработки, сделали AWS достойным конкурентом в войне облачных сервисов. Никогда ранее на рынке не встречалось сочетание посекундной тарификации и спотового механизма образования цен на виртуальные узлы. Создание систем

машинного обучения для промышленной эксплуатации на основе инфраструктуры AWS — совершенно беспроблемный вариант, при котором легче, чем когда-либо, контролировать расходы на научные исследования и разработку.

Другие варианты дальнейшего повышения реалистичности этого ИИ-решения включают в себя запуск заданий в соответствии с определенным событием, например на основе пакетного задания AWS, которое будет отслеживать изменения цен и динамически определять оптимальный момент, и сочетания машин, с помощью линейной оптимизации и кластеризации. В качестве дальнейшего усовершенствования можно рассматривать сочетание этого ИИ-решения с такими технологиями, как Nomad от компании HashiCorp (<https://www.nomadproject.io/>) для динамического запуска заданий на всех облаках в виде образов Docker.

10

Недвижимость

На игровом поле уже не важно, любят вас или нет. Важно только играть на уровне и делать все возможное, что требуется для победы вашей команды. Вот и все.

*Леброн Джеймс
(LeBron James)*

Есть ли у вас какой-нибудь хороший набор данных для исследований? Такой вопрос мне, как лектору или инструктору на курсах, задавали едва ли не чаще всех других. В качестве «дежурного» ответа я нередко называю набор данных о недвижимости компании Zillow (<https://www.zillow.com/research/data/>). С рынком недвижимости в США сталкивается каждый, в связи с чем это прекрасная тема для разговора о машинном обучении.

Исследование цен на недвижимость в США

Жизнь в Области залива Сан-Франциско кого угодно заставит часто и подолгу размышлять о ценах на недвижимость. Для этого есть веские причины. Медиана цены на недвижимость в Области залива увеличивается с шокирующей быстротой. С 2010 по 2017 г. медианная цена на частный

дом в Сан-Франциско выросла с примерно \$775 000 до 1,5 млн. Эти данные можно исследовать с помощью блокнота Jupiter. Весь проект и соответствующие данные доступны на https://github.com/noahgiff/real_estate_ml.

В начале блокнота импортируется несколько библиотек, и библиотека Pandas конфигурируется для отображения чисел с плавающей точкой вместо экспоненциального представления:

```
In [1]: import pandas as pd
...: pd.set_option('display.float_format', lambda x: '%.3f' % x)
...: import numpy as np
...: import statsmodels.api as sm
...: import statsmodels.formula.api as smf
...: import matplotlib.pyplot as plt
...: import seaborn as sns
```

Double import seaborn?

```
...: import seaborn as sns; sns.set(color_codes=True)
...: from sklearn.cluster import KMeans
...: color = sns.color_palette()
...: from IPython.core.display import display, HTML
...: display(HTML("<style>.container \
    { width:100% !important; }</style>"))
...: %matplotlib inline
```

Затем импортируем данные компании Zillow для частных домов (<https://www.zillow.com/research/data/>) и получаем описательную статистику с помощью метода describe:

```
In [6]: df.head()
```

```
In [7]: df.describe()
```

```
Out[7]:
```

	RegionID	RegionName	SizeRank	1996-04	1996-05
count	15282.000	15282.000	15282.000	10843.000	10974.000
mean	80125.483	46295.286	7641.500	123036.189	122971.396
std	30816.445	28934.030	4411.678	78308.265	77822.431
min	58196.000	1001.000	1.000	24400.000	23900.000
25%	66785.250	21087.750	3821.250	75700.000	75900.000
50%	77175.000	44306.500	7641.500	104300.000	104450.000
75%	88700.500	70399.500	11461.750	147100.000	147200.000
max	738092.000	99901.000	15282.000	1769000.000	1768100.000

Далее выполняем очистку, переименовывая столбцы и форматируя данные:

```
In [8]: df.rename(columns={"RegionName": "ZipCode"}, inplace=True)
...: df["ZipCode"] = df["ZipCode"].map(lambda x: "{:.0f}".format(x))
...: df["RegionID"] = df["RegionID"].map(
...:     lambda x: "{:.0f}".format(x))
...: df.head()
...:
```

Out[8]:

RegionID	ZipCode	City	State	Metro	CountyName	SizeRank
84654	60657	Chicago	IL	Chicago	Cook	1.000
84616	60614	Chicago	IL	Chicago	Cook	2.000
93144	79936	El Paso	TX	El Paso	El Paso	3.000
84640	60640	Chicago	IL	Chicago	Cook	4.000
61807	10467	New York	NY	New York	Bronx	5.000

Для многих видов анализа в этом блокноте пригодятся медианные значения для США в целом. В следующем примере мы агрегируем значения для определенных мест или городов и вычисляем для них медианные значения. Создается новый объект DataFrame — `df_comparison`, которым мы далее воспользуемся для построения графиков с помощью библиотеки `Plotly`:

```
In [9]: median_prices = df.median()
```

```
In [10]: median_prices.tail()
```

Out[10]:

```
2017-05    180600.000
2017-06    181300.000
2017-07    182000.000
2017-08    182500.000
2017-09    183100.000
dtype: float64
```

```
In [11]: marin_df = df[df["CountyName"] == "Marin"].median()
...: sf_df = df[df["City"] == "San Francisco"].median()
...: palo_alto = df[df["City"] == "Palo Alto"].median()
...: df_comparison = pd.concat([marin_df, sf_df,
...:     palo_alto, median_prices], axis=1)
...: df_comparison.columns = ["Marin County",
...:     "San Francisco", "Palo Alto", "Median USA"]
...:
```

Интерактивная визуализация данных в Python

Существует две распространенные библиотеки для интерактивной визуализации данных в Python: Plotly (<https://github.com/plotly/plotly.py>) и Bokeh (<https://bokeh.pydata.org/en/latest/>). Мы воспользуемся для визуализации данных Plotly, но с помощью Bokeh также можно рисовать аналогичные графики. Plotly — коммерческая компания, чьи продукты можно использовать как в офлайн-режиме, так и с помощью экспорта на веб-сайт компании. У компании Plotly есть также фреймворк с открытым исходным кодом для языка Python — Dash (<https://plot.ly/products/dash/>), пригодный для создания аналитических веб-приложений. Большинство графиков этой главы вы можете найти на сайте <https://plot.ly/~ngift>.

В данном примере мы воспользуемся библиотекой Cufflinks (<https://plot.ly/ipython-notebooks/cufflinks/>), благодаря которой построение графиков объектов DataFrame библиотеки Pandas в Plotly становится тривиальной задачей. Библиотека Cufflinks рассматривается как «инструмент для повышения производительности» библиотеки Pandas. Одна из наиболее сильных сторон Cufflinks — построение соответствующих графиков практически нативным для Pandas образом.

```
In [12]: import cufflinks as cf
...: cf.go_offline()
...: df_comparison.iplot(title="Bay Area Median\
...:   Single Family Home Prices 1996-2017",
...:                       xTitle="Year",
...:                       yTitle="Sales Price",
...:                       #bestfit=True, bestfit_colors=["pink"],
...:                       #subplots=True,
...:                       shape=(4,1),
...:                       #subplot_titles=True,
...:                       fill=True,)
...:
```

На рис. 10.1 показан график с отключенными интерактивными возможностями. Покупать недвижимость в Пало-Альто, похоже, достаточно рискованно.

На рис. 10.2 курсор остановлен над декабрем 2009 г., демонстрируя точку близ дна последнего обвала рынка недвижимости, при котором медианная цена на дома в Пало-Альто была близка к \$1,2 млн, в Сан-Франциско — около \$750 тыс., а в целом по США — около \$170 тыс.

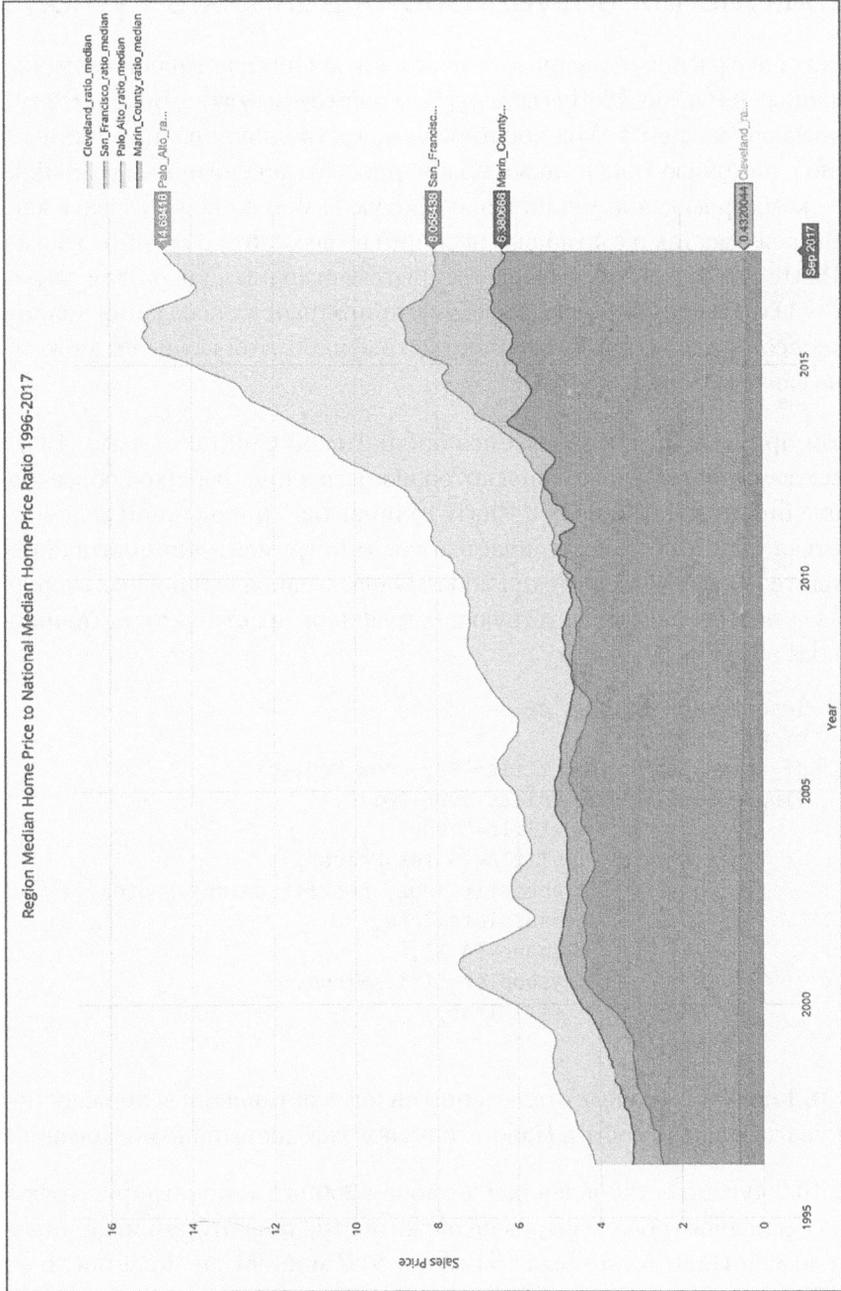


Рис. 10.1. Будут ли цены в Пало-Альто бесконечно расти в геометрической прогрессии?

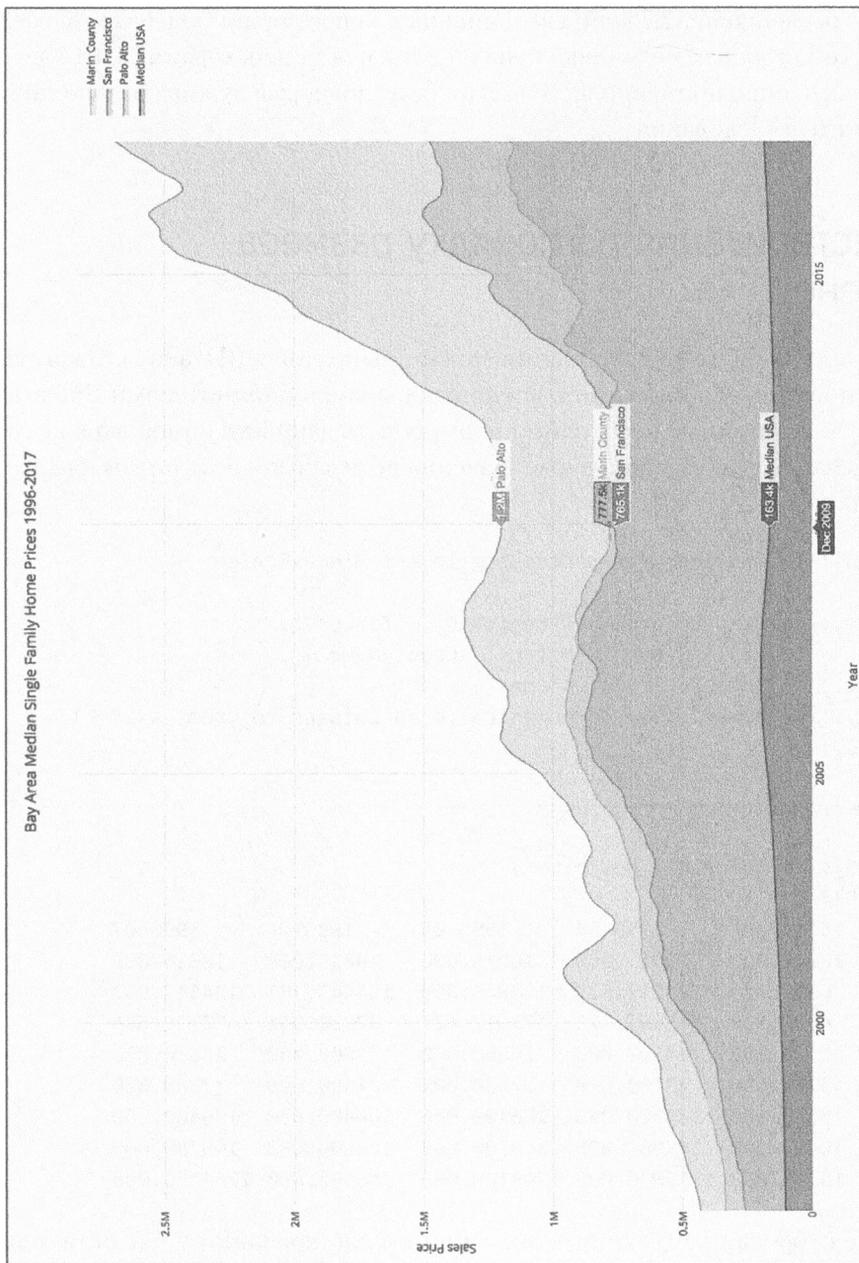


Рис. 10.2. Низшая точка рынка недвижимости в декабре 2009 г.

Пройдясь по графику, можно увидеть, что в декабре 2017 г. цена в Пало-Альто была около \$2,7 млн, увеличившись за восемь лет более чем вдвое. С другой стороны, медианная цена на дома в остальных частях США повысилась лишь примерно на 5 %. Это, безусловно, заслуживает дополнительного исследования.

Кластеризация по порядку размера и цене

Для дальнейшего исследования можно выполнить 3D-визуализацию на основе метода k-средних, воспользовавшись библиотеками Sklearn и Plotly. Во-первых, необходимо нормировать данные с помощью класса MinMaxScaler, чтобы аномальные значения не исказили результаты кластеризации:

```
In [13]: from sklearn.preprocessing import MinMaxScaler

In [14]: columns_to_drop = ['RegionID', 'ZipCode',
    ...:                    'City', 'State', 'Metro', 'CountyName']
    ...: df_numerical = df.dropna()
    ...: df_numerical = df_numerical.drop(columns_to_drop, axis=1)
    ...:
```

Далее сгенерируем статистику:

```
In [15]: df_numerical.describe()
Out[15]:
```

	SizeRank	1996-04	1996-05	1996-06	1996-07
count	10015.000	10015.000	10015.000	10015.000	10015.000
mean	6901.275	124233.839	124346.890	124445.791	124517.993
std	4300.338	78083.175	77917.627	77830.951	77776.606
min	1.000	24500.000	24500.000	24800.000	24800.000
25%	3166.500	77200.000	77300.000	77300.000	77300.000
50%	6578.000	105700.000	106100.000	106400.000	106400.000
75%	10462.000	148000.000	148200.000	148500.000	148700.000
max	15281.000	1769000.000	1768100.000	1766900.000	1764200.000

После отброса пропущенных значений и кластеризации у нас остается около 10 000 строк:

```
In [16]: scaler = MinMaxScaler()
...: scaled_df = scaler.fit_transform(df_numerical)
...: kmeans = KMeans(n_clusters=3, random_state=0).fit(scaled_df)
...: print(len(kmeans.labels_))
...:
```

10015

Добавляем столбец с коэффициентом роста цены и очищаем данные перед визуализацией:

```
cluster_df = df.copy(deep=True)
cluster_df.dropna(inplace=True)
cluster_df.describe()
cluster_df['cluster'] = kmeans.labels_

cluster_df['appreciation_ratio'] = \
    round(cluster_df["2017-09"]/cluster_df["1996-04"],2)
cluster_df['CityZipCodeAppRatio'] = \
    cluster_df['City'].map(str) + "-" + cluster_df['ZipCode'] + "-" +
    cluster_df["appreciation_ratio"].map(str)
cluster_df.head()
```

Далее мы воспользуемся Plotly в офлайн-режиме (то есть без отправки данных на серверы Plotly) и нарисуем график с тремя осями: X — коэффициент роста, Y — 1996 г. и Z — 2017 г. Кластеры закрашены в разные цвета. Некоторые паттерны на рис. 10.3 сразу заметны. Более всего за последние 20 лет выросли цены на недвижимость в Джерси-Сити — от \$142 тыс. до 1,344 млн, то есть более чем в девять раз.

Среди других выделяющихся значений — несколько почтовых индексов в Пало-Альто. Цены там также выросли более чем в шесть раз, что еще поразительнее, если учесть, насколько они были высокими изначально. За последние десять лет бум стартапов в Пало-Альто, включая Facebook, привел там к искаженному росту цен, по сравнению с равномерным во всей Области залива Сан-Франциско.

Интересно было бы также визуализировать коэффициент роста для этих же столбцов, чтобы узнать, сохраняется ли такая тенденция в Пало-Альто и дальше. Код будет аналогичен коду для построения графика на рис. 10.3.

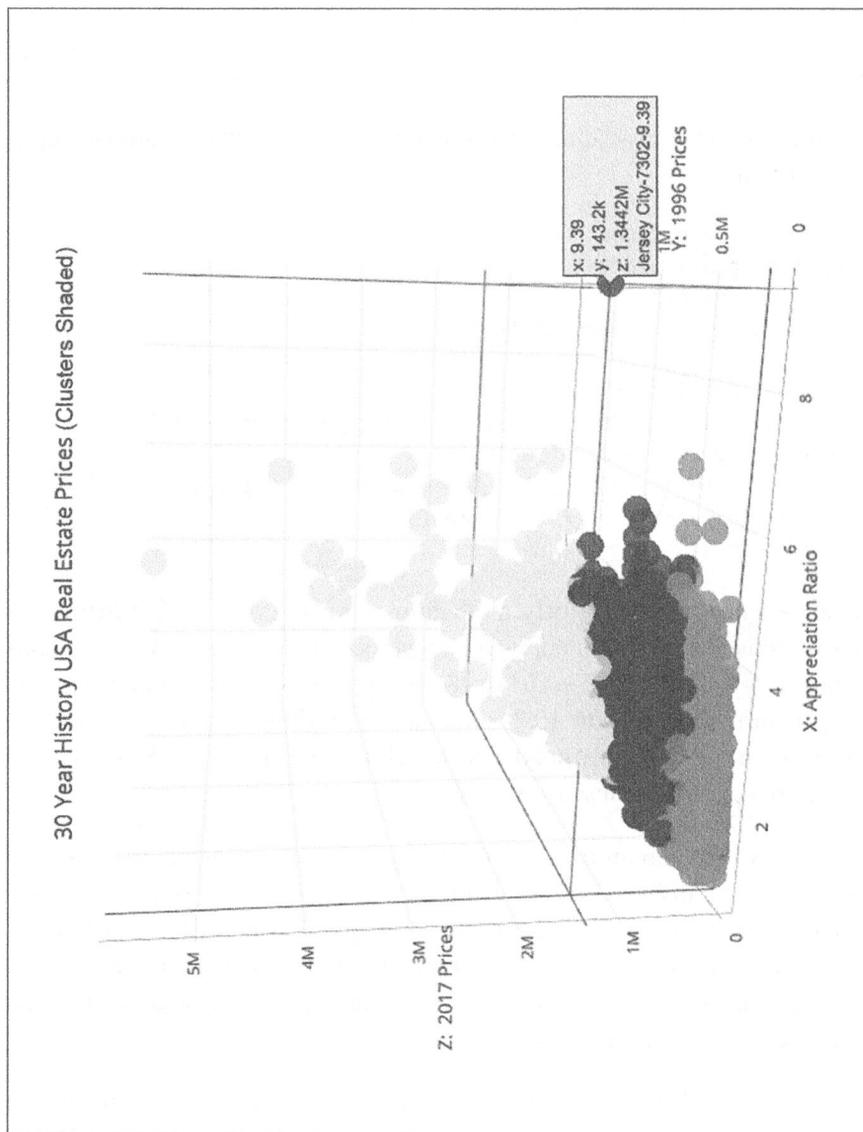


Рис. 10.3. Что за чертовщина творится с ценами в Джерси-Сити?

```

In [17]: from sklearn.neighbors import KNeighborsRegressor
...: neigh = KNeighborsRegressor(n_neighbors=2)
...:

In [19]: #df_comparison.columns = ["Marin County",
...: "San Francisco", "Palo Alto", "Median USA"]
...: cleveland = df[df["City"] == "Cleveland"].median()
...: df_median_compare = pd.DataFrame()
...: df_median_compare["Cleveland_ratio_median"] =\
...:     cleveland/df_comparison["Median USA"]
...: df_median_compare["San_Francisco_ratio_median"] =\
...:     df_comparison["San Francisco"]/df_comparison["Median USA"]
...: df_median_compare["Palo_Alto_ratio_median"] =\
...:     df_comparison["Palo Alto"]/df_comparison["Median USA"]
...: df_median_compare["Marin_County_ratio_median"] =\
...:     df_comparison["Marin County"]/df_comparison["Median USA"]
...:

In [20]: import cufflinks as cf
...: cf.go_offline()
...: df_median_compare.iplot(title="Region Median Home Price to
...: National Median Home Price Ratio 1996-2017",
...:                          xTitle="Year",
...:                          yTitle="Ratio to National Median",
...:                          #bestfit=True, bestfit_colors=["pink"],
...:                          #subplots=True,
...:                          shape=(4,1),
...:                          #subplot_titles=True,
...:                          fill=True,)
...:

```

На рис. 10.4 медиана роста цен для Пало-Альто напоминает геометрическую прогрессию из-за краха рынка недвижимости в 2008 г., а цены в остальной части Области залива Сан-Франциско, похоже, не столь волатильны. Можно обоснованно предположить, что цены на недвижимость в Пало-Альто чрезмерно раздуты, и долго так продолжаться не может. Рано или поздно их экспоненциальному росту придет конец.

Не помешает поискать дополнительные паттерны и в индексе аренды недвижимости.

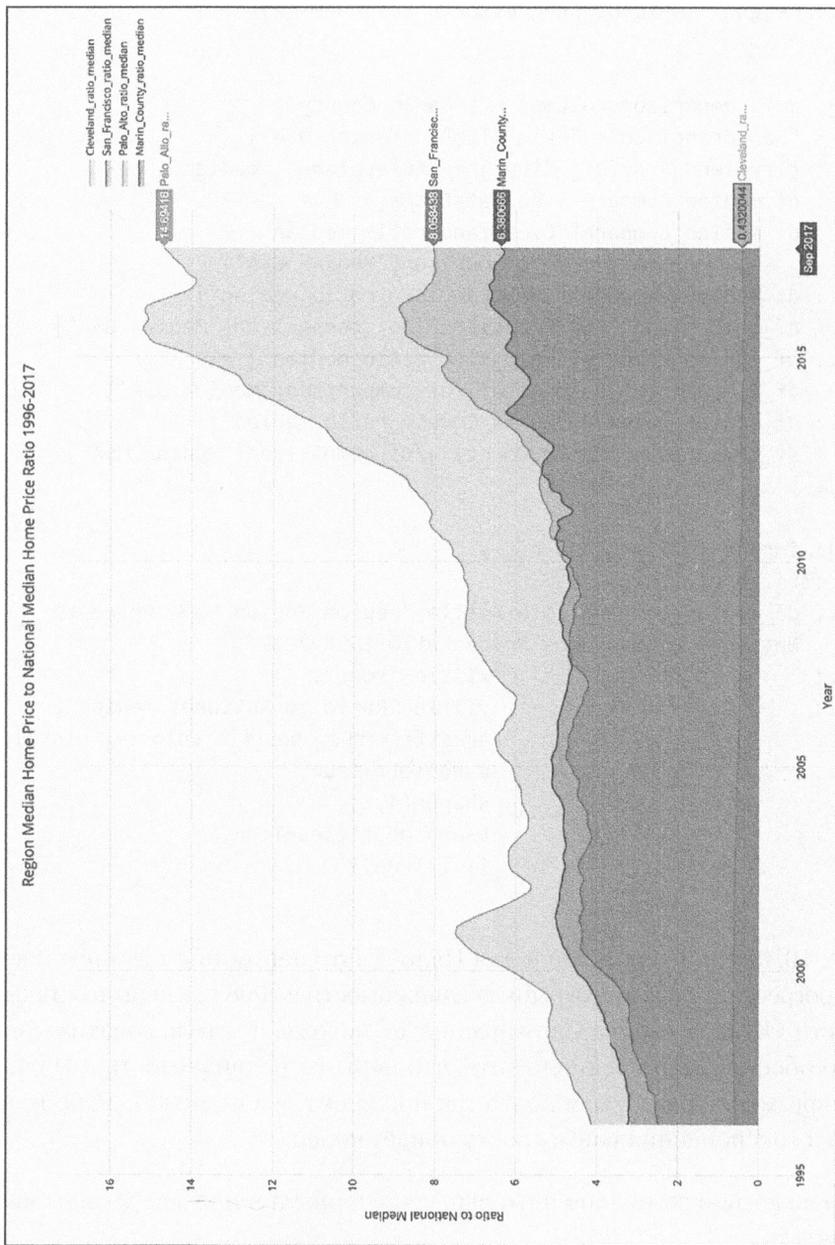


Рис. 10.4. Цены на дома в Пало-Альто выросли примерно за десять лет от пятикратно превышающих медиану по стране до пятнадцатикратно превышающих

Очистим импортированные исходные данные и переименуем столбец Metro в City:

```
In [21]: df_rent = pd.read_csv(
...:     "../data/City_MedianRentalPrice_Sfr.csv")
...: df_rent.head()
...: median_prices_rent = df_rent.median()
...: df_rent[df_rent["CountyName"] == "Marin"].median()
...: df_rent.columns
...:
Out[21]:
Index(['RegionName', 'State', 'Metro',
      'CountyName', 'SizeRank', '2010-01',
```

```
In [22]: df_rent.rename(columns={"Metro": "City"}, inplace=True)
...: df_rent.head()
...:
```

```
Out[22]:
```

	RegionName	State		City	CountyName
0	New York	NY		New York	Queens
1	Los Angeles	CA	Los Angeles-Long Beach-Anaheim		Los Angeles
2	Chicago	IL		Chicago	Cook
3	Houston	TX		Houston	Harris
4	Philadelphia	PA		Philadelphia	Philadelphia

Далее сформируем медианы в новом объекте DataFrame:

```
In [23]: median_prices_rent = df_rent.median()
...: marin_df = df_rent[df_rent["CountyName"] ==\
...:     "Marin"].median()
...: sf_df = df_rent[df_rent["City"] == "San Francisco"].median()
...: cleveland = df_rent[df_rent["City"] == "Cleveland"].median()
...: palo_alto = df_rent[df_rent["City"] == "Palo Alto"].median()
...: df_comparison_rent = pd.concat([marin_df,
...:     sf_df, palo_alto, cleveland, median_prices_rent], axis=1)
...: df_comparison_rent.columns = ["Marin County",
...:     "San Francisco", "Palo Alto", "Cleveland", "Median USA"]
...:
```

Наконец, воспользуемся снова библиотекой Cufflinks для построения графика медианной арендной платы за недвижимость:

```
In [24]: import cufflinks as cf
...: cf.go_offline()
...: df_comparison_rent.iplot(
...:     title="Median Monthly Rents Single Family Homes",
...:     xTitle="Year",
```

```

...:         yTitle="Monthly",
...:         #bestfit=True, bestfit_colors=["pink"],
...:         #subplots=True,
...:         shape=(4,1),
...:         #subplot_titles=True,
...:         fill=True,)
...:

```

На рис. 10.5 тенденции не так бросаются в глаза, в частности, из-за распределения данных по более короткому промежутку времени, но это отражает лишь неполную картину. Пало-Альто в указанном наборе данных нет, а арендная плата в остальных городах из Области залива Сан-Франциско намного ближе к медианной, в то время как арендная плата в Кливленде, штат Огайо, составляет около половины медианы по США.

В нашем заключительном исследовании мы посмотрим на аналогичный коэффициент арендной платы в масштабе США в целом. В следующем коде создается новый пустой объект DataFrame с коэффициентом арендной платы, после чего данные опять отправляются в Plotly:

```

In [25]: df_median_rents_ratio = pd.DataFrame()
...: df_median_rents_ratio["Cleveland_ratio_median"] =\
...:     df_comparison_rent["Cleveland"]/df_comparison_rent["Median USA"]
...: df_median_rents_ratio["San_Francisco_ratio_median"] =\
...:     df_comparison_rent["San Francisco"]/df_comparison_rent["Median
...:     USA"]
...: df_median_rents_ratio["Palo_Alto_ratio_median"] =\
...:     df_comparison_rent["Palo Alto"]/df_comparison_rent["Median USA"]
...: df_median_rents_ratio["Marin_County_ratio_median"] =\
...:     df_comparison_rent["Marin County"]/df_comparison_rent["Median
...:     USA"]
...:

```

```

In [26]: import cufflinks as cf
...: cf.go_offline()
...: df_median_rents_ratio.iplot(title="Median Monthly Rents Ratios
...:     Single Family Homes vs National Median",
...:         xTitle="Year",
...:         yTitle="Rent vs Median Rent USA",
...:         #bestfit=True, bestfit_colors=["pink"],
...:         #subplots=True,
...:         shape=(4,1),
...:         #subplot_titles=True,
...:         fill=True,)
...:

```

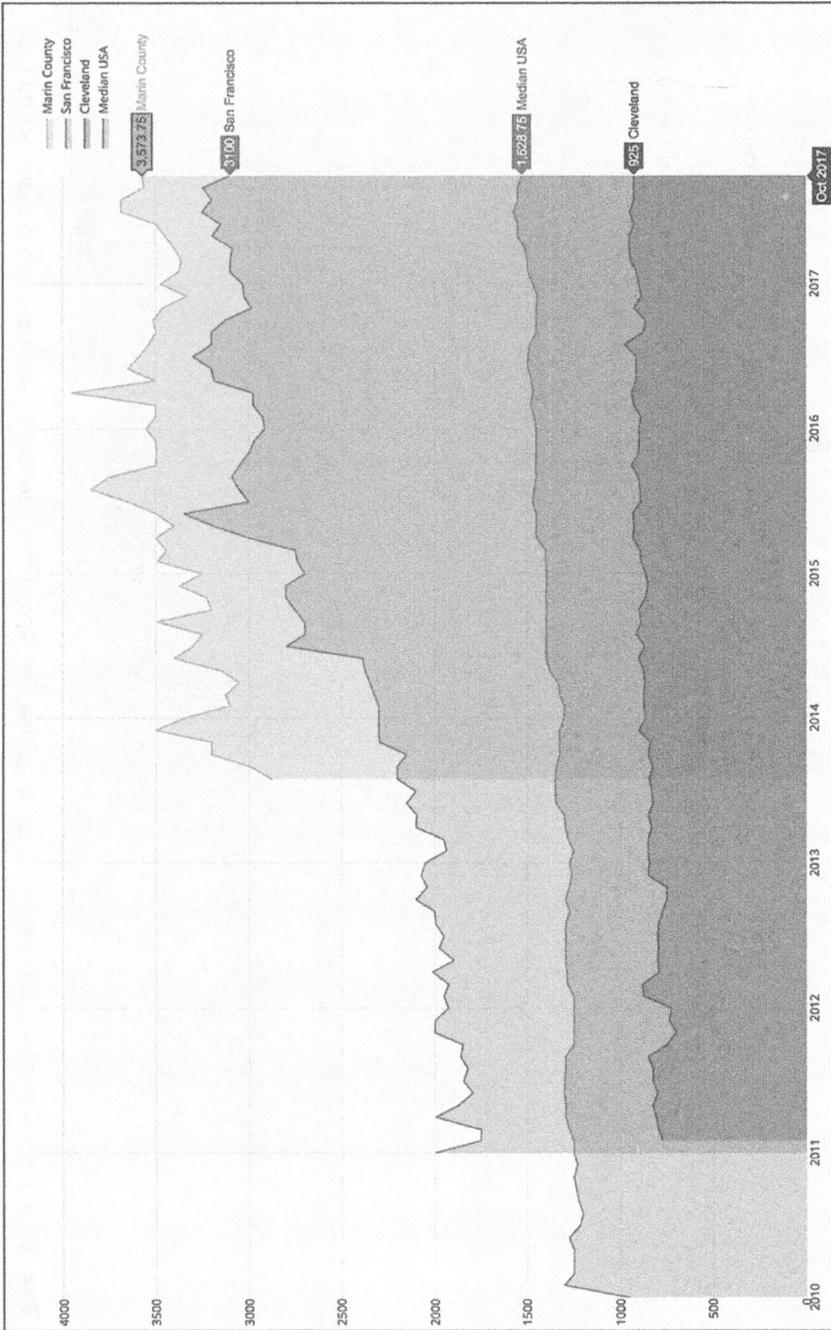


Рис. 10.5. Арендная плата в области залива Сан-Франциско повысилась почти вдвое с 2011 г., а в остальной части США осталась почти неизменной

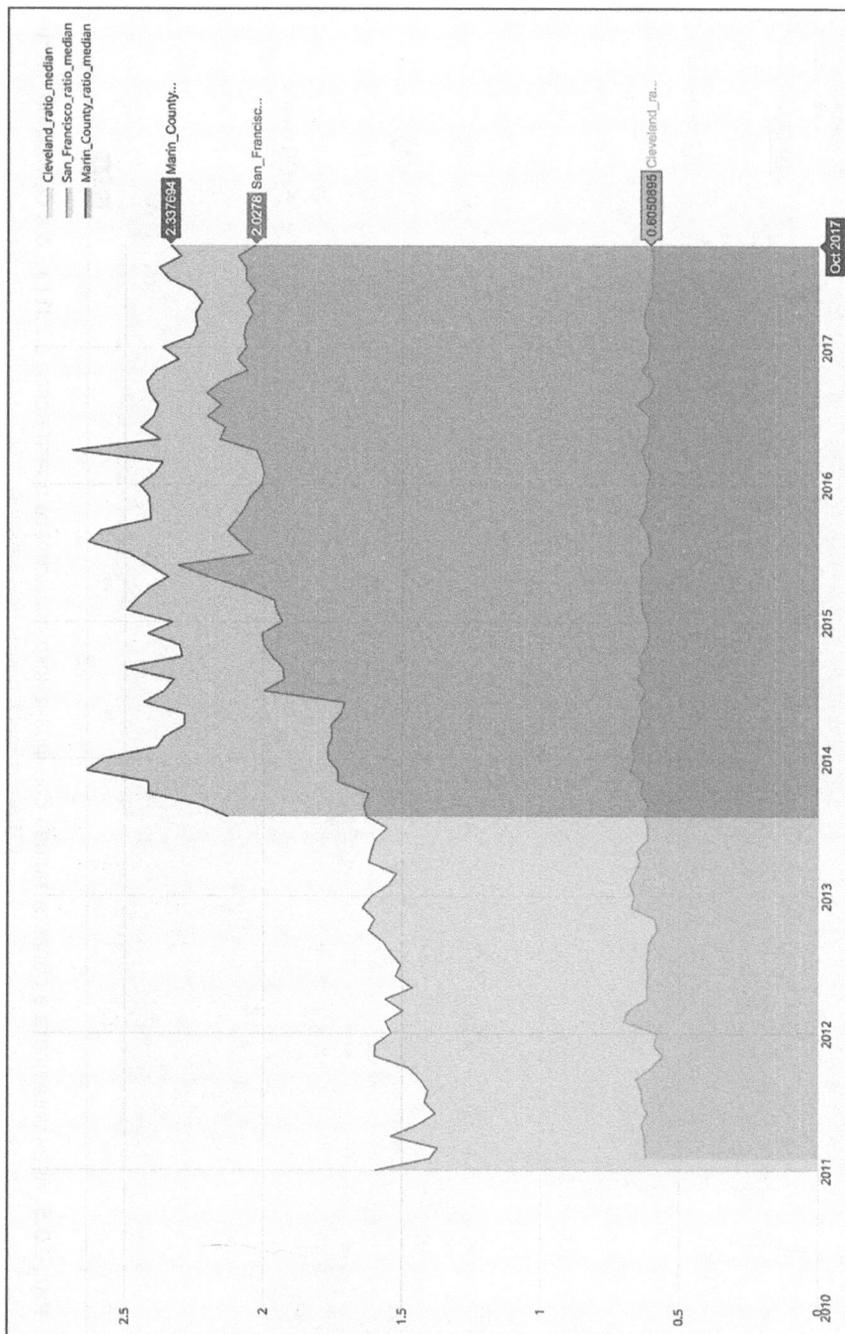


Рис. 10.6. Месячная арендная плата в области залива Сан-Франциско намного выше медианного значения по США

Рисунок 10.6 демонстрирует отличную от коэффициента роста картину. Медиана арендной платы в Сан-Франциско выше медианы в остальной части США вдвое, но все же отнюдь не в восемь раз, как медианная цена на дома. Судя по данным об арендной плате, лучше лишний раз задуматься перед покупкой дома в 2018 г., особенно в Пало-Альто. Аренда, несмотря на дороговизну, может оказаться более удачным решением.

Резюме

В этой главе мы провели исследование данных на основе общедоступного набора компании Zillow. Для создания интерактивных визуализаций данных на языке Python мы воспользовались библиотекой Plotly. Для получения дополнительной информации из сравнительно простого набора данных мы применили кластеризацию методом k-средних и 3D-визуализацию. Мы обнаружили, в частности, возможно, необоснованно раздутые цены на недвижимость в Области залива Сан-Франциско, особенно в Пало-Альто, в 2017 г. Имеет смысл провести дальнейшее исследование этого набора данных путем создания модели классификации относительно того, когда стоит продавать и когда купить недвижимость в каждой из местностей в США.

Дальнейшие перспективы развития подобного примера проекта включают в себя использование высокоуровневых API, например предоставляемых компанией House Canary (<https://api-docs.housecanary.com/#getting-started>). Возможно, ваше предприятие сможет создать на основе их прогностических моделей приложение ИИ, а затем добавить в него дополнительные слои ИИ и машинного обучения. Еще одна возможность: воспользоваться AWS SageMaker для создания прогноза на основе глубинного обучения, после чего развернуть эту модель для принятия бизнес-решений в вашей организации.

11

Промышленная эксплуатация ИИ для пользовательского контента

Победителем становится тот, кто способен двигаться вперед, несмотря на боль.

*Роджер Баннистер
(Roger Bannister)*

Какое отношение русские тролли, Facebook и выборы в США имеют к машинному обучению? В самом центре петли обратной связи соцсетей лежат рекомендательные системы и генерируемый пользователями благодаря им контент (user-generated content, UGC). Пользователи, присоединяющиеся к соцсети, получают рекомендации относительно пользователей и контента, которые их могут заинтересовать. Президентские выборы 2016 г. в США наглядно продемонстрировали, насколько важно понимать механизм работы рекомендательных систем, их возможности и ограничения.

Системы на основе ИИ отнюдь не панацея, они лишь предоставляют некий набор возможностей. Рекомендация подходящего товара в интернет-магазине может оказаться очень полезной, но в равной степени будет неприятно, если вам порекомендуют контент, который окажется фейком (например, сгенерированным агентами другого государства, которые хотят посеять разногласия в вашей стране).

В этой главе рассматриваются рекомендательные системы и обработка написанных на естественных языках текстов (natural language processing,

NLP) и с высокоуровневой точки зрения, и на уровне написания кода. Будут приведены примеры применения таких фреймворков, как основанная на языке Python рекомендательная система Surprise, а также инструкции по созданию своих собственных. Рассматриваемые вопросы включают премию Netflix (Netflix Prize), сингулярное разложение (singular-value decomposition, SVD), коллаборативную фильтрацию (collaborative filtering), практические вопросы использования рекомендательных систем, NLP и анализ тональности высказываний (sentiment analysis) в промышленных масштабах с помощью облачных API.

Получившее премию Netflix решение не было внедрено в промышленную эксплуатацию

Еще до того, как термин «наука о данных» стал общеизвестным и появился Kaggle, внимание всего мира было приковано к премии Netflix. Премия Netflix представляла собой конкурс, нацеленный на улучшение методов рекомендации новых фильмов. Многие из идей, возникших в ходе этого конкурса, позднее вдохновили различные компании и продукты. Конкурс с призом \$1 млн в 2006 г. стал предвестником эры современного ИИ. Что любопытно, в 2006 г., с запуском Amazon EC2, началась и эра облачных вычислений.

Облачные вычисления и широкое распространение ИИ были тесно связаны. Одним из крупнейших пользователей облачных сервисов посредством AWS была компания Netflix. Несмотря на все эти любопытные с исторической точки зрения факты, получивший премию Netflix алгоритм так никогда и не был внедрен в промышленную эксплуатацию. Команда разработчиков, победившая в 2009 г., BellKor's Pragmatic Chaos, добилась улучшения показателей более чем на 10 % при среднеквадратической ошибке 0,867 (<https://netflixprize.com/index.html>). Статья, созданная командой разработчиков (https://www.netflixprize.com/assets/ProgressPrize2008_BellKor.pdf), описывает данное решение как линейную композицию более чем 100 результатов. Уместно будет привести следующую цитату из этой статьи: «Из этого можно сделать вывод о том, что множество моделей может помочь при постепенном наращивании результатов, необходимом для победы в конкурсе, однако

на практике можно создать отличную систему на основе всего нескольких тщательно выбранных моделей».

Победивший в конкурсе Netflix подход не был внедрен в промышленную эксплуатацию из-за того, что сложность его разработки была слишком велика по сравнению с предполагаемой прибылью от использования. Основной применявшийся для рекомендаций алгоритм, SVD, как отмечается в статье «Быстрый алгоритм SVD для больших матриц» (<http://sysrun.haifa.il.ibm.com/hrl/bigml/files/Holmes.pdf>), «подходит для небольших наборов данных или офлайн-обработки, однако во многих современных приложениях применяется обучение в режиме реального времени и/или наборы данных большого объема или с большим числом размерностей». На практике одна из главных проблем промышленного применения машинного обучения — время и вычислительные ресурсы, необходимые для получения результатов.

Я сталкивался с этим при создании для различных компаний рекомендательных систем. Простой алгоритм, запускаемый пакетным образом, вполне может выдать полезные рекомендации. Но при выборе более сложного подхода или необходимости выдавать рекомендации в режиме реального времени сложность внедрения в промышленную эксплуатацию и/или сопровождения резко возрастает. Следовательно, чем проще — тем лучше, стоит выбирать машинное обучение в пакетном режиме, а не в режиме реального времени. Или выбирать простую модель, а не комплекс различных методик. А также подумать, не имеет ли смысл вызвать API механизма рекомендаций вместо того, чтобы создавать решение самим.

Ключевые понятия рекомендательных систем

На рис. 11.1 показана петля обратной связи рекомендаций соцсети. Чем больше у системы пользователей, тем больше контента она генерирует. А чем больше контента она генерирует, тем больше создает рекомендаций нового контента. Эта петля обратной связи, в свою очередь, привлекает новых пользователей и новый контент. Как уже упоминалось в начале данной главы, подобные возможности могут стать источником как положительных, так и отрицательных характеристик платформы.

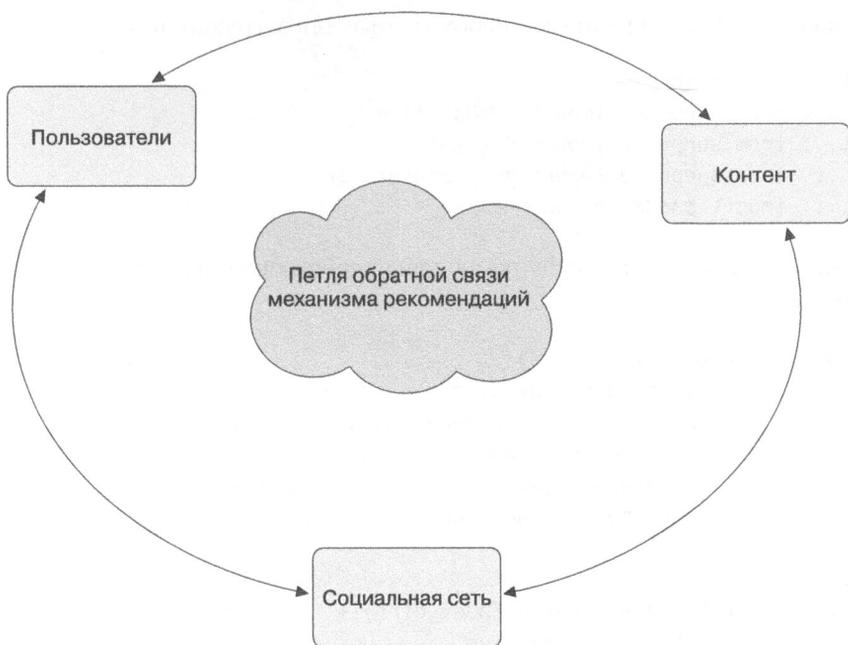


Рис. 11.1. Петля обратной связи рекомендаций социальной сети

Использование фреймворка Surprise в языке Python

Для изучения принципов работы рекомендательных систем можно воспользоваться, в частности, фреймворком Surprise (<http://surpriselib.com/>). В числе удобных его возможностей — встроенные наборы данных MovieLens (<https://grouplens.org/datasets/movielens/>) и Jester, встроенный SVD и другие распространенные алгоритмы, включая алгоритмы вычисления коэффициентов сходства (similarity measures). Он также включает инструменты для оценки качества рекомендаций в виде среднеквадратической погрешности (root mean squared error, RMSE) и средней абсолютной погрешности (mean absolute error, MAE), а также времени, которое занимает обучение модели.

Приведу пример его использования в близкой к реальной ситуации, несколько изменив один из предыдущих примеров.

Сначала выполняем импорты, необходимые для загрузки библиотеки:

```
In [2]: import io
...: from surprise import KNNBaseline
...: from surprise import Dataset
...: from surprise import get_dataset_dir
...: import pandas as pd
```

Создаем вспомогательную функцию для преобразования идентификаторов в имена:

```
In [3]: def read_item_names():
...:     """Считываем файл u.item из набора данных 100-k
...:         из MovieLens и возвращаем два отображения
...:         для преобразования исходных идентификаторов
...:         в названия фильмов, а названий фильмов –
...:         в первоначальные идентификаторы.
...:     """
...:
...:     file_name = get_dataset_dir() + '/ml-100k/ml-100k/u.item'
...:     rid_to_name = {}
...:     name_to_rid = {}
...:     with io.open(file_name, 'r', encoding='ISO-8859-1') as f:
...:         for line in f:
...:             line = line.split('|')
...:             rid_to_name[line[0]] = line[1]
...:             name_to_rid[line[1]] = line[0]
...:
...:     return rid_to_name, name_to_rid
```

Вычисляем коэффициенты сходства между элементами:

```
In [4]: # Прежде всего обучаем алгоритм, чтобы вычислить коэффициенты
...:     # сходства между элементами
...:     data = Dataset.load_builtin('ml-100k')
...:     trainset = data.build_full_trainset()
...:     sim_options = {'name': 'pearson_baseline',
...:                   'user_based': False}
...:     algo = KNNBaseline(sim_options=sim_options)
...:     algo.fit(trainset)
...:
...:
```

Estimating biases using als...

```
Computing the pearson_baseline similarity matrix...
```

```
Done computing similarity matrix.
```

```
Out[4]: <surprise.prediction_algorithms.knns.KNNBaseline>
```

Наконец, выдаем «10 рекомендаций», аналогичные другому примеру из этой главы:

```
In [5]: # Получаем отображения между исходными идентификаторами
# и названиями фильмов
...: rid_to_name, name_to_rid = read_item_names()
...:
...: # Извлекаем внутренний идентификатор фильма
...: # Toy Story ("История игрушек")
...: toy_story_raw_id = name_to_rid['Toy Story (1995)']
...: toy_story_inner_id = algo.trainset.to_inner_iid(
...:     toy_story_raw_id)
...:
...: # Извлекаем внутренние идентификаторы ближайших соседей
...: # фильма Toy Story
...: toy_story_neighbors = algo.get_neighbors(
...:     toy_story_inner_id, k=10)
...:
...: # Преобразуем внутренние идентификаторы соседей в названия
...: toy_story_neighbors = (algo.trainset.to_raw_iid(inner_id)
...:     for inner_id in toy_story_neighbors)
...: toy_story_neighbors = (rid_to_name[rid]
...:     for rid in toy_story_neighbors)
...:
...: print('The 10 nearest neighbors of Toy Story are:')
...: for movie in toy_story_neighbors:
...:     print(movie)
...:
```

The 10 nearest neighbors of Toy Story are:

Beauty and the Beast (1991)

Raiders of the Lost Ark (1981)

That Thing You Do! (1996)

Lion King, The (1994)

Craft, The (1996)

Liar Liar (1997)

Aladdin (1992)

Cool Hand Luke (1967)

Winnie the Pooh and the Blustery Day (1968)

Indiana Jones and the Last Crusade (1989)

При рассмотрении этого примера необходимо учесть проблемы, которые могут возникнуть при внедрении его в промышленную эксплуатацию. Вот пример псевдокода функции API, которую мог бы написать кто-нибудь из вашей компании:

```
def recommendations(movies, rec_count):  
    """Возвращает рекомендации"""  
  
movies = ["Beauty and the Beast (1991)", "Cool Hand Luke (1967)",.. ]  
  
print(recommendations(movies=movies, rec_count=10))
```

При реализации такой функции могут возникнуть вопросы вроде: на какие компромиссы вы идете, выбирая из группы лишь несколько первых записей; насколько хорошо этот алгоритм проявит себя при работе с очень большим набором данных? Единственно правильных ответов на подобные вопросы не существует в природе, но о них следует помнить при развертывании рекомендательных систем для промышленной эксплуатации.

Облачные решения для создания рекомендательных систем

В облачной платформе Google есть заслуживающий внимания пример использования машинного обучения на сервисе Google Compute Engine для получения рекомендаций товаров (<https://cloud.google.com/solutions/recommendations-using-machine-learning-on-compute-engine>). В нем, наряду с проприетарным облачным SQL, используются PySpark и алгоритм ALS (alternating least squares algorithm — метод чередующихся наименьших квадратов). Есть пример создания рекомендательной системы на основе их платформы (а также фреймворка Spark и сервиса Elastic Map Reduce (EMR)) и у Amazon (<https://aws.amazon.com/blogs/big-data/building-a-recommendation-engine-with-spark-ml-on-amazon-emr-using-zeppelin/>).

В обоих случаях с целью повышения производительности алгоритма используется фреймворк Spark, с помощью которого производится разделение вычислительной нагрузки по кластеру машин. Наконец, AWS активно продвигает сервис SageMaker (<https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html>), способный выполнять распределенные задания Spark нативным образом или взаимодействовать с кластером EMR.

Проблемы, возникающие на практике при работе с рекомендациями

Множество книг и статей, посвященных рекомендациям, фокусируются исключительно на технических аспектах рекомендательных систем. Эта же книга посвящена прагматической стороне вопроса, и с такой точки зрения есть несколько касающихся рекомендательных систем нюансов, которые стоит обсудить. Часть из них рассматривается в данном разделе: производительность, ETL, опыт взаимодействия пользователя (user experience, UX) и тролли/боты.

Как уже обсуждалось, сложность одного из наиболее популярных алгоритмов — $O(\text{число_выборок}^2 \times \text{число_признаков})$, то есть второго порядка. Это значит, что обучать модель в режиме реального времени и получить при этом оптимальное решение очень сложно. Следовательно, обучение рекомендательной системы в большинстве случаев придется производить в виде пакетного задания, без различных уловок вроде «жадных» эвристических правил и/или создания лишь небольшого подмножества рекомендаций для активных пользователей, имеющих особый спрос товаров, и т. д.

Когда я делал с нуля систему рекомендации подписчиков для социальной сети, то оказалось, что многие из этих проблем имеют самое насущное значение. Обучение модели занимало многие часы, так что другого варианта, кроме как запускать его ночью, не было. Кроме того, позднее я создал располагающуюся в оперативной памяти копию наших обучающих данных, так что ограничения алгоритма были связаны только с CPU, а не с вводом/выводом.

Производительность — непростой вопрос при создании рекомендательной системы для промышленной эксплуатации как в краткосрочной, так и в долгосрочной перспективе. По мере роста числа пользователей и товаров вашей компании вполне может оказаться, что принятый вами изначально подход не масштабируется так, как нужно. Возможно, приемлемые сначала, при 10 000 пользователей платформы, блокнот Jupiter, Pandas и scikit-learn окажутся недостаточно масштабируемым решением.

А вот основанный на PySpark алгоритм обучения, работающий по методу опорных векторов (<http://spark.apache.org/docs/2.1.0/api/python/pyspark.mllib.html>), может резко повысить производительность и снизить время обслуживания. А далее, возможно, вам понадобится перейти на специализированные

микросхемы для машинного обучения, например TPU или NVIDIA Volta. Обеспечение возможности расширения до нужных масштабов при создании изначально работоспособных решений — важнейший навык, неоценимый при реализации прагматических ИИ-решений, которые действительно годятся для промышленной эксплуатации.

Практические проблемы рекомендательных систем: интеграция с промышленными API. Я обнаружил, что в стартапах, занимающихся рекомендательными системами, при промышленной эксплуатации всплывает множество проблем. Причем в посвященных машинному обучению книгах этим проблемам уделяется не слишком много внимания. В примерах использования фреймворка Surprise всегда есть наготове обширная база «правильных ответов». На практике же обычно пользователей и товаров настолько мало, что смысла обучать модель нет. Что же делать?

В качестве приемлемого решения можно разбить работу рекомендательной системы на три этапа. На первом из них в качестве рекомендаций выдаются наиболее популярные пользователи, контент или товары. По мере роста объемов пользовательского контента на платформе — этап номер два — можно применить метод оценки сходства (без обучения модели). Ниже приведен предназначенный именно для этого «самодельный» код, пару раз употребившийся мной при промышленной эксплуатации. Для начала воспользуемся показателем Танимото (называемым также расстоянием Жаккара):

```
"""Алгоритмы науки о данных"""
def tanimoto(list1, list2):
    """Коэффициент Танимото

    In [2]: list2=['39229', '31995', '32015']
    In [3]: list1=['31936', '35989', '27489',
                  '39229', '15468', '31993', '26478']
    In [4]: tanimoto(list1,list2)
    Out[4]: 0.1111111111111111

    Определяет степень схожести двух множеств в числовом выражении
    с помощью операции пересечения

    """
    intersection = set(list1).intersection(set(list2))
    return float(len(intersection))/(len(list1) + \
                  len(list2) - len(intersection))
```

Далее начинается самое интересное. Скачиваем информацию о подписках и преобразуем в объект DataFrame библиотеки Pandas:

```
import os
import pandas as pd

from .algorithms import tanimoto

def follows_dataframe(path=None):
    """Создаем объект Dataframe для информации о подписчиках"""

    if not path:
        path = os.path.join(os.getenv('PYTHONPATH'),
                             'ext', 'follows.csv')

    df = pd.read_csv(path)
    return df
```

```
def follower_statistics(df):
    """Возвращает число подписчиков
```

```
In [15]: follow_counts.head()
Out[15]:
followerId
581bea20-962c-11e5-8c10-0242528e2f1b    1558
74d96701-e82b-11e4-b88d-068394965ab2     94
d3ea2a10-e81a-11e4-9090-0242528e2f1b     93
0ed9aef0-f029-11e4-82f0-0aa89fecadc2     88
55d31000-1b74-11e5-b730-0680a328ea36     64
Name: followingId, dtype: int64
```

```
"""
follow_counts = df.groupby(['followerId'])['followingId'].\
    count().sort_values(ascending=False)
return follow_counts
```

```
def follow_metadata_statistics(df):
    """Генерирует метаданные по подписчикам
```

```
In [13]: df_metadata.describe()
Out[13]:
count    2145.000000
mean         3.276923
std        33.961413
```

```

min            1.000000
25%           1.000000
50%           1.000000
75%           3.000000
max           1558.000000
Name: followingId, dtype: float64

"""

dfs = follower_statistics(df)
df_metadata = dfs.describe()
return df_metadata

def follow_relations_df(df):
    """Возвращает объект DataFrame для подписчика со всеми его
    связями"""

    df = df.groupby('followerId').followingId.apply(list)
    dfr = df.to_frame("follow_relations")
    dfr.reset_index(level=0, inplace=True)
    return dfr

def simple_score(column, followers):
    """Используется в качестве функции apply для объекта DataFrame"""

    return tanimoto(column, followers)

def get_followers_by_id(dfr, followerId):
    """Возвращает список подписчиков по заданному followerID"""

    followers = dfr.loc[dfr['followerId'] == followerId]
    fr = followers['follow_relations']
    return fr.tolist()[0]

def generate_similarity_scores(dfr, followerId, limit=10,
threshold=.1):
    """Генерирует список рекомендаций для заданного followerID"""
    followers = get_followers_by_id(dfr, followerId)
    recs = dfr['follow_relations'].\
        apply(simple_score, args=(followers,)).\
        where(dfr>threshold).dropna().sort_values()[-limit:]
    filters_rec = recs.where(recs>threshold)
    return filters_rec

def return_similarity_scores_with_ids(dfr, scores):

```

```

"""Возвращает показатели и FollowerID"""

dfs = pd.DataFrame(dfr, index=scores.index.tolist())
dfs['scores'] = scores[dfs.index]
dfs['following_count'] = dfs['follow_relations'].apply(len)
return dfs

```

Использовать этот API необходимо в такой последовательности:

```
In [1]: follows import *
```

```
In [2]: df = follows_dataframe()
```

```
In [3]: dfr = follow_relations_df(df)
```

```
In [4]: dfr.head()
```

```
In [5]: scores = generate_similarity_scores(dfr,
      "00480160-0e6a-11e6-b5a1-06f8ea4c790f")
```

```
In [5]: scores
```

```
Out[5]:
```

```
2144    0.000000
```

```
713     0.000000
```

```
714     0.000000
```

```
715     0.000000
```

```
716     0.000000
```

```
717     0.000000
```

```
712     0.000000
```

```
980     0.333333
```

```
2057    0.333333
```

```
3       1.000000
```

```
Name: follow_relations, dtype: float64
```

```
In [6]: dfs = return_similarity_scores_with_ids(dfr, scores)
```

```
In [6]: dfs
```

```
Out[6]:
```

```

                                     followerId \
980    76cce300-0e6a-11e6-83e2-0242528e2f1b
2057   f5ccb5f0-0e69-11e6-b5a1-06f8ea4c790f
3      00480160-0e6a-11e6-b5a1-06f8ea4c790f

```

```

                                     follow_relations    scores \
980    [f5ccb5f0-0e69-11e6-b5a1-06f8ea4c790f, 0048016...    0.333333

```

```
2057 [76cсe300-0e6a-11e6-83e2-0242528e2f1b, 0048016... 0.333333
3 [f5ccbf50-0e69-11e6-b5a1-06f8ea4c790f, 76cсe30... 1
```

```
following_count
980 2
2057 2
3 2
```

Данные рекомендации на основе показателей этапа 2 при текущей реализации должны будут работать в виде пакетного API. Помимо этого, при масштабировании библиотека Pandas рано или поздно столкнется с проблемами производительности. Разумным ходом будет перейти в какой-то момент на PySpark или Pandas на движке Ray (<https://rise.cs.berkeley.edu/blog/pandas-on-ray/?twitter=@bigdata>).

Для этапа 3 придется «расчехлить главный калибр» и воспользоваться чем-то наподобие фреймворка Surprise или пакета PySpark для обучения основанной на SVD модели и определения ее точности. Впрочем, зачем тратить на это усилия, когда в начале существования вашей компании формальное обучение модели машинного обучения никакой особой пользы не принесет?

Еще одна проблема промышленной эксплуатации API: что делать с отвергнутыми рекомендациями? Ничто не раздражает пользователя больше, чем постоянное получение рекомендаций чего-то ненужного или уже имеющегося. Так что придется решить еще одну навязчивую проблему промышленной эксплуатации. В идеале у пользователя должна быть возможность нажать на кнопку «больше не показывать» в списке рекомендаций, иначе они быстро превратятся в мусор. Кроме того, если пользователь что-то до вас доносит, то почему бы не воспользоваться этой информацией и не передать ее обратно в модель вашего механизма рекомендаций?

Облачный NLP и анализ тональности высказываний

У всех трех главных провайдеров облачных сервисов — AWS, GCP и Azure — есть надежные механизмы NLP с возможностью вызова через API. В этом разделе будут приведены примеры использования всех трех. Кроме того, мы создадим на платформе AWS реальный конвейер ИИ промышленного уровня для NLP с помощью бессерверной технологии.

NLP на платформе Azure

У когнитивных сервисов компании Microsoft (Microsoft Cognitive Services) есть API для текстовой аналитики с определением языка, извлечением ключевых фраз и анализом тональности высказываний. На рис. 11.2 создается конечная точка для вызовов API. В этом примере мы пройдемся по API на основе негативных отзывов о фильмах из набора данных рецензий на фильмы Корнеллского университета (<http://www.cs.cornell.edu/people/pabo/movie-review-data/>).

Прежде всего выполняем импорты в первом же блоке кода блокнота Jupiter:

```
In [1]: import requests
...: import os
...: import pandas as pd
...: import seaborn as sns
...: import matplotlib as plt
...:
```

Далее получаем из среды ключ API. Данный ключ API не «зашит» в коде, он был получен из раздела Keys показанной на рис. 11.2 консоли и экспортирован как переменная среды. Кроме того, мы присваиваем переменной значение, соответствующее URL API, который пригодится нам далее:

```
In [4]: subscription_key=os.environ.get("AZURE_API_KEY")
In [5]: text_analytics_base_url = \
...:     https://eastus.api.cognitive.microsoft.com/\
...:     text/analytics/v2.0/
```

Затем преобразуем один из негативных отзывов в формат, ожидаемый API:

```
In [9]: documents = {"documents":[]}
...: path = "../data/review_polarity/\
...:     txt_sentoken/neg/cv000_29416.txt"
...: doc1 = open(path, "r")
...: output = doc1.readlines()
...: count = 0
...: for line in output:
...:     count +=1
...:     record = {"id": count, "language": "en", "text": line}
...:     documents["documents"].append(record)
...:
...: #выводим
...: documents
```

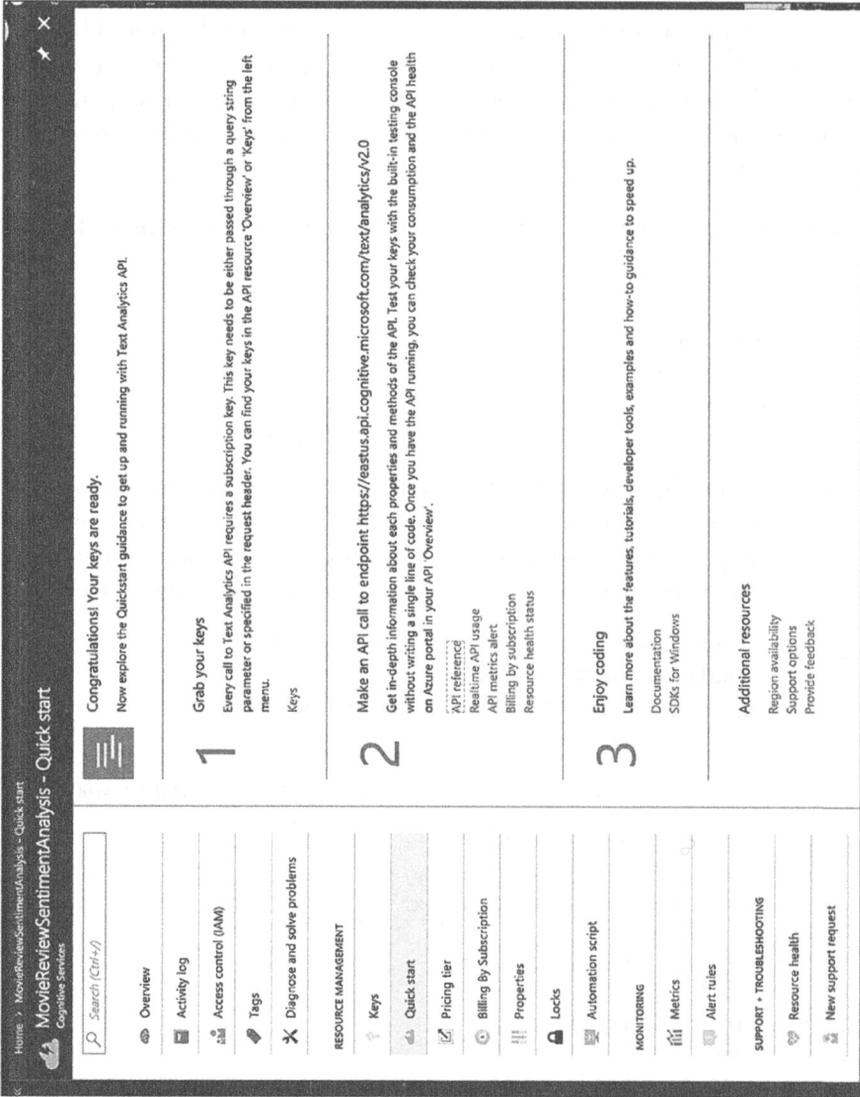


Рис. 11.2. API когнитивных сервисов платформы Microsoft Azure

При этом создается структура данных следующей формы:

Out[9]:

```
{'documents': [{'id': 1,
  'language': 'en',
  'text': 'plot : two teen couples go to a\
    church party , drink and then drive . \n'},
  {'id': 2, 'language': 'en',
  'text': 'they get into an accident . \n'},
  {'id': 3,
  'language': 'en',
  'text': 'one of the guys dies ,\
    but his girlfriend continues to see him in her life,\
    and has nightmares . \n'},
  {'id': 4, 'language': 'en', 'text': "what's the deal ? \n"},
  {'id': 5,
  'language': 'en',
```

Наконец, применяем API анализа тональности высказываний для оценки отдельных документов:

```
{'documents': [{'id': '1', 'score': 0.5},
  {'id': '2', 'score': 0.13049307465553284},
  {'id': '3', 'score': 0.09667149186134338},
  {'id': '4', 'score': 0.8442018032073975},
  {'id': '5', 'score': 0.808459997177124
```

Теперь можно преобразовать возвращаемые оценки в объект `DataFrame` библиотеки `Pandas` для разведочного анализа данных. Неудивительно, что медианное значение тональностей для отрицательных отзывов равно 0,23 по шкале от 0 до 1, где 1 — исключительно положительный отзыв, а 0 — исключительно отрицательный.

```
In [11]: df = pd.DataFrame(sentiments['documents'])
```

```
In [12]: df.describe()
```

Out[12]:

	score
count	35.000000
mean	0.439081
std	0.316936
min	0.037574
25%	0.159229

50% 0.233703
 75% 0.803651
 max 0.948562

Для большей наглядности нарисуем график плотности распределения. Рисунок 11.3 демонстрирует, что преобладают весьма негативные тональности.

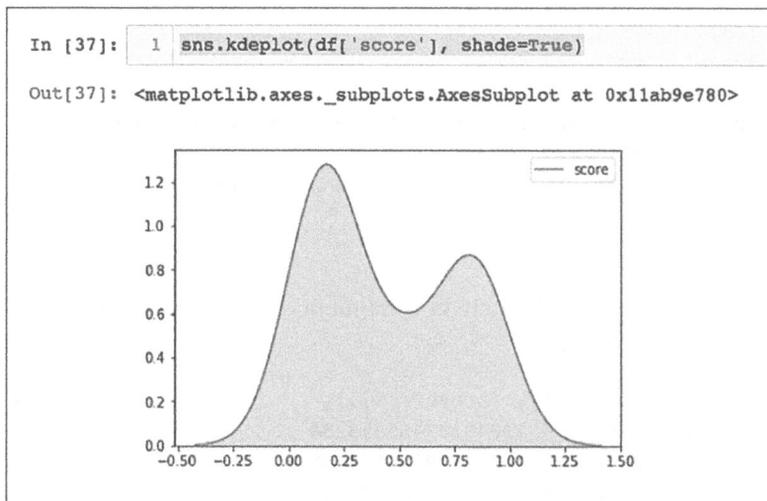


Рис. 11.3. График плотности распределения оценок тональностей

NLP на платформе GCP

У API облачного сервиса Google обработки написанных на естественных языках текстов (<https://cloud.google.com/natural-language/docs/how-to>) есть множество привлекательных черт. В их числе — возможность его использования двумя способами: для анализа тональности заданной строки и анализа тональности текста из облачного хранилища Google. У облачных сервисов Google имеется также обладающая потрясающими возможностями утилита командной строки, сильно упрощающая изучение их API. Наконец, у них есть несколько захватывающих API ИИ, и некоторые из них мы рассмотрим в этой главе: анализ тональности высказываний, анализ сущностей, анализ синтаксиса, анализ тональности сущностей и классификацию контента.

Изучаем API сущностей

С помощью утилиты командной строки `gcloud` очень удобно исследовать, что делают различные API. Например, отправляем через командную строку фразу относительно Леброна Джеймса и команды «Кливленд Кавальерс»:

```
→ gcloud ml language analyze-entities --content=\
"LeBron James plays for the Cleveland Cavaliers."
{
  "entities": [
    {
      "mentions": [
        {
          "text": {
            "beginOffset": 0,
            "content": "LeBron James"
          },
          "type": "PROPER"
        }
      ],
      "metadata": {
        "mid": "/m/01jz6d",
        "wikipedia_url": "https://en.wikipedia.org/wiki/LeBron_James"
      },
      "name": "LeBron James",
      "salience": 0.8991045,
      "type": "PERSON"
    },
    {
      "mentions": [
        {
          "text": {
            "beginOffset": 27,
            "content": "Cleveland Cavaliers"
          },
          "type": "PROPER"
        }
      ],
      "metadata": {
        "mid": "/m/0jm7n",
        "wikipedia_url": "https://en.wikipedia.org/wiki/Cleveland_Cavaliers"
      },
    },
  ],
}
```

```

    "name": "Cleveland Cavaliers",
    "salience": 0.100895494,
    "type": "ORGANIZATION"
  }
],
"language": "en"
}

```

Второй способ изучения API — с помощью Python. Для получения ключа API и аутентификации следуйте инструкциям, которые можно найти по адресу <https://cloud.google.com/docs/authentication/getting-started>. Далее запустите блокнот Jupyter в командной оболочке, в которой экспортирована переменная `GOOGLE_APPLICATION_CREDENTIALS`:

```

→ X export GOOGLE_APPLICATION_CREDENTIALS=\
    /Users/noahgift/cloudai-65b4e3299be1.json
→ X jupyter notebook

```

После завершения данного процесса аутентификации остается самое простое. Во-первых, необходимо импортировать языковой API Python (если он у вас еще не установлен, то его можно установить с помощью системы управления пакетами `pip`: `pip install --upgrade google-cloud-language`):

```

In [1]: # Импортируем клиентскую библиотеку облачных сервисов Google
...: from google.cloud import language
...: from google.cloud.language import enums
...: from google.cloud.language import types

```

Далее мы отправляем нашу фразу в API и получаем обратно метаданные сущности с анализом:

```

In [2]: text = "LeBron James plays for the Cleveland Cavaliers."
...: client = language.LanguageServiceClient()
...: document = types.Document(
...:     content=text,
...:     type=enums.Document.Type.PLAIN_TEXT)
...: entities = client.analyze_entities(document).entities
...:

```

При использовании командной строки результаты аналогичны, но возвращаются в виде списка Python:

```

[name: "LeBron James"
type: PERSON

```

```

metadata {
  key: "mid"
  value: "/m/01jz6d"
}
metadata {
  key: "wikipedia_url"
  value: "https://en.wikipedia.org/wiki/LeBron_James"
}
salience: 0.8991044759750366
mentions {
  text {
    content: "LeBron James"
    begin_offset: -1
  }
  type: PROPER
}
, name: "Cleveland Cavaliers"
type: ORGANIZATION
metadata {
  key: "mid"
  value: "/m/0j7n"
}
metadata {
  key: "wikipedia_url"
  value: "https://en.wikipedia.org/wiki/Cleveland_Cavaliers"
}
salience: 0.10089549422264099
mentions {
  text {
    content: "Cleveland Cavaliers"
    begin_offset: -1
  }
  type: PROPER
}
]

```

Можно сделать, в частности, вывод, что этот API легко объединить с другими типами исследований, наподобие произведенных в главе 6 «Прогноз популярности в соцсетях в НБА». Несложно представить себе создание приложения ИИ с этими API NLP в качестве отправной точки, для поиска расширенной информации о социальных авторитетах. Еще один вывод: возможности предоставляемой когнитивными API GCP утилиты командной строки весьма широки.

Бессерверный конвейер ИИ промышленного уровня для NLP на платформе AWS

AWS лучше любого из остальных трех основных провайдеров облачных сервисов подходит для создания пригодных для промышленной эксплуатации приложений, удобных в написании и сопровождении. Решающее их новшество — сервис AWS Lambda. Он подходит как для организации конвейеров, так и для обслуживания конечных точек HTTP, например, как при работе с Chalice. На рис. 11.4 представлен реально эксплуатируемый конвейер NLP.

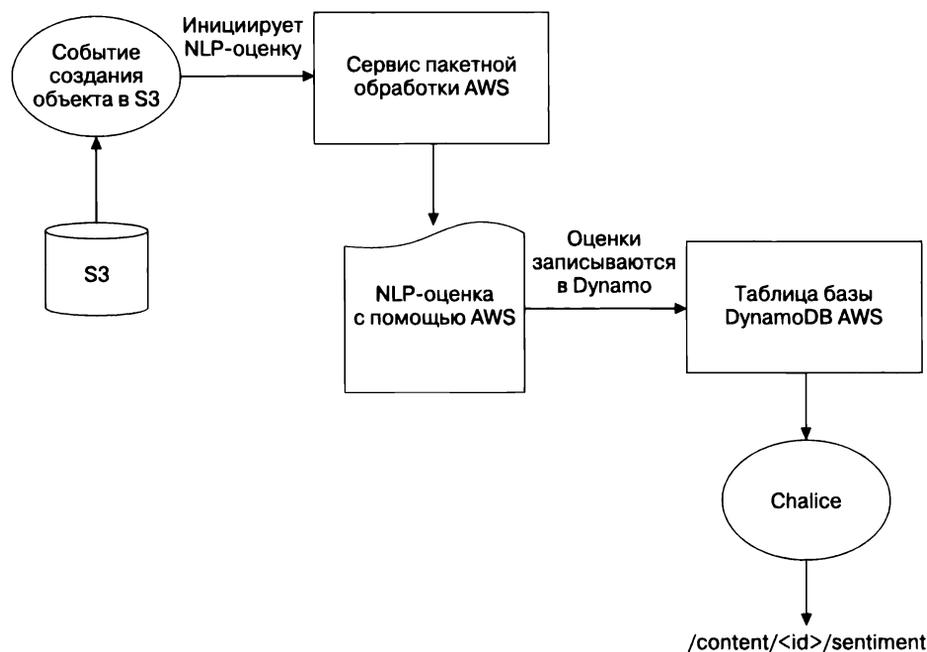


Рис. 11.4. Бессерверный NLP-конвейер промышленного уровня на платформе AWS

Чтобы приступить к анализу тональности высказываний на платформе AWS, необходимо сначала импортировать несколько библиотек:

```
In [1]: import pandas as pd
...: import boto3
...: import json
```

Далее выполняем простую проверку:

```
In [5]: comprehend = boto3.client(service_name='comprehend')
...: text = "It is raining today in Seattle"
...: print('Calling DetectSentiment')
...: print(json.dumps(comprehend.detect_sentiment(\
...:     Text=text, LanguageCode='en'), sort_keys=True, indent=4))
...:
...: print('End of DetectSentiment\n')
...:
```

В результате получаем оценку тональности (SentimentScore):

Calling DetectSentiment

```
{
  "ResponseMetadata": {
    "HTTPHeaders": {
      "connection": "keep-alive",
      "content-length": "164",
      "content-type": "application/x-amz-json-1.1",
      "date": "Mon, 05 Mar 2018 05:38:53 GMT",
      "x-amzn-requestid": \
        "7d532149-2037-11e8-b422-3534e4f7cfa2"
    },
    "HTTPStatusCode": 200,
    "RequestId": "7d532149-2037-11e8-b422-3534e4f7cfa2",
    "RetryAttempts": 0
  },
  "Sentiment": "NEUTRAL",
  "SentimentScore": {
    "Mixed": 0.002063251566141844,
    "Negative": 0.013271247036755085,
    "Neutral": 0.9274052977561951,
    "Positive": 0.057260122150182724
  }
}
```

End of DetectSentiment

Теперь в более реалистичном примере воспользуемся вышеприведенным документом с негативными отзывами о фильмах. Читаем документ:

```
In [6]: path = "/Users/noahgift/Desktop/review_polarity/\
txt_sentoken/neg/cv000_29416.txt"
...: doc1 = open(path, "r")
...: output = doc1.readlines()
...:
```

Далее оцениваем один из документов (напоминаю, что API NLP рассматривают каждую строку как отдельный документ):

```
In [7]: print(json.dumps(comprehend.detect_sentiment(\
Text=output[2], LanguageCode='en'), sort_keys=True, inden
...: t=4))
```

```
{
  "ResponseMetadata": {
    "HTTPHeaders": {
      "connection": "keep-alive",
      "content-length": "158",
      "content-type": "application/x-amz-json-1.1",
      "date": "Mon, 05 Mar 2018 05:43:25 GMT",
      "x-amzn-requestid": \
        "1fa0f6e8-2038-11e8-ae6f-9f137b5a61cb"
    },
    "HTTPStatusCode": 200,
    "RequestId": "1fa0f6e8-2038-11e8-ae6f-9f137b5a61cb",
    "RetryAttempts": 0
  },
  "Sentiment": "NEUTRAL",
  "SentimentScore": {
    "Mixed": 0.1490383893251419,
    "Negative": 0.3341641128063202,
    "Neutral": 0.468740850687027,
    "Positive": 0.04805663228034973
  }
}
```

Ничего удивительного, что у этого документа негативная оценка тональности, ведь он предварительно и был оценен как негативный. Интересно также, что этот API может выдать одну единую оценку для всех документов. По сути, таким образом он возвращает медианное значение тональности. Вот как это выглядит на практике:

```
In [8]: whole_doc = ', '.join(map(str, output))
```

```
In [9]: print(json.dumps(\
comprehend.detect_sentiment(\
  Text=whole_doc, LanguageCode='en'), sort_keys=True, inden
...: t=4))
```

```
{
```

```

"ResponseMetadata": {
  "HTTPHeaders": {
    "connection": "keep-alive",
    "content-length": "158",
    "content-type": "application/x-amz-json-1.1",
    "date": "Mon, 05 Mar 2018 05:46:12 GMT",
    "x-amzn-requestid": \
      "8296fa1a-2038-11e8-a5b9-b5b3e257e796"
  },
  "Sentiment": "MIXED",
  "SentimentScore": {
    "Mixed": 0.48351600766181946,
    "Negative": 0.2868672013282776,
    "Neutral": 0.12633098661899567,
    "Positive": 0.1032857820391655
  }
}
)= 'en'), sort_keys=True, inden
...: t=4))

```

Интересный вывод: у API AWS есть свои «тузы в рукаве», как и свои мелкие недостатки, отсутствующие в API Azure. В предыдущем примере Azure выводимые Seaborn результаты демонстрируют наличие бимодального распределения, в котором отзывов о том, что фильм понравился, — меньшинство, а что не понравился — большинство. AWS весьма удачно описывает это путем представления результата в виде тональности MIXED («неоднородная»).

Осталось только создать простое приложение Chalice, которое бы выдавало записанные в базу данных Дупамто результаты. Вот как выглядит соответствующий код:

```

from uuid import uuid4
import logging
import time

from chalice import Chalice
import boto3
from boto3.dynamodb.conditions import Key
from pythonjsonlogger import jsonlogger

#Переменные среды приложения
REGION = "us-east-1"

```

```
APP = "nlp-api"
NLP_TABLE = "nlp-table"

#Инициализация журналирования
log = logging.getLogger("nlp-api")
LOGHANDLER = logging.StreamHandler()
FORMMATTER = jsonlogger.JsonFormatter()
LOGHANDLER.setFormatter(FORMMATTER)
log.addHandler(LOGHANDLER)
log.setLevel(logging.INFO)

app = Chalice(app_name='nlp-api')
app.debug = True

def dynamodb_client():
    """Создает клиент Dynamodb"""

    extra_msg = {"region_name": REGION, "aws_service": "dynamodb"}
    client = boto3.client('dynamodb', region_name=REGION)
    log.info("dynamodb CLIENT connection initiated", extra=extra_msg)
    return client

def dynamodb_resource():
    """Создает ресурс Dynamodb"""

    extra_msg = {"region_name": REGION, "aws_service": "dynamodb"}
    resource = boto3.resource('dynamodb', region_name=REGION)
    log.info("dynamodb RESOURCE connection initiated",\
            extra=extra_msg)
    return resource

def create_nlp_record(score):
    """Создает запись в таблице NLP_TABLE

    """

    db = dynamodb_resource()
    pd_table = db.Table(NLP_TABLE)
    guid = str(uuid4())
    res = pd_table.put_item(
        Item={
            'guid': guid,
            'UpdateTime' : time.asctime(),
            'nlp-score': score
        }
    )
```

```

extra_msg = {"region_name": REGION, "aws_service": "dynamodb"}
log.info(f"Created NLP Record with result{res}", extra=extra_msg)
return guid

def query_nlp_record():
    """Просматривает таблицу NLP_TABLE и извлекает из нее все записи"""

    db = dynamodb_resource()
    extra_msg = {"region_name": REGION, "aws_service": "dynamodb",
                 "nlp_table": NLP_TABLE}
    log.info(f"Table Scan of NLP table", extra=extra_msg)
    pd_table = db.Table(NLP_TABLE)
    res = pd_table.scan()
    records = res['Items']
    return records

@app.route('/')
def index():
    """Маршрут по умолчанию"""

    return {'hello': 'world'}

@app.route("/nlp/list")
def nlp_list():
    """Список всех оценок из таблицы NLP_TABLE"""

    extra_msg = {"region_name": REGION,
                 "aws_service": "dynamodb",
                 "route": "/nlp/list"}
    log.info(f"List NLP Records via route", extra=extra_msg)
    res = query_nlp_record()
    return res

```

Резюме

Если данные — новая нефть, то генерируемые пользователями данные (UGC) — нефтеносные пески. Исторически наладить на основе нефтеносных песков промышленную добычу нефти было нелегко, но рост цен на энергию и развитие технологий сделали это экономически целесообразным. Точно так же API ИИ трех основных провайдеров облачных сервисов привели к технологическому прорыву в сфере просеивания «песчаных данных». Кроме того, цены на хранение и вычислительные ресурсы неуклонно падали, значительно повышая целесообразность обращения генерируемых

пользователями данных в актив, из которого можно извлекать прибыли. Еще одно новшество, благодаря которому стоимость обработки UGC снизилась, — аппаратные ускорители ИИ. Существенные усовершенствования технологий распараллеливания с помощью микросхем ASIC, таких как TPU, GPU и программируемые пользователями вентиляемые матрицы (field-programmable graphic array, FPGA), сильно упрощают решение обсуждавшихся выше проблем с масштабированием.

В данной главе было приведено множество примеров извлечения прибыли из этих наших «нефтеносных песков», но, как и с настоящими нефтеносными песками, тут есть свои компромиссы и проблемы. Возможны такие способы «обмана» петель обратной связи от UGC к ИИ и использования их уязвимых мест, что последствия будут иметь глобальный масштаб. Кроме того, с более практической точки зрения следует учесть определенные нюансы при начале эксплуатации системы в онлайн-режиме. Каким бы простым ни делали облачные API ИИ создание программных решений, существуют вопросы, от которых нельзя просто отмахнуться, например опыт взаимодействия пользователя, производительность и коммерческие последствия реализованных программных решений.

Приложения

A

Аппаратные ускорители для ИИ

Аппаратные ускорители для ИИ — относительно новая, но быстро развивающаяся технология. Существует несколько их разновидностей: новые или сделанные на заказ образцы, образцы на основе GPU, ИИ-сопроцессоры и научно-исследовательские разработки. Наиболее шумевшие из категории новинок — TPU, благодаря предоставляемому ими удобству разработки ПО на основе библиотеки TensorFlow.

Изделия на основе GPU, вероятно, наиболее распространенный сейчас вариант ускорителей для ИИ. Профессор Иэн Лэйн (Ian Lane) из Университета Карнеги-Мэллона говорит о них: «С помощью GPU можно расшифровывать заранее записанный речевой сигнал или мультимедийный контент гораздо быстрее. По сравнению с реализацией на основе CPU распознавание может производиться в 33 раза быстрее».

В категории FPGA (field-programmable gate arrays) стоит отметить компанию Reconfigure.io (<https://reconfigure.io/>). Она дает разработчикам возможность удобного использования FPGA для ускорения работы их программных решений, включая ИИ-решения. С помощью простых инструментов и широких возможностей облачной сборки и развертывания Reconfigure.io обеспечивает скорость, низкое значение задержки и эффективность, доступные ранее лишь проектировщикам аппаратного обеспечения. Они предоставляют интерфейс на основе языка Go для компиляции, оптимизации и развертывания кода на Go в облачных FPGA на AWS. FPGA активно применяются всеми основными провайдерами облачных сервисов из-за своего удобства при использовании ИИ в сети, а также потребности в низком энергопотреблении.

Хотя и GPU и FPGA значительно превосходят CPU по производительности, интегральные схемы специального назначения (application-specific integrated circuits, ASIC) превосходят их в этом отношении на порядок.

Так что главное преимущество таких вещей, как FPGA, — в ускорении разработки приложений за счет использования более привычных инструментов, вроде языка Go.

Вот несколько вопросов, касающихся аппаратных ускорителей для ИИ, которые не помешает обдумать.

1. Каковы требования к производительности приложения и к общей структуре издержек ЦОД, а также критерии для принятия решения о реализации аппаратных ускорителей с целью применения в предполагаемом приложении?
2. Каковы сценарии использования предполагаемого приложения в ЦОД для ускорения работы?

Каждая компания, желающая создавать передовой ИИ, должна отслеживать аппаратные ускорители для ИИ. Причина — в производительности. Невозможно игнорировать столь колоссальные технологические достижения, как 30-кратное превосходство GPU и FPGA над CPU в смысле производительности, не говоря уже о дальнейшем десятикратном преимуществе над GPU и FPGA специализированных ASIC. Они могут привести к разработке неизвестных доселе видов ИИ.

Б

Выбор размера кластера

В данной книге приведено множество примеров кластеризации методом k-средних. Один из самых часто задаваемых вопросов на эту тему: сколько кластеров должно быть? Единственного правильного ответа на подобный вопрос не существует, поскольку кластеризация — процесс формирования меток и у двух разных экспертов в предметной области могут быть разные мнения на сей счет.

На рис. Б.1 приведена кластеризация статистики НБА сезона 2013–2014 гг., где я маркировал восемь кластеров метками, которые показались мне уместными. Другой эксперт по НБА, возможно, создал бы меньше или больше кластеров.

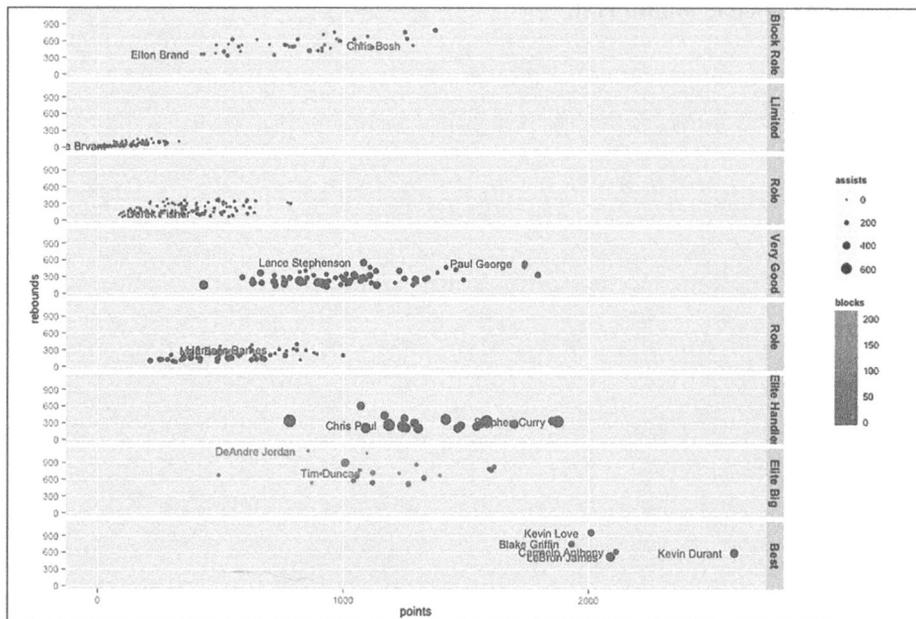


Рис. Б.1. Кластеризация сезона НБА

Существуют, однако, некоторые способы, которые могут помочь в выборе числа кластеров. В документации по библиотеке `scikit-learn` приведено несколько хороших примеров оценки эффективности кластеризации (<http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation>). Два популярных метода для этого — метод «локтя» и метод анализа силуэтов. Именно с них рекомендуется начать, если похоже, что распределение по кластерам можно улучшить.

Ной Гифт

**Прагматичный ИИ. Машинное обучение
и облачные технологии**

Перевел с английского И. Пальти

Заведующая редакцией
Руководитель проекта
Ведущий редактор
Литературный редактор
Художественный редактор
Корректор
Верстка

*Ю. Сергиенко
О. Сивченко
Н. Гринчик
Е. Рафалюк-Бузовская
В. Мостипан
Е. Павлович
Г. Блинов*

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: 12.2018. 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 18.12.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 308.

Отпечатано в АО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор»

142300, Московская область, г. Чехов, ул. Полиграфистов, 1

Сайт: www.chpd.ru, E-mail: sales@chpd.ru

тел: 8(499) 270-73-59

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: <http://www.piter.com/page/avtoru>

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, гоб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, гоб. 6216;
e-mail: books@piter.com

Вопросы по продаже электронных книг: тел.: (812) 703-73-74, гоб. 6217;
e-mail: kuznetsov@piter.com

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF – самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – Анна Титова, (812) 703-73-73, titova@piter.com
Москва – Сергей Клебанов, (495) 234-38-15, klebanov@piter.com

Мощные готовые облачные бизнес-решения от Amazon, Google и Microsoft. Проверенные приемы Data Science с примерами на Python

Облачные технологии — ваш путь к укрощению искусственного интеллекта.

Тщательно изучив эту незаменимую книгу от Ноя Гифта, легендарного эксперта по языку Python, вы легко научитесь писать облачные приложения с использованием средств искусственного интеллекта и машинного обучения, решать реалистичные задачи из таких востребованных и актуальных областей, как спортивный маркетинг, управление проектами, ценообразование, сделки с недвижимостью.

Все примеры разобраны на языке Python, № 1 в сфере современных стремительных вычислений.

Прагматичный подход + амбициозные задачи = искусственный интеллект.

НОЙ ГИФТ. Преподаватель и консультант по программе Management MSBA в аспирантуре Калифорнийского университета в Дэвисе. На протяжении карьеры занимал должности СТО, главного менеджера, СТО-консультанта, архитектора облачных решений. Основатель компании Pragmatic AI Labs, занимается консультированием стартапов и других компаний по вопросам машинного обучения и облачных архитектур. Член Python Software Foundation (сфера ответственности — машинное обучение), автор книг по машинному обучению и DevOps.

«Это краткое руководство — то самое недостающее звено, объединяющее безграничные возможности искусственного интеллекта и те нетривиальные задачи, которые необходимо решать для развертывания реальных проектов. Понятная и практичная, эта книга станет вашим ключом к настоящему Python и алгоритмам ИИ».

Кристофер Бруссо, основатель и генеральный директор корпоративной платформы ИИ Surface Owl

«Фантастическое дополнение к списку обязательной литературы для фанатов новых технологий! Столько всего можно сказать про эту книгу! Ной Гифт создал по-настоящему практичное руководство для всех, кто имеет дело с коммерческим применением машинного обучения. Она не только рассказывает, как использовать машинное обучение для больших наборов данных, но и дает читателю представление о внутренних взаимосвязях этой технологии. Книга позволит множеству команд разработчиков и исследователей данных с первых дней наладить эффективную разработку и поддержку самых сложных проектов».

Нивас Дурайрадж, специалист по технической поддержке, AWS (сертифицированный архитектор решений AWS уровня Professional)



Заказ книг:
тел.: (812) 703-73-74
books@piter.com

 [instagram.com/piterbooks](https://www.instagram.com/piterbooks)

 [youtube.com/ThePiterBooks](https://www.youtube.com/ThePiterBooks)

 vk.com/piterbooks

WWW.PITER.COM
каталог книг и интернет-магазин

 facebook.com/piterbooks

ISBN: 978-5-4461-1061-2



9 785446 110612