Текст, таблицы, изображения и аудио

Анализ данных с LLM



Иммануэль Труммер

MANNING

Переводчик *С. Черников*

Иммануэль Труммер

Анализ данных с LLM. Текст, таблицы, изображения и аудио. — СПб.: Питер, 2026.

ISBN 978-5-4461-4505-8

© ООО Издательство "Питер", 2026

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

Предисловие

Моей любимой семье.

Первое использование большой языковой модели сродни волшебству. Я до сих пор помню свой первый опыт общения с моделью GPT-3 (сейчас уже устаревшей). Тогда мне показалось, что компьютер наконец-то понимает меня и может адекватно реагировать на самые разные сложные запросы. Более того, я давал различные задания — от анализа текста до написания кода — и модель выполняла их, основываясь только на моих инструкциях! Я привык к миру, в котором нейронные сети приходилось обучать для решения узкоспециализированных задач, используя большое количество обучающих данных, характерных для конкретной задачи, которые нужно было маркировать вручную. Поэтому то, что я увидел, стало настоящей революцией, открывшей дверь в мир невероятных возможностей.

Я увлекся и с тех пор посвятил значительную часть своей профессиональной карьеры использованию удивительных возможностей языковых моделей. Я пришел из сферы анализа данных, поэтому для меня было совершенно естественно смотреть на модели под этим же углом: как с их помощью извлечь максимальную пользу из наборов данных? С момента, как я стал использовать языковые модели, типы данных, с которыми они могут работать, сильно изменились. Изначальная сфера использования моделей — анализ текстов — значительно расширилась и теперь охватывает мультимодальные данные, включая изображения, аудио, видео и текст. Это делает языковые модели бесценным инструментом для любых задач науки о данных, позволяя пользователям

создавать сложные аналитические конвейеры с помощью всего нескольких строк кода на языке Python, а также инструкции для модели на естественном языке с описанием задачи, которую нужно решить.

По роду деятельности я регулярно встречаюсь со специалистами по анализу и обработке данных, которые могли бы извлечь огромную пользу из возможностей, предоставляемых языковыми моделями. Однако освоить эту новую область может быть непросто.

Чтобы собрать воедино информацию, необходимую для использования языковых моделей в различных задачах анализа данных, мне приходилось читать разные блоги и онлайн-учебники. Мне хотелось бы иметь под рукой такую книгу, как эта, когда я начинал свой путь. Надеюсь, она станет для вас полезным источником информации, а работа с ней доставит вам удовольствие!

Благодарности

Выражаю особую благодарность редакции издательства Manning и всем сотрудникам, которые работали над подготовкой рукописи к печати и помогли этой книге обрести ее окончательную форму. Кроме того, спасибо Тимоти Эндрю Робертсу, научному редактору книги.

Кроме того, я благодарен всем рецензентам: Элу Пежевски, Амитабху Премраджу Чикоту, Аниндите Натх, Анто Аравинту, Брендану О'Харе, Клеменс Баадер, Даррину Бишопу, Дотану Коэну, Эли Майост, Джорджу Э. Картеру, Гири Сваминатану, Харчарану Каббаю, Икечукву Оконкво, Хауме Вальсу Альтадилу, Джереми Чену, Джону Гатри, Джону В. Маккарти, Джону Уильямсу, Кшиштофу Држеевскому, Лексу Дреннану, Марсио Франсиско Ногейре, Марджори Розуэлл, Марвину Шварцу, Полу Силистеану, Рахулу Джайну, Роберту Розплоху, Сумиту Бхаттачарии, Свапне Йелесварапу, Тьяго Бритто Борхесу, Тодду Куку, Тони Холдройду, Ватсалу Десаи, Виноту Нагешварану и Уолтеру Александру Мата Лопесу.

Ваши предложения помогли сделать книгу лучше.

О книге

Эта книга написана, чтобы помочь разработчикам создавать приложения для анализа мультимодальных данных с использованием современных языковых технологий. Здесь рассматриваются языковые модели и основные библиотеки Python, которые понадобятся для работы с ними. С помощью серии мини-проектов показано, как использовать языковые модели для анализа текста, табличных данных, графов, изображений, видео- и аудиофайлов. Изучив такие темы, как промпт-инжиниринг (создание подсказок для языковых моделей), дообучение и современные программные фреймворки, вы сможете с помощью моделей быстро создавать сложные, производительные и экономически эффективные приложения для анализа данных.

Кому следует прочитать эту книгу

Кем бы вы ни были — разработчиком программного обеспечения, специалистом по работе с данными или любителем, интересующимся этой сферой, — эта книга для вас, если вы хотите использовать мощные возможности больших языковых моделей для выполнения различных видов анализа данных.

Предварительный опыт работы с языковыми моделями не требуется, так как в книге изложены все основы. Однако полезно иметь опыт программирования на Python, хотя бы на базовом уровне, поскольку примеры взаимодействия с моделями приведены именно на этом языке.

Структура издания

Книга состоит из десяти глав, разделенных на три части. Часть I знакомит с языковыми моделями и позволяет составить первое впечатление о преимуществах их использования для анализа данных.

- Глава 1 содержит общее описание языковых моделей и объяснение принципов их применения в области анализа данных.

- В главе 2 рассказывается о чат-боте ChatGPT, показан анализ текстовых и табличных данных в веб-интерфейсе ChatGPT.

Часть II знакомит с библиотекой компании OpenAI для Python и показывает, как анализировать различные типы данных с помощью языковых моделей непосредственно на Python.

- Глава 3 рассказывает о библиотеке компании OpenAI для Python, позволяющей пользователям отправлять запросы к языковым моделям и настраивать их поведение различными способами.

- В главе 4 показано, как использовать языковые модели для обработки текстовых данных: например, для классификации документов или извлечения конкретных сведений.

- В главе 5 демонстрируется процесс создания интерфейсов запросов на естественном языке с использованием языковых моделей, которые переводят вопросы на естественном языке в формальные запросы, обращаясь к табличным или графовым данным.

- В главе 6 описывается использование мультимодальных языковых моделей для обработки изображений или видеоданных, чтобы решать такие задачи, как обнаружение объектов, ответы на вопросы и генерация описаний.

- В главе 7 представлено несколько вариантов использования языковых моделей при анализе аудиоданных: например, транскрибация аудиозаписей, реализация интерфейсов голосовых запросов или перевод голосового ввода на другие языки.

Часть III посвящена темам для углубленного изучения, позволяющим оптимизировать выбор моделей, конфигураций и фреймворков.

- В главе 8 рассматриваются различные поставщики больших языковых моделей, а также дается краткий обзор предлагаемых ими моделей и соответствующих библиотек Python.

- В главе 9 демонстрируются методы, которые можно использовать для минимизации стоимости обработки и максимизации качества результатов при работе с языковыми моделями, включая оптимизацию выбора модели и настройки параметров, а также дообучение.

- В главе 10 рассматривается несколько программных фреймворков, в частности LangChain и LlamaIndex, которые позволяют создавать сложные приложения на основе больших языковых моделей, сокращая затраты на реализацию.

Начинать чтение рекомендуется с главы 1, в которой приведены важные термины и понятия. Вы можете пропустить главу 2, если уже работали с языковыми моделями через веб-интерфейсы. В большинстве других глав речь идет о библиотеке OpenAI для Python. Поэтому стоит прочитать главу 3, прежде чем переходить к следующим. Главы 4–7 посвящены различным типам данных, и их можно читать в любом порядке. Аналогично главы 8–10 не связаны друг с другом, поэтому очередность их изучения может быть любой.

О коде

Эта книга содержит множество примеров исходного кода в нумерованных и ненумерованных листингах. Полный код в пронумерованных листингах доступен для скачивания по ссылкам, размещенным на сайте книги **www.dataanalysiswithllms.com** в разделе **Resources**. Ссылки на файлы с кодом и сопутствующие тестовые данные сгруппированы в соответствии с главами книги. Кроме того, весь репозиторий кода и данные можно скачать с сайта издательства: **www.manning.com/books/data-analysis-with-llms**.

Исходный код отформатирован моноширинным шрифтом, чтобы отделить его от обычного текста. Во многих случаях оригинальный код был переформатирован: добавлены разрывы строк и изменены отступы, чтобы он уместился на странице. Кроме того, если код листинга был описан в тексте, то комментарии из него часто удалялись. Многие листинги снабжены аннотациями, в которых описаны важные концепции.

Об авторе

Иммануэль Труммер — доцент факультета компьютерных наук Корнеллского университета. Область его исследований находится на пересечении анализа данных и машинного обучения. В частности, он занимается изучением применения больших языковых моделей в сфере анализа данных, что привело к появлению ряда публикаций, отмеченных наградами, и к сотрудничеству с компаниями. Его видеоуроки набрали более миллиона просмотров на YouTube. Помимо работы с языковыми моделями, Иммануэль любит играть на скрипке, исследует живописные уголки северной части штата Нью-Йорк и старается проводить как можно больше времени со своей семьей.

Иллюстрация на обложке

Иллюстрация называется *Le Spéculateur*, или «Размышляющий», и взята из книги Анри-Леона Кюрмера, опубликованной в 1841 году. Каждая иллюстрация в том издании была нарисована и раскрашена вручную.

В те времена по одежде было легко определить, где живут люди и каков их род занятий или положение в обществе. Издательство Manning делает акцент на изобретательности и инициативности в компьютерной сфере, оформляя обложки своих книг в стиле богатого разнообразия региональных культур, существовавших столетия назад. Мы освежаем это разнообразие в памяти, используя картины из различных памятных коллекций.

От издательства

Мы выражаем огромную благодарность клубу рецензентов ИТ-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

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

Мы будем рады узнать ваше мнение!

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

О научном редакторе русскоязычного издания

Сергей Боярко — старший инженер-разработчик в компании «Туту». Разрабатывает AI-решения для поддержки операторов и автоматизации контактного центра. Занимался разработкой голосовых роботов и персональных ассистентов с использованием современных LLM и мультиагентных подходов. Аккредитованный судья Федерации спортивного программирования, преподаватель ПГГПУ, ведет курсы «Программирование на Python» и «Проектирование информационных интеллектуальных систем».

Часть I. Знакомство с языковыми моделями

Так что же представляют собой языковые модели? И как можно использовать их для анализа данных?

В этой части книги содержатся ответы на оба вопроса.

В главе 1 мы обсудим принципы, лежащие в основе языковых моделей, и выясним, что делает их особенными. Кроме того, мы рассмотрим самые разные способы использования языковых моделей для анализа данных, в том числе варианты их работы непосредственно с данными и возможность использования в качестве интерфейсов для более специализированных инструментов анализа данных.

В главе 2 мы «поболтаем» с ChatGPT — то есть пообщаемся с популярной языковой моделью компании OpenAI. Мы убедимся, что ChatGPT способна адаптироваться к выполнению различных заданий, касающихся текста: начиная от классификации текста и заканчивая извлечением конкретных фрагментов информации из текста на основе краткого описания задачи. Кроме того, мы увидим, что ChatGPT хорошо справляется с переводом вопросов о данных, сформулированных на естественном языке, на формальные языки запросов, например на SQL.

После прочтения этих глав у вас должно сложиться четкое представление о том, что такое языковые модели и как их можно использовать для анализа данных.

1. Анализ данных с помощью больших языковых моделей

В этой главе

- Введение в языковые модели.
- Анализ данных с помощью языковых моделей.
- Эффективное использование языковых моделей.

Языковые модели — это мощные нейронные сети, которые можно использовать для решения различных задач по обработке данных. В этой главе вы познакомитесь с такими моделями и узнаете, как и зачем их применяют для анализа данных.

1.1. На что способны языковые модели

Начнем раздел с небольшого стихотворения и связанного с ним рисунка (рис. 1.1), сочетающего в себе две главные темы этой книги — анализ данных и большие языковые модели.

Стихотворение и иллюстрация были сгенерированы GPT-4o (o означает omni[1]), языковой моделью компании OpenAI, которая обрабатывает мультимодальные данные, исключительно на основе инструкций: «Напиши стихотворение, используя слова “анализ данных” и “большие языковые модели”», а затем «Теперь нарисуй соответствующую картинку!». И рисунок, и стихотворение, похоже, относятся к заданной теме. Конечно, стихотворение вряд ли может

рассчитывать на литературную премию, но его текст связный, имеет структуру, характерную для стихотворения, и рифмуется!

В ночной тиши, при свете серверов
Потоки данных мчатся без оков.
Столбцы и строки — выверенный строй,
В них смысл сокрыт, он встречи ждет с тобой.

В числах — эхо прожитых времен,
Закономерностей уверенный закон.
И вот приходит новый искатель —
Языковая модель, прилежный читатель.

Внимает всей мощью, что схемы вложили,
Чтоб факты сухие свой смысл раскрыли.
От рассеянных чисел — к четким фразам,
Лишь язык данных ей близок и ясен.

Здесь цифры кружат, текст вступает в спор,
Где код и смысл ведут единый разговор.
Два мира слиты в связке без границ —
Логика и речь без пауз и страниц.

В таком союзе, выверенном, строгом,
У данных смысл становится итогом.
В глубинах сети, где импульсы бегут,
Анализ и язык единый путь найдут.



Рис. 1.1. Иллюстрация GPT-4o, связывающая понятия «анализ данных» и «большие языковые модели»

Пожалуй, самое главное, что для создания стихотворения и картинки потребовались лишь короткие инструкции, написанные на естественном языке. И если раньше для методов машинного обучения нужны были большие объемы обучающих данных, характерные для конкретной задачи, то теперь это требование устарело. И конечно, задача достаточно конкретная, чтобы убедить нас в том, что языковая модель не копирует существующие решения из Интернета, а генерирует уникальный контент.

Написание стихотворений и создание изображений — лишь два из множества возможных вариантов использования языковых моделей (хотя, пожалуй, самые интересные). Модели наподобие GPT-4o способны решать различные задачи, такие как краткое изложение текстовых документов, написание программного кода и ответы на вопросы о картинках. В этой книге вы узнаете, как использовать языковые модели для решения множества задач анализа данных, начиная с извлечения информации из больших коллекций текстовых документов и заканчивая написанием кода для анализа данных. Прочитав ее, вы сможете быстро создавать конвейеры анализа данных, основанные на языковых моделях, и извлекать полезные сведения из различных форматов данных.

Что означает GPT

Аббревиатура GPT (от Generative Pretrained Transformer) расшифровывается как «генеративный предварительно обученный трансформер».

Генеративный: GPT — это большая нейронная сеть, которая генерирует содержимое (например, текст или код) в ответ на входной текст. Этот факт отличает ее от других нейронных

сетей, которые, например, могут только классифицировать входной текст, используя фиксированный набор заранее определенных категорий.

Предварительно обученный: GPT предварительно обучается на больших объемах данных, решая типовые задачи, такие как предсказание следующего слова в тексте. Как правило, задача предварительного обучения отличается от задач, для которых обычно используется модель. Однако предварительное обучение помогает быстрее осваивать более специализированные задачи.

Трансформер — это новая архитектура нейронной сети, которая особенно полезна для обучения задачам, где входные и выходные данные имеют переменную длину (например, текстовые документы). На данный момент это доминирующая архитектура для подходов, используемых в области генеративного ИИ.

1.2. Чему вы научитесь

Эта книга посвящена тому, как с помощью языковых моделей решать задачи анализа данных. Такие задачи можно классифицировать по типу изучаемых данных и по типу анализа. В книге рассматривается широкий спектр типов данных и задач анализа.

Мы сосредоточимся на *мультимодальном* анализе, то есть на использовании языковых моделей для анализа различных типов данных. Конкретно в этой книге рассматриваются следующие типы данных.

- *Текст.* Это могут быть электронные письма, газетные статьи и комментарии на веб-форуме. Текстовые данные распространены повсеместно и содержат ценную информацию. В книге мы рассмотрим, как использовать языковые модели для автоматической классификации текстовых документов на основе их содержания, как извлекать из текста конкретные фрагменты информации и как группировать текстовые документы по смежным темам.

- *Изображения.* Как говорится, одна картинка стоит тысячи слов. Изображения помогают нам понять сложные концепции, запечатлеть приятные воспоминания о последнем отпуске и проиллюстрировать текущие события. Языковые модели способны легко извлекать информацию из изображений. Например, мы будем использовать модели для ответа на произвольные вопросы об изображениях или для идентификации людей на фотографиях на основе базы данных профилей.

- *Видеоролики.* Значительную часть данных в Интернете составляют видеоданные. Даже на вашем смартфоне они, скорее всего, занимают немалую часть общего объема памяти. Книга покажет, что языковые модели можно применять и для анализа видеороликов: например, для генерации подходящих названий на основе содержания.

- *Аудиозаписи.* Для многих людей речь является наиболее естественной формой общения. Аудиозаписи фиксируют монологи и беседы, а также дополняют видеоролики. В книге мы рассмотрим, как расшифровывать аудиозаписи, как переводить разговорную речь на другие языки, а также как создать интерфейс запросов, который будет отвечать на устные вопросы о данных.

- *Таблицы.* Для примера возьмем набор данных, содержащий информацию о клиентах. Эти данные удобнее всего представлять в виде таблицы: в столбцах хранится информация об адресе, номере телефона и кредитной карте, а в строках — сведения о разных клиентах. В книге мы рассмотрим, как использовать языковые модели для написания кода, выполняющего сложные операции с такими табличными данными.

- *Графы.* Многие наборы данных (от социальных сетей до сетей метрополитена) удобно представлять в виде графов, моделирующих сущности (например, людей или станции метро) и их связи (например, дружеские отношения или пересадки). Мы посмотрим, как можно использовать языковые модели для создания кода, анализирующего большие графы различными способами.

В книге мы в основном будем работать с моделями компании OpenAI, используя ее же библиотеку для Python. Ближе к концу книги мы рассмотрим языковые модели и других поставщиков. Поскольку у библиотек разных поставщиков обычно схожая функциональность, то переход на другие модели не займет много времени.

Структурированные и неструктурированные данные

Типы данных часто делят на две группы: *структурированные* и *неструктурированные данные*. У структурированных есть структура, которая облегчает их эффективную обработку с помощью специализированных инструментов. Примеры структурированных данных — таблицы и графы. При работе с такими данными языковую модель обычно используют как интерфейс к специализированным инструментам обработки

данных. Неструктурированные данные — такие как текст, изображения, видео- и аудиофайлы — не имеют структуры, которую можно было бы легко использовать для эффективной обработки. Поэтому при взаимодействии с неструктурированными данными языковую модель обычно приходится применять непосредственно к самим данным.

Как правило, использование языковых моделей влечет за собой денежные расходы, пропорциональные объему обрабатываемых данных. Плата зависит от используемой языковой модели, ее конфигурации, а также от того, как сформулированы входные данные для этой модели. В этой книге вы не только научитесь решать различные задачи анализа данных с помощью языковых моделей, но и узнаете, как делать это с минимальными затратами.

1.3. Как использовать языковые модели

Работа с современными языковыми моделями строится на методе *составления промптов (запросов)*. В этом разделе мы обсудим сам метод, а также интерфейсы, которые можно для этого использовать.

1.3.1. Составление промптов

Всего несколько лет назад модели машинного обучения обучали для решения одной конкретной задачи. Например, модель могла классифицировать текст отзыва как «положительный» (то есть автор отзыва доволен) или «отрицательный» (то есть автор недоволен). При использовании этой модели в качестве входных данных достаточно было ввести текст отзыва. Не нужно было

описывать во входных данных задачу (классификация отзыва), поскольку модель была специально обучена выполнять только ее.

Ситуация изменилась в последние годы с появлением больших языковых моделей наподобие GPT. Такие модели больше не обучаются для решения конкретных задач. Вместо этого они призваны служить универсальными инструментами, которые в принципе способны справиться с любой задачей по желанию пользователя. При работе с такой моделью пользователь должен четко описать, какие действия она должна выполнить.

Промпт представляет собой входные данные для языковой модели. Он может содержать мультимодальные данные: например, текст и изображения. Как минимум чтобы заставить языковую модель решить конкретную задачу, промпт должен содержать текст с инструкциями о том, какие действия должна выполнить модель. Помимо инструкций, в промпте должен содержаться весь необходимый контекст.

Например, если в инструкции сказано определить, виден ли на картинке автомобиль, то промпт также должен содержать картинку. Инструкции должны быть конкретными и уточнять, к примеру, ожидаемый формат вывода. Например, мы хотим, чтобы модель выдавала 1 при наличии автомобиля и 0 — при его отсутствии, что позволит нам быстро сложить выведенные моделью числа и подсчитать количество автомобилей. Все это необходимо четко прописать в промпте (в противном случае модель может ответить: «Да, на картинке есть автомобиль», что усложнит подсчет на этапе последующей обработки). Помимо инструкций и контекста, промпт может содержать примеры, помогающие языковой модели понять задачу.

Обучение на нескольких примерах или без примеров

Чтобы помочь языковой модели лучше понять задачу, в качестве части промпта можно предоставить примеры. Они похожи на задачу, для решения которой планируется использовать модель. Примеры содержат входные данные и желаемый вывод. Такой подход иногда называется *обучением на нескольких примерах (few-shot learning)*, поскольку модель обучается задаче, используя некоторое количество примеров[2]. *Обучение без примеров (zero-shot learning)*, в свою очередь, означает, что модель будет изучать задачу, не имея ни одного примера, основываясь только на ее описании.

1.3.2. Пример промпта

Проиллюстрируем промпты с помощью примера. Классический сценарий использования языковых моделей — анализ отзывов о товаре с целью определить тональность отзыва: является ли он положительным (то есть покупатель рекомендует товар) или отрицательным (то есть клиент недоволен товаром). Предположим, у нас есть отзыв, который необходимо определить как положительный или отрицательный. При наличии специализированной модели, обученной классифицировать отзывы на конкретную интересующую нас категорию товаров, достаточно отправить отзыв в эту модель. Она специально обучена для решения конкретной задачи, поэтому уже «знает», что делать с входными данными и какой формат вывода нужен. Однако, поскольку мы используем большие языковые модели, вместе с отзывом необходимо предоставить дополнительный контекст.

Промпт должен содержать всю необходимую модели информацию, описывая задачу, которую нужно решить, и весь

контекст. В нашем примере, вероятно, стоит добавить в промпт следующую информацию:

- *текст отзыва*, который требуется классифицировать;
- *описание задачи*, которую нужно решить;
- *форматы вывода*, которые нам нужны;
- *релевантный контекст* — например, мы рассматриваем отзывы на ноутбуки или газонокосилки?

По желанию можно добавить несколько примеров отзывов, уже классифицированных должным образом. Это может помочь модели выполнить задачу более точно.

В промпте ниже содержится вся необходимая информация для примера отзыва (листинг 1.1).

Листинг 1.1. Составление промпта для классификации отзыва о ноутбуке

```
Мы будем рассматривать отзывы о товарах – ноутбуках.      ❶ Контекст
Для каждого отзыва выведи "Доволен" или "Недоволен" в зависимости от того,
доволен покупатель товаром или нет.      ❷ Описание задачи и формат вывода
Примеры:
Отличный ноутбук! Всем рекомендую к покупке!
Доволен                                  ❸ Первый пример
Ноутбук не работал. Пришлось его вернуть.
Недоволен                                ❹ Второй пример
Экран слишком маленький, а операционная система очень
долго загружается.                       ❺ Отзыв
```

Промпт начинается с описания релевантного контекста ❶. Клиенты пишут отзывы о ноутбуках, поэтому, например, если они отмечают, что товар «тяжелый», то чаще всего это недостаток (в отличие от анализа отзывов, скажем, о паровых катках). В описании задачи ❷ содержится инструкция для модели, что делать с отзывами, и указывается желаемый формат вывода (вывести «Доволен» или «Недоволен»). Далее

идет список примеров. Строго говоря, для такой простой задачи может и не потребоваться добавлять в промпт примеры. Однако это порой может повысить точность вывода. Здесь мы добавили два примера отзывов (❸ и ❹), а также указали желаемый вывод для этих отзывов. Наконец, мы добавили сам отзыв ❺, который модель должна классифицировать.

Учитывая предыдущую информацию в промпте, современные языковые модели, получив такой промпт, скорее всего, выдадут результат «Недоволен». Что, собственно, нам и требовалось получить.

1.3.3. Интерфейсы

Каким же образом отправлять промпты языковой модели? Поставщики наподобие OpenAI обычно предлагают веб-интерфейсы, позволяющие пользователям отправлять их языковым моделям отдельные промпты. В главе 2 мы будем пользоваться веб-интерфейсом OpenAI для отправки промптов, предписывающих модели проанализировать текст или написать код для обработки данных.

Веб-интерфейс работает хорошо до тех пор, пока в него отправляют всего несколько промптов. Однако для анализа большой коллекции текстовых документов потребуется отправить множество промптов (по одному на каждый текстовый документ). Очевидно, что вводить тысячи промптов вручную не хочется. Именно в такой ситуации и можно использовать библиотеку OpenAI на языке Python. Она позволяет отправлять промпты моделям OpenAI непосредственно из кода Python и работать с ответами модели в среде Python. Так можно автоматизировать загрузку данных, генерацию промптов и любую необходимую последующую обработку ответов модели. Кроме того, это позволяет объединять языковые модели с другими полезными

инструментами: например, чтобы с помощью языковой модели писать код для обработки данных и тут же выполнять его, используя другие инструменты.

С работой библиотеки OpenAI для Python вы познакомитесь в главе 3, а затем будете использовать ее в других главах. Такие поставщики языковых моделей, как Google, Anthropic и Cohere, предлагают аналогичные Python-библиотеки для отправки промптов своим языковым моделям. Более подробно об этих библиотеках мы поговорим в главе 8.

1.4. Использование языковых моделей для анализа данных

Как же использовать языковые модели именно для анализа данных? В книге описываются два варианта.

- Во-первых, можно использовать языковую модель *непосредственно* для работы с данными. Это означает, что языковая модель получит данные, которые необходимо проанализировать, как часть промпта (вместе с инструкциями по выполнению анализа).

- Во-вторых, языковая модель может применяться для анализа данных *косвенно*. В этом случае она не видит сами данные: то есть они не добавляются в промпт. Вместо этого с помощью языковой модели пишется код для обработки данных, выполняемый в специализированных инструментах обработки.

Какой подход применить, зависит от свойств данных и задачи. Рассмотрим оба метода подробнее.

1.4.1. Использование языковых моделей непосредственно для работы с данными

Проще всего анализировать данные с помощью языковых моделей — поместив их непосредственно в промпт. Именно так мы поступили в подразделе 1.3.2. Чтобы проанализировать отзыв, мы добавили текст отзыва в промпт вместе с инструкциями о том, что с ним нужно сделать. Такой же подход можно применять и для работы с другими типами данных, нетекстовыми. Например, при использовании мультимодальных моделей наподобие GPT-4o, помимо инструкций по анализу, можно просто добавить в промпт сами изображения, которые необходимо проанализировать.

Как правило, требуется проанализировать не одно изображение или отзыв, а их набор. Например, предположим, что нам нужно классифицировать целый набор отзывов и определить для каждого из них, каким он является: положительным или отрицательным. В таких случаях обычно применяется следующий подход, реализованный на языке Python с помощью библиотеки OpenAI для Python (или эквивалентной библиотеки, позволяющей пользователям отправлять промпты другим моделям других провайдеров). Мы загружаем отзывы, которые требуется классифицировать, и генерируем по одному промпту для каждого отзыва. Затем отправляем эти промпты в языковую модель, извлекаем результаты классификации из ответов, сгенерированных моделью для каждого отзыва, и сохраняем результаты в файле на диске.

В этом сценарии требуется решить одну и ту же задачу (классификация отзывов) для нескольких текстовых документов (то есть отзывов). Как вы можете представить, промпты для разных отзывов должны быть схожими. Несмотря на то что текст отзыва, который нужно классифицировать,

каждый раз меняется, описание задачи и другие части промпта остаются неизменными.

Для создания промпта на языке Python используется *шаблон промпта*. Этот шаблон определяет промпт, связанный с конкретной задачей, которую необходимо решить. В нашем примере мы используем шаблон промпта для разделения отзывов на две категории: положительные или отрицательные. Шаблон содержит метки-заполнители для обозначения частей промпта, которые изменяются в зависимости от входных данных. Учитывая, что наш шаблон предназначен для классификации отзывов, в нем нужно оставить место для текста отзыва. Затем при генерации промпта в Python мы заменим эту метку-заполнитель (placeholder) текстом отзыва, который требуется классифицировать.

Например, для классификации отзывов можно использовать следующий шаблон (листинг 1.2).

В этом шаблоне обобщен промпт, с помощью которого мы намеревались классифицировать один конкретный отзыв (см. листинг 1.1 в подразделе 1.3.2). Опять же, мы предоставляем контекст (сообщение, что классифицируются отзывы о ноутбуках) ❶ и инструкции, описывающие задачу, которую нужно решить, а также формат вывода ❷ . Кроме того, мы приводим несколько примеров отзывов с соответствующими результатами классификации (❸ и ❹). Несмотря на то что отзывы, которые требуется классифицировать, меняются в зависимости от исходных данных, примеры отзывов менять не нужно. Эти отзывы просто иллюстрируют, какую задачу должна решить языковая модель. Наконец, ❺ у нас есть место для текста отзыва. При итерациях с разными отзывами мы будем генерировать промпты для каждого из них, подставляя текст отзыва в метку-заполнитель.

https://t.me/it_boooks/2

Листинг 1.2. Шаблон промпта для классификации отзывов о ноутбуках

```
Мы будем рассматривать отзывы о товарах — ноутбуках.        ❶ Контекст
Для каждого отзыва выведи "Доволен" или "Недоволен" в зависимости от того,
доволен покупатель товаром или нет.
Примеры:                                    ❷ Описание задачи и формат вывода
Отличный ноутбук! Всем рекомендую к покупке!
Доволен                                     ❸ Первый пример
Ноутбук не работал. Пришлось его вернуть.
Недоволен                                   ❹ Второй пример
[Отзыв]                                     ❺ Метка-заполнитель для текста отзыва
```

В примере шаблона промпта есть только одна метка-заполнитель. Вообще говоря, разные элементы промпта могут варьироваться в зависимости от входных данных. В этом случае мы добавляем метки-заполнители для каждого элемента и при создании промптов заменяем их все.

На рис. 1.2 показано использование шаблонов промптов при анализе данных непосредственно с помощью языковых моделей. Для каждого элемента данных (например, отзыва, который нужно классифицировать) мы заменяем метки--заполнители в шаблоне, чтобы сгенерировать промпт (можно также сказать, что мы *создаем экземпляр* промпта). Затем мы отправляем этот промпт в языковую модель, чтобы она решила интересующую нас задачу анализа данных.

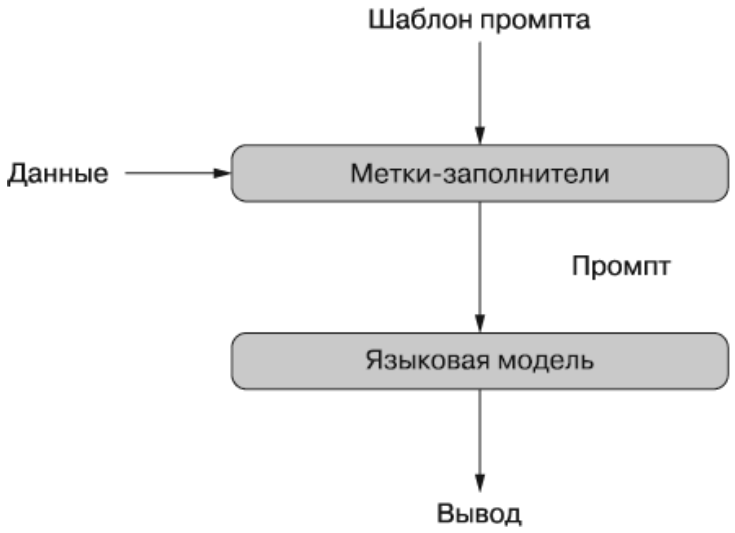


Рис. 1.2. Использование языковых моделей непосредственно для анализа данных. В шаблоне промпта описывается задача анализа. Он содержит метки-заполнители, которые заменяются данными для анализа. Когда замена выполнена, получившийся промпт передается в языковую модель, чтобы та могла создать вывод

1.4.2. Анализ данных с помощью внешних инструментов

Ввод данных непосредственно в шаблон не всегда самый эффективный вариант. Для работы с некоторыми типами данных предусмотрены специализированные инструменты. В таких случаях языковую модель зачастую лучше использовать для написания кода для обработки данных, чем непосредственно для анализа данных. Затем код, сгенерированный языковой моделью, можно выполнить с помощью специализированного инструмента.

Мы применим этот подход к структурированным данным. Для обработки табличных и графовых данных существуют специализированные инструменты, поддерживающие широкий спектр операций анализа. Такие операции, как фильтрация и агрегация данных, могут выполняться очень эффективно именно на структурированных данных. Даже если бы те же операции можно было надежно выполнять с помощью языковых моделей (а это не так), мы бы не стали их применять, поскольку оплата поставщикам наподобие компании OpenAI пропорциональна размеру входных данных. Обрабатывать большие структурированные наборы данных (например, таблицы с миллионами строк) с помощью языковых моделей непомерно дорого. В следующих главах мы рассмотрим несколько типов инструментов для обработки структурированных данных.

- *Реляционная система управления базами данных (РСУБД)* — хранит и обрабатывает реляционные данные, то есть коллекции таблиц с данными. Большинство РСУБД

поддерживают язык структурированных запросов SQL. Мы применим языковые модели для перевода вопросов о данных в запросы на языке SQL.

- *Графовая СУБД* — работает с графовыми данными, представляющими сущности и отношения между ними. Различные системы управления графовыми данными поддерживают разные языки запросов. В главе 5 мы рассмотрим, как с помощью языковых моделей преобразовать вопросы о данных в запросы на языке *Cypher*, который поддерживается Neo4j — графовой СУБД.

Для примера предположим, что нам необходимо предоставить непрофессиональным пользователям возможность анализировать реляционную базу данных, то есть набор таблиц с данными. Возможно, таблица содержит результаты опроса и требуется обеспечить пользователям возможность агрегировать ответы различных групп респондентов. Результаты опроса хранятся в реляционной системе управления базами данных (наиболее подходящем инструменте для этого типа данных). Используя языковые модели, можно позволить пользователям задавать вопросы о данных на естественном языке (то есть на простом русском). Языковая модель позаботится о переводе этих вопросов в формальные запросы. Точнее, учитывая, что данные хранятся в реляционной системе управления базами данных, вопросы нужно будет преобразовать в SQL-запросы.

Чтобы решить эту задачу, снова применим шаблон промпта. В данном случае нас интересует перевод текста в SQL, то есть языковая модель будет использоваться для перевода вопросов на естественном языке в SQL-запросы. Задача (перевод текста в SQL) и данные (база, содержащая результаты опроса) останутся

неизменными, но вопросы пользователя с течением времени будут меняться. Поэтому на месте вопроса пользователя мы ставим в шаблоне промпта метку-заполнитель. Теоретически приведенный ниже шаблон (листинг 1.3) должен позволить преобразовывать вопросы, касающиеся данных опроса, в SQL-запросы.

Листинг 1.3. Шаблон промпта для перевода вопросов на язык SQL

```
База данных:                    ❶ Описание базы данных
База данных содержит результаты опроса, хранящиеся в таблице SurveyResults,
которая включает следующие столбцы: ...
Вопрос:                         ❷ Вопрос, который требуется перевести
[Вопрос]
Переведи вопрос на язык SQL!    ❸ Описание задачи
```

Сначала в промпте описывается структура данных ❶. Это необходимо для того, чтобы система могла писать корректные запросы (например, такие, которые будут ссылаться на корректные имена таблиц и столбцы в этих таблицах). Описание в примере шаблона сокращено. О том, как точно описать структуру реляционной базы данных, мы поговорим в следующих главах. Далее в шаблоне содержится вопрос, который требуется перевести ❷. Это метка-заполнитель, позволяющая пользователям задавать разные вопросы, используя один и тот же шаблон промпта. Наконец, шаблон содержит (краткое) описание задачи ❸: нужно перевести вопросы в SQL-запросы!

На рис. 1.3 представлен процесс перевода текста на язык SQL. Получив соответствующий шаблон промпта, мы заменяем метку-заполнитель вопросом пользователя, переводим вопрос в SQL-запрос с помощью языковой модели и, наконец, выполняем запрос в реляционной системе управления базами данных. Результат запроса показывается пользователю.

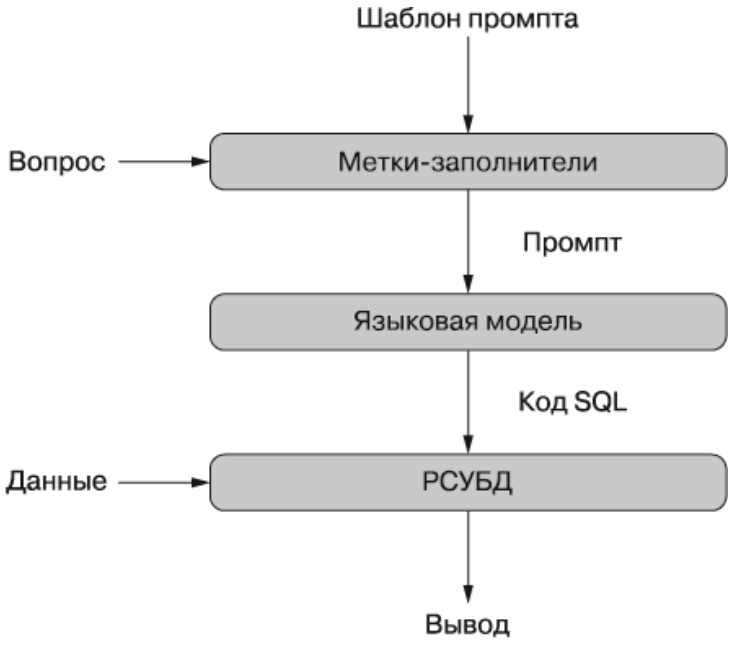


Рис. 1.3. Косвенное использование языковых моделей для создания интерфейса на естественном языке для табличных данных. Шаблон промпта содержит метки-заполнители для вопросов о данных. После того как они будут заменены вопросами, получившийся промпт используется в качестве входных данных для языковой модели. Она переводит вопрос в SQL-запрос, который выполняется через реляционную систему управления базами данных

1.5. Минимизация затрат

При обработке данных с помощью языковой модели мы обычно платим ее поставщику. Чем больше объем обрабатываемых данных, тем выше плата. Прежде чем анализировать большие объемы данных, мы должны убедиться, что не переплачиваем. Например, использование более крупных языковых моделей (другими словами, когда у сети, реализующей языковую модель, больше «нейронов») часто обходится дороже, но при решении сложных задач затраты могут окупиться за счет более качественных результатов. Однако если для успешного решения текущей задачи не нужна большая модель, то стоит сэкономить деньги и использовать модель меньшего размера. К счастью, есть немало способов

достичь оптимального компромисса между стоимостью обработки и качеством результатов. Далее мы кратко обсудим различные варианты, а подробнее поговорим о них в следующих главах.

1.5.1. Выбор оптимальной модели

Компания OpenAI предлагает множество различных версий модели GPT: от сравнительно небольших до гигантских, таких как GPT-4. На момент написания этой книги работа с GPT-4 в расчете на один входной токен обходилась более чем в 100 раз дороже использования самой дешевой версии.

> **Что такое токены**
>
> Плата за обработку данных языковыми моделями наподобие GPT-4 пропорциональна количеству лексем, считываемых и генерируемых моделью. *Токен* — это атомарная единица, в виде которой модель внутренне представляет текст. Обычно один токен соответствует примерно четырем символам[3].

При подобной разнице в цене нам стоит хорошенько подумать, какой именно модели достаточно для удовлетворения наших потребностей. Например, для такой простой задачи, как классификация отзывов, вряд ли понадобится самая дорогостоящая модель компании OpenAI. Но если мы решим написать с помощью модели сложный код для обработки данных, то использование самой дорогой модели вполне может окупиться.

Безусловно, не обязательно ограничиваться моделями, которые предлагает компания OpenAI. Языковые модели есть у многих поставщиков, таких как Google, Anthropic и Cohere. В принципе, можно даже создать собственную, используя модели, находящиеся в открытом доступе: например, на

платформе Hugging Face. Одни из них универсальные (наподобие моделей GPT компании OpenAI), а другие обучены для решения более конкретных задач. Если нас заинтересуют задачи, для которых существуют специализированные модели, то мы сможем использовать одну из них. Более подробно модели от других поставщиков описаны в главе 8.

Выбрать подходящую модель — задача не из легких. Прежде всего следует изучить данные эталонных тестов, например теста всесторонней оценки языковых моделей Стэнфордского университета[4] (Holistic Evaluation of Language Models, HELM, **https://crfm.stanford.edu/helm/**; рис. 1.4). Этот тест сравнивает качество результатов, полученных различными языковыми моделями при решении разных типов задач. В конечном счете, скорее всего, придется опробовать на задаче и выборке данных несколько моделей, чтобы убедиться, что выбрана оптимальная. В главе 9 мы рассмотрим, как систематически выполнять эталонное тестирование различных моделей на задаче-примере.

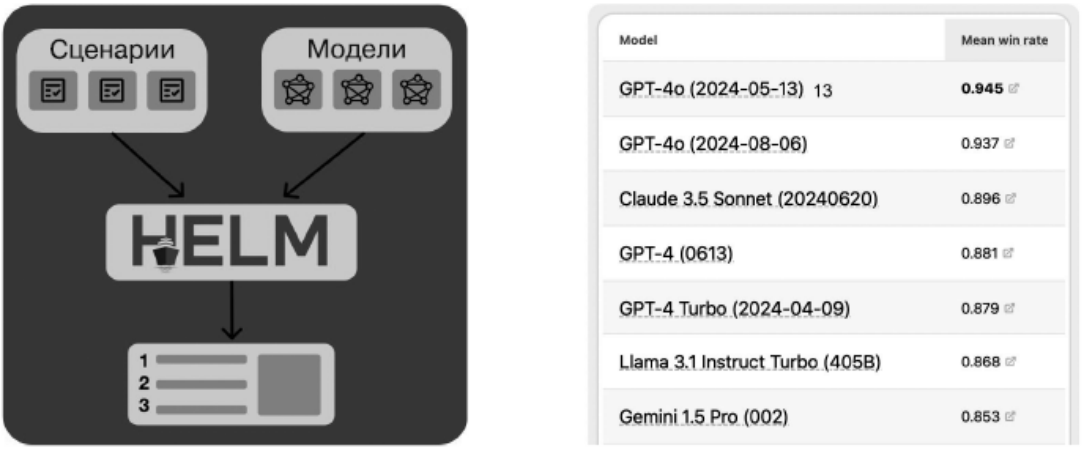


Рис. 1.4. Всесторонняя оценка языковых моделей: сравнение моделей, предлагаемых разными поставщиками, по различным метрикам

1.5.2. Оптимальная настройка моделей

Библиотека для Python от компании OpenAI предлагает множество параметров настройки, позволяющих управлять поведением модели. Например, можно задавать вероятность появления определенных слов в выводе модели. Это бывает полезно, скажем, при классификации отзывов. Если вывод модели должен содержать один из нескольких возможных вариантов (например, «положительный» и «отрицательный»), то имеет смысл ограничить возможные варианты вывода этими двумя. Это позволит избежать случаев генерации вывода, не соответствующего ни одному из названий классов. Другой пример — дообучение по критериям, используемым для принятия решения о том, когда модель должна прекращать генерацию вывода. Например, если мы знаем, что вывод должен состоять из одного токена (скажем, названия класса при классификации отзывов), то можно явно ограничить длину вывода одним токеном. Тогда модель не сможет сгенерировать больше выходных данных, чем необходимо (при этом мы экономим деньги, так как затраты зависят от длины генерируемого вывода).

Более подробно об этих и многих других параметрах настройки мы поговорим в главе 3, а в главе 9 рассмотрим варианты, при которых с помощью этих параметров можно повышать производительность языковых моделей.

Еще один вариант настройки моделей — их дообучение. По сути, это означает, что мы создаем собственный вариант существующей модели. Обучая модель на небольшом количестве обучающих данных, характерных для конкретной задачи, мы получаем модель, которая потенциально будет решать задачу лучше, чем исходная версия модели. Например, если требуется классифицировать отзывы, то можно обучить модель на нескольких сотнях примеров отзывов и соответствующих результатов классификации. Это позволит

использовать гораздо более компактную и дешевую дообученную модель, которая справляется с решением конкретной задачи так же хорошо, как и более крупная модель, не прошедшая дообучение.

Конечно, дообучение тоже стоит денег, и не всегда можно сразу понять, есть ли смысл применять его для конкретной задачи. Более подробно о дообучении и связанных с ним компромиссах мы поговорим в главе 9.

1.5.3. Промпт-инжиниринг

Шаблон промпта может существенно повлиять на качество результатов, генерируемых языковой моделью. Качественный шаблон четко определяет задачу, которую должна решить модель, и предоставляет весь необходимый контекст. В следующих главах мы рассмотрим, как подбирать нужный шаблон для разных задач, и охватим различные типы данных. После работы с этими примерами, применяя те же принципы, вы сможете разрабатывать собственные шаблоны промптов для новых задач.

Как и в случае с выбором модели, выбрать лучший шаблон промпта, не используя тестирование, бывает непросто. В главе 9 мы протестируем шаблоны в примерном сценарии и увидим, что разные шаблоны приводят к разным результатам. В некоторых случаях, потратив немного времени на поиск оптимального шаблона промпта, можно добиться удовлетворительного уровня производительности даже при использовании довольно дешевых моделей (тогда как работа с неоптимизированным шаблоном может потребовать применения более дорогостоящей модели).

Где взять шаблоны промптов

Поиск хорошего шаблона промпта для новой задачи может занять некоторое время. Если вам некогда, то пусть другие сделают это за вас! Точнее, в Интернете можно найти платформы, которые позволяют пользователям покупать и продавать шаблоны промптов. Одна из них — PromptBase (**https://promptbase.com**). Допустим, вы хотите перевести запросы на русском языке в SQL-запросы. Введя соответствующие ключевые слова, на этой платформе вы найдете не один, а несколько альтернативных шаблонов промптов. Если по описанию шаблон подходит, то его можно купить и использовать для анализа данных.

1.6. Продвинутые фреймворки и агенты

Мы в основном будем пользоваться библиотекой для Python от OpenAI и аналогичными библиотеками от других поставщиков. Например, эти библиотеки позволяют отправлять промпты языковым моделям и получать от них ответы. Во многих случаях таких библиотек бывает вполне достаточно, однако при разработке сложных приложений на основе языковых моделей стоит обратить внимание на более продвинутые фреймворки.

В книге мы рассмотрим два таких фреймворка для работы с языковыми моделями: LangChain (**https://langchain.com**) и LlamaIndex (**www.llamaindex.ai**). Оба облегчают разработку приложений на языке Python для анализа данных с использованием языковых моделей.

Помимо многих других возможностей, эти фреймворки позволяют легко создавать агенты, которые используют языковые модели. Такой подход удобен для решения сложных задач анализа данных, требующих, например, объединения

данных из нескольких источников. В основном в книге мы решаем задачи анализа данных с помощью одного обращения к языковой модели, будь то анализ текстового документа или преобразование вопроса о данных в формальный запрос. Если для решения задачи требуется совершить несколько действий, например выполнить предварительную обработку перед вызовом языковой модели или последующую обработку ответа модели, то в коде необходимо строго прописать соответствующую логику обработки.

Этот подход работает до тех пор, пока мы можем надежно предсказать последовательность действий, необходимых для обработки данных. Однако в некоторых случаях бывает трудно спрогнозировать требуемые действия. Например, пользователи могут задавать вопросы, касающиеся то текстового файла, то реляционной базы данных. И в зависимости от вопроса придется писать SQL-запрос или извлекать информацию из текстовых документов. Или, возможно, понадобится информация и из текста, и из реляционной базы данных — придется извлечь из текста относящуюся к вопросу информацию, а затем, используя ее, сформулировать SQL-запрос.

В таких случаях невозможно заранее строго прописать в коде все последовательности действий. Вместо этого стоит разработать подход, при котором модель будет достаточно гибкой и сможет сама решать, какое действие необходимо выполнить следующим. Это можно сделать с помощью агентов и языковых моделей. При таком подходе языковая модель используется для декомпозиции сложных задач анализа на более мелкие. Кроме того, модель способна вызывать *инструменты* (tools): произвольные функции, интерфейсы которых описаны на естественном языке. Такие инструменты могут, например, инкапсулировать выполнение SQL-запроса к

реляционной базе данных. После вызова соответствующего инструмента языковая модель получит доступ к результату вызова (например, к результату запроса) и сможет использовать этот результат для планирования следующих действий. Мы рассмотрим, как с помощью агентов можно решать сложные задачи анализа данных, когда изначально неясно, какие источники данных и методы обработки потребуются для их решения.

Резюме

- Языковые модели способны решать новые задачи, не проходя специальное обучение[5].
- Входными данными для языковой модели является промпт.
- Промпты могут сочетать в себе текст и другие типы данных, например изображения.
- Промпт содержит описание задачи, контекст и (не обязательно) примеры.
- Языковые модели могут анализировать некоторые типы данных напрямую.
- Для непосредственного анализа данных необходимо добавить эти данные в промпт.
- Шаблоны промптов содержат метки-заполнители: например, для представления элементов данных.
- Подставляя в шаблон промпта вместо меток-заполнителей информацию, мы получаем промпт.

- Языковые модели могут помочь анализировать данные с помощью внешних инструментов.
- Языковые модели могут управлять другими инструментами и указывать им, как обрабатывать данные.
- Модели могут иметь разный размер и существенно различаться по стоимости.
- Модели можно настраивать с помощью различных параметров конфигурации.
- Фреймворки LangChain и LlamaIndex помогают разрабатывать сложные приложения.
- Агенты используют языковые модели для решения сложных задач.

---

1 От лат. omnis — «всеобъемлющий», «универсальный». — *Примеч. пер.*

2 В задачах машинного обучения количество примеров обычно варьируется от 1 до 100. Для языковых моделей, из-за ограничений контекстного окна и роста стоимости запросов, чаще используется от 1 до 10 примеров. Частный случай с одним примером называется one-shot learning. — *Примеч. науч. ред.*

3 Модели, описанные в книге, используют токенизаторы, обученные преимущественно на англоязычных данных, из-за чего русские морфемы, суффиксы и окончания редко попадают в словарь модели. В результате текст на русском языке в среднем может разбиваться на большее число токенов. Это приводит к увеличению стоимости обработки русскоязычного текста по сравнению с английским. В более новых версиях моделей используются новые токенизаторы, которые уменьшают различие в числе токенов между языками, однако полностью не устраняют. — *Примеч. науч. ред.*

4 Для оценки моделей на русском языке можно воспользоваться следующим бенчмарком: https://mera.a-ai.ru/ru/text. — *Примеч. науч. ред.*

5 Имеются в виду задачи, ранее не встречавшиеся модели во время обучения. — *Примеч. науч. ред.*

2. Общение с ChatGPT

https://t.me/it_boooks/2

В этой главе

- Доступ к веб-интерфейсу ChatGPT.
- Использование ChatGPT для непосредственной обработки данных.
- Косвенное использование ChatGPT для обработки данных.

Пора познакомиться с ChatGPT! В этой главе мы будем работать с ChatGPT и начнем с анализа данных. Если вы никогда не пользовались ChatGPT, то здесь узнаете, как получить доступ к этому чат-боту, и составите первое впечатление о его возможностях (и ограничениях). Если вы уже пользовались им, но еще не применяли для анализа данных, то найдете в этой главе некоторые из множества способов использования ChatGPT в таком контексте.

Сначала мы обсудим веб-интерфейс, позволяющий получить доступ к ChatGPT от компании OpenAI. Мы рассмотрим процесс регистрации в OpenAI, обсудим основные функции, предлагаемые интерфейсом, и начнем в интерфейсе первый диалог с ChatGPT. Затем с помощью ChatGPT выполним анализ данных в нескольких примерах. Мы рассмотрим два способа использования ChatGPT для анализа данных: непосредственное использование и косвенное. При непосредственном использовании ChatGPT мы поручаем чат-боту выполнять фактическую обработку данных, предоставляя в качестве входных данных сами данные и описание задачи.

Такой подход эффективен для типов данных, которые ChatGPT обрабатывает по умолчанию (например, текстовые данные).

Однако ChatGPT можно использовать для анализа данных и косвенно. В этом случае ChatGPT просто служит переводчиком, переводя описания задач анализа на формальные языки, понятные внешним инструментам обработки данных. Фактическая обработка данных выполняется внешними инструментами. В этой главе вы увидите, что чат-бот ChatGPT эффективен в обоих сценариях.

2.1. Доступ к веб-интерфейсу

Откройте браузер и введите в адресную строку адрес **https://chat.openai.com/**. Вы создадите учетную запись OpenAI, которая позволит использовать ChatGPT. Если она у вас уже есть, то можете пропустить описанные ниже шаги, войти в свою учетную запись и перейти к следующему разделу.

Чтобы создать учетную запись, нажмите кнопку регистрации. В результате вы попадете на следующий экран (рис. 2.1).

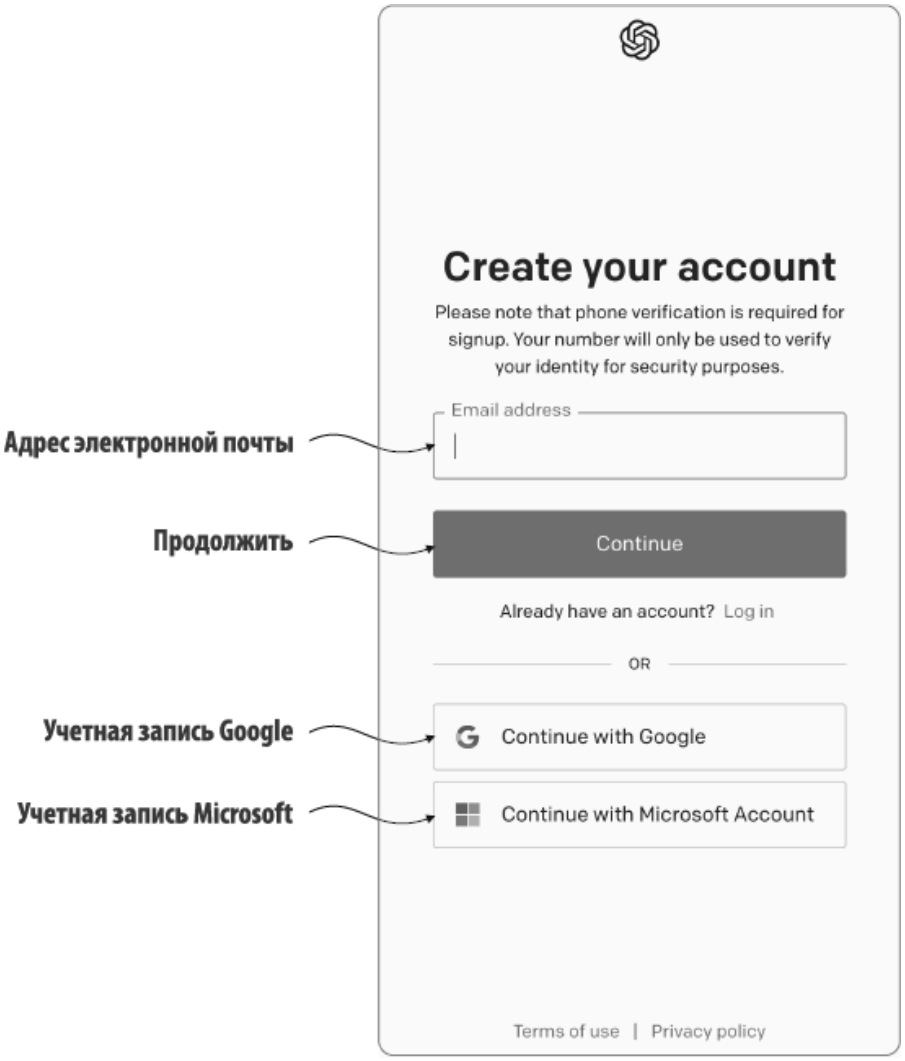


Рис. 2.1. Страница регистрации учетной записи OpenAI. Введите свой адрес электронной почты и нажмите кнопку Continue (Продолжить), чтобы продолжить работу, или зарегистрируйтесь, используя учетную запись Google или Microsoft

Существует несколько вариантов регистрации учетной записи OpenAI:

- с помощью учетной записи Google (нажав кнопку **Continue with Google** (Продолжить с Google));
- с помощью учетной записи Microsoft (нажав кнопку **Continue with Microsoft Account** (Продолжить с Microsoft));

- используя произвольный адрес электронной почты, введя его в поле **Email Address** (Адрес электронной почты) и нажав кнопку **Continue** (Продолжить). Далее следуйте инструкциям, которые появятся на экране.

Создав учетную запись любым из перечисленных способов, войдите в нее и выполните действия, описанные в следующем разделе.

> **Подписки OpenAI**
> Компания OpenAI предлагает различные типы учетных записей: одни из них бесплатные, а за другие взимается ежемесячная плата. Для приведенных ниже примеров достаточно бесплатной учетной записи. Вы можете выбрать платную подписку, чтобы получить доступ к большему количеству моделей и быстрее получать ответы от ChatGPT. В зависимости от того, какую подписку вы выберете, изображения на экране могут немного отличаться от снимков экранов, приведенных в этой главе.

2.2. Знакомство

Войдя в свою учетную запись OpenAI, вы должны увидеть интерфейс, показанный на рис. 2.2.

Этот интерфейс позволяет вести диалог с чат-ботом ChatGPT, вводя текст и получая ответы. Рассмотрим самые важные компоненты интерфейса.

Прежде всего здесь есть несколько примеров ввода, созданных по умолчанию. Можно просто нажать любой из них, чтобы начать чат.

Помимо этих примеров, можно ввести в текстовое поле произвольный текст. Далее в главе мы будем называть этот элемент интерфейса *полем ввода*. Наконец, в любой момент можно начать новый чат, нажав кнопку **New chat** (Новый чат). При этом из памяти ChatGPT стирается вся история диалога.

Какую модель выбрать

В списке **Model** (Модель) (см. рис. 2.2) можно выбрать одну из нескольких языковых моделей. ChatGPT поддерживает несколько моделей семейства GPT от компании OpenAI. Примеры, рассматриваемые ниже, должны работать с любой из доступных моделей. Можно протестировать несколько разных моделей, чтобы выяснить, как будут различаться результаты. В зависимости от того, какой вариант подписки вы выбрали, количество запросов, отправляемых конкретным моделям, может быть ограничено.

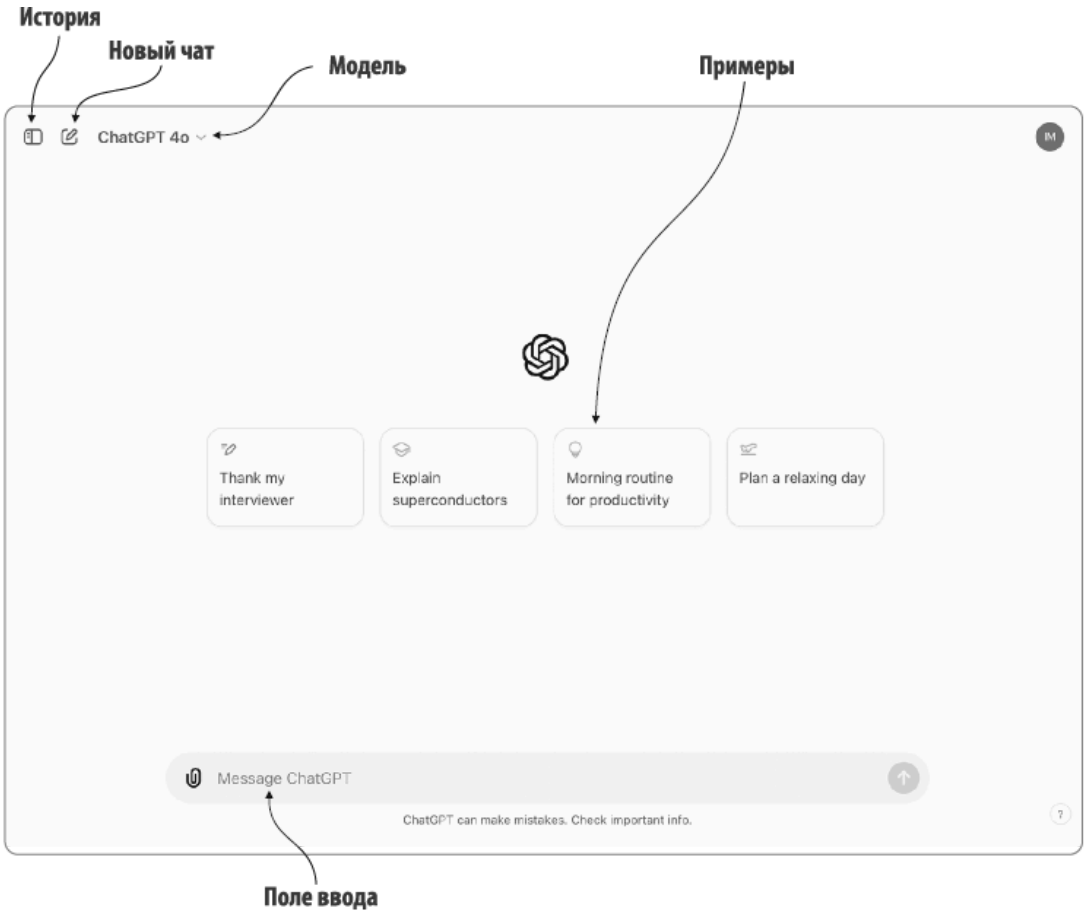


Рис. 2.2. Веб-интерфейс ChatGPT. Взаимодействуйте с ChatGPT, нажимая на готовые примеры вводных данных или вводя произвольный текст в поле ввода. Нажмите кнопку New chat (Новый чат), чтобы начать новую беседу

Пора поздороваться! Нажмите на поле ввода и поприветствуйте ChatGPT. Можно ввести какой угодно текст. Чат-бот ChatGPT был обучен вести диалоги с пользователями-людьми и должен быть в состоянии дать логичный ответ на большинство вводимых данных. Например, расскажите немного о себе. Или попросите у ChatGPT совета или помощи в решении предстоящих задач. Либо вы можете попросить написать стихотворение на выбранную вами тему. Возможно, вы захотите немного поболтать с ChatGPT, чтобы лучше понять его возможности и ограничения.

Обратите внимание, что ChatGPT может возвращаться к ранее введенным данным. Например, если вам не понравился

предыдущий ответ, то можете попросить ChatGPT исправить его или изменить. При этом повторять первоначальный запрос не нужно. Кроме того, если ответ вас не устраивает, но вы не хотите подробно разъяснять требования, то можете нажать кнопку **Regenerate** (Попробуй еще раз) (рис. 2.3).



Рис. 2.3. После того как чат-бот ChatGPT сгенерирует ответ, под ним появится кнопка Regenerate (Попробуй еще раз). Нажмите ее, чтобы получить другой ответ на последнее введенное сообщение

Нажав эту кнопку, вы заставите ChatGPT заново сгенерировать последний ответ. ChatGPT допускает определенную степень рандомизации при генерации результатов, поэтому велика вероятность, что второй результат будет отличаться (и, возможно, в лучшую сторону) от первого.

Как правило, весь вводимый вами текст является частью одного чата. Если в какой-то момент вы захотите начать новый чат (по сути, стерев «память» ChatGPT о предыдущей беседе), то просто нажмите кнопку **New chat** (Новый чат) (см. рис. 2.2). Обратите внимание, что предыдущие беседы не теряются, даже когда вы начинаете новую. Компания OpenAI хранит прошлые беседы и позволяет пользователям вернуться и изучить их (или продолжить беседу с того места, где они остановились в прошлый раз). Доступ к истории предыдущих бесед можно получить, нажав кнопку **History** (История) (см. рис. 2.2). Каждой беседе присваивается короткое название, которое генерируется автоматически на основе ее содержания.

Первое использование ChatGPT часто производит сильное впечатление. Он генерирует остроумные и логичные ответы на самые разные темы. Это может ввести в заблуждение

пользователей, которые начинают слишком доверять информации, предоставляемой ChatGPT. Важно помнить о различных ограничениях, свойственных современным языковым моделям. Всегда проверяйте результаты работы языковой модели, прежде чем использовать их.

Что такое галлюцинации

Термином «галлюцинация» в языковых моделях обозначают ситуации, в которых языковая модель при отсутствии информации выдумывает новое содержание и интегрирует его в свои ответы. Часто результат кажется убедительным и распознать галлюцинации бывает трудно. В современных исследованиях [1] рассматриваются методы, позволяющие снизить вероятность возникновения таких эпизодов. Однако на момент написания этой книги не существовало надежных стратегий, позволяющих избежать галлюцинаций. Поэтому не стоит слепо доверять информации, предоставляемой языковыми моделями, и всегда следует проверять и подтверждать ее, используя альтернативные источники.

2.3. Обработка текста с помощью ChatGPT

Начнем использовать ChatGPT для обработки данных! В этом разделе с помощью ChatGPT мы проанализируем распространенный тип неструктурированных данных — текст. Это «родной» формат ввода и вывода для ChatGPT, поэтому можно использовать чат для анализа текстовых данных напрямую, не привлекая внешние инструменты анализа. Это позволит сохранить простоту процесса, а именно это нам и нужно на первых этапах анализа данных с помощью ChatGPT.

Представьте ситуацию: вы сотрудник компании Banana, производителя бытовой электроники, в том числе ноутбуков и смартфонов. На сайте компании пользователи могут оставлять текстовые комментарии в свободной форме. Оказалось, что многие пользователи пишут в них о том, удовлетворены они или нет конкретными продуктами компании. Пользователи часто указывают на отдельные элементы этих продуктов, которые выделяются или требуют улучшения. В совокупности все эти отзывы должны быть очень полезны для принятия решений при разработке будущих продуктов Banana! Но, чтобы это стало возможным, сначала их нужно проанализировать и выявить общие моменты. Именно такое задание дает вам руководитель: подготовить статистику, которая обобщит отзывы о различных продуктах и категориях товаров. Кроме того, он хочет, чтобы в процессе чтения отзывов вы написали на них несколько вежливых ответов. В идеале вы хотите избежать необходимости тратить ближайшие недели на чтение и обобщение отзывов. Нельзя ли автоматизировать процесс? Возможно, ChatGPT сможет помочь.

Далее с помощью веб-интерфейса ChatGPT мы проанализируем образец отзыва. Если получится, то можно будет масштабировать процесс и автоматически анализировать все отзывы! Если вы уже запускали диалог, то сначала нажмите кнопку **New chat** (Новый чат) (см. рис. 2.2), чтобы начать новую беседу. Так вы будете уверены, что темы прошлых бесед с ChatGPT не повлияют на работу с примерами анализа, которые мы будем рассматривать далее.

Сначала расскажем ChatGPT об отзыве, который требуется проанализировать. Нажмите на поле ввода, введите следующий текст после метки **Вы**, а затем нажмите **Enter**.

Вы: Изучи этот отзыв о товаре.

«Я купил BananaBook год назад и очень им доволен! Экран потрясающий, и с него удобно читать даже при ярком солнечном свете. Он достаточно быстрый для всех операций, которые мне нужно на нем выполнять, например для написания кода, офисной работы. Единственный недостаток — это вес, хотелось бы, чтобы он был полегче, так как мне приходится часто носить его с собой. Но за такую цену это отличное предложение».

GPT: Судя по отзыву, BananaBook — это товар, который автор отзыва приобрел год назад и которым до сих пор доволен. Экран BananaBook впечатляет, позволяя пользователю легко читать содержимое даже при ярком солнечном свете. Кроме того, устройство достаточно быстрое для нужд пользователя, в том числе для написания кода и офисной работы. Однако автор отзыва отметил, что BananaBook немного тяжеловат и это может доставлять неудобства при переноске. В целом, по мнению автора отзыва, BananaBook — отличное решение за свою цену.

Текст после метки GPT — это ответ, сгенерированный языковой моделью ChatGPT во время пробной сессии. В данном случае чат-бот просто повторяет отзыв, хотя и немного его перефразирует, показывая тем самым, что он принял и «понял» ваши данные.

Почему результаты вывода разные

Чат-бот ChatGPT допускает определенную степень рандомизации при генерации результатов. Следовательно, вы, как правило, будете видеть разные результаты, даже если

дважды отправите абсолютно одинаковые данные. Поэтому вероятность, что будет выведен точно такой же результат, как показан здесь, — невелика. Однако, скорее всего, результат будет примерно такого же качества.

Начнем анализ текста с простой классификации, сопоставив отзыв с категорией. В частности, мы попробуем отнести отзыв к одной из трех категорий товаров: мобильные телефоны, персональные компьютеры и игровые приставки.

ПРЕДУПРЕЖДЕНИЕ *Не нажимайте* кнопку New chat (Новый чат) перед тем, как ввести следующее сообщение! Назначение этой кнопки — стереть из памяти ChatGPT историю предыдущего разговора. А нам это совершенно не нужно. Все последующие анализы будут относиться к тому же отзыву, который мы ввели вначале. А это возможно только в том случае, если ChatGPT будет его помнить.

Вы: Для какой категории товаров написан отзыв (мобильные телефоны, персональные компьютеры, игровые приставки).

GPT: Исходя из информации, представленной в отзыве, неясно, к какой категории товаров относится BananaBook. Автор отзыва не уточнил, является ли товар мобильным телефоном, персональным компьютером или игровой приставкой. Однако, судя по тому, что автор отзыва упомянул об использовании устройства для написания кода и офисной работы, вероятнее всего, BananaBook — это персональный компьютер или ноутбук, а не мобильный телефон или игровая консоль.

Неплохо! Чат-бот четко понял задачу и дал развернутый ответ. BananaBook — это вымышленное устройство, и не сразу понятно, о каком типе товара идет речь. Однако, как справедливо отмечает ChatGPT, написание кода и офисная работа чаще всего ассоциируются с персональными компьютерами (в большей степени, чем с мобильными телефонами или игровыми приставками). Несмотря на неопределенность, ChatGPT может дать логичный ответ и хорошо его обосновать. В данном случае ответ чат-бота даже более точен, чем требуется: ChatGPT не только относит отзыв к одной из трех представленных категорий, но и выдвигает гипотезу о том, что отзыв относится к ноутбуку (подкатегории персональных компьютеров). Это предположение кажется разумным в данном контексте, учитывая, что пользователь регулярно носит компьютер с собой (что обычно характерно для ноутбуков, а не для персональных компьютеров).

Предыдущий ответ логичен, но, пожалуй, длинноват. Фактически его длина почти такая же, как у исходного отзыва. Нет смысла использовать ChatGPT для анализа текста, если полученный ответ будет таким же длинным, как изначальный текст. Попросим ChatGPT писать более лаконично. Нам нужно не подробное обоснование, а только название наиболее вероятной категории, к которой относится отзыв.

Вы: Отвечай лаконично, напиши только название наиболее вероятной категории товара (мобильные телефоны, персональные компьютеры, игровые приставки).

GPT: Персональные компьютеры.

Так гораздо лучше! Очевидно, что ChatGPT смог понять наши инструкции и соответствующим образом адаптировать

свой предыдущий ответ. Помимо того что лаконичный вывод читается быстрее, у него есть еще одно преимущество. Если в конечном счете нам потребуется проанализировать не один отзыв, а множество, то важно, чтобы результаты для всех были единообразными. Наличие единообразного вывода облегчает подсчет количества отзывов в каждой из трех категорий.

Классифицируем отзывы по категориям. На этот раз напрямую укажем желаемый формат вывода (обратите внимание, что уточнение «отвечай лаконично» отсылает к предыдущим инструкциям).

Вы: К какой категории товаров — продукты питания, косметика, электроника — относится отзыв? Отвечай лаконично.

GPT: Электроника.

Отлично! Категория указана верно, и формат вывода соответствует ожиданиям. Пока мы выяснили, что ChatGPT умеет классифицировать отзывы по различным критериям. Эта возможность может быть полезной, в частности, для формирования статистики путем подсчета количества отзывов в каждой категории. Но в идеале нам также нужно проанализировать содержание этих отзывов. Какие товары или категории товаров требуется улучшить в первую очередь? Чтобы определить это, необходимо знать, какими являются отзывы: положительными или отрицательными, то есть доволен пользователь товаром или нет. Это еще один способ классификации текстов отзывов. Однако теперь объектом классификации является не товар, о котором идет речь в отзыве, а содержание отзыва.

Вы: Покупатель доволен товаром или нет? Ответь «доволен» или «недоволен».

GPT: Доволен.

Да, все верно! Используя комбинацию описанных выше возможностей, мы могли бы с помощью ChatGPT уже сейчас подсчитать количество положительных и отрицательных отзывов для различных категорий товаров. Это даст нам информацию о том, какие категории товаров необходимо срочно улучшить, но данные могут быть недостаточно конкретными. Например, было бы интересно узнать, какими характеристиками товара пользователи довольны или недовольны больше всего. Попробуем извлечь эту информацию с помощью ChatGPT.

Вы: Какие достоинства товара упоминаются в отзыве? Оформи ответ в виде нумерованного списка, опиши каждый элемент максимум тремя словами.

GPT: 1. Впечатляющий экран.

2. Быстрая производительность.

3. Можно писать код.

4. Отличная цена.

Снова правильно! Чат-бот извлекает наиболее значимые преимущества товара, упомянутые в отзыве. Обратите внимание, что формат вывода снова соответствует

инструкциям: это нумерованный список и каждая запись содержит не более трех слов. А что насчет недостатков, упомянутых в отзыве?

Вы: Какие недостатки товара упоминаются в отзыве? Оформи ответ в виде нумерованного списка, опиши каждый элемент максимум тремя словами.

GPT: 1. Большой вес.

Действительно, единственным недостатком BananaBook, по мнению пользователя, является вес. Обратите внимание, что ChatGPT придерживается заданного формата вывода (нумерованный список), даже если результат содержит всего одну запись.

Классификация отзывов и извлечение конкретных достоинств и недостатков — это отличный способ подготовить данные для статистики. Однако при этом может потеряться информация: например, более подробное обоснование, почему пользователям нравятся или не нравятся определенные характеристики товара. Иногда может понадобиться прочитать выборку отзывов. Но чтобы не тратить время на чтение слишком длинных отзывов, было бы хорошо сначала обобщить их. Сделаем это с помощью ChatGPT.

Вы: Кратко перескажи отзыв в трех предложениях.

GPT: Автор отзыва приобрел электронное устройство BananaBook год назад и до сих пор им доволен. BananaBook имеет впечатляющий экран, который позволяет легко читать текст при ярком солнечном свете, и достаточно быстр для написания кода и офисной работы. Единственный недостаток —

большой вес, но за свою цену BananaBook — отличное приобретение.

В результате получился корректный краткий пересказ, в котором использовано ровно столько предложений, сколько указано в инструкции. Можно ли сделать его еще короче?

Вы: Резюмируй отзыв в одном предложении.

GPT: Автор отзыва доволен электронным устройством BananaBook из-за его впечатляющего экрана, быстрой работы и отличной цены, несмотря на то что устройство немного тяжеловато.

Да, этот вариант тоже работает! Действительно, после первоначального тестирования кажется, что чат-бот способен помочь автоматизировать различные задачи анализа. Наконец, выясним, можно ли его использовать для задач, выходящих за рамки простого анализа. Поддержание хороших отношений с клиентами приоритетно для компании Banana! И поэтому руководитель поручает вам по возможности писать ответы на комментарии, выдерживая их в дружеском тоне. На это тоже требуется время. Нельзя ли с помощью ChatGPT автоматизировать и этот этап? Давайте посмотрим…

Вы: Напиши в дружеском тоне ответ на отзыв от компании-производителя Banana Book!

GPT: Конечно, вот дружеский ответ на отзыв:

Уважаемый покупатель,

Мы рады, что вы все еще довольны своим электронным устройством BananaBook спустя год после покупки! Мы разработали BananaBook, стремясь предложить нашим пользователям впечатляющий экран и быструю производительность, и рады видеть, что эти характеристики отвечают вашим потребностям при написании кода и в офисной работе. Мы ценим ваше мнение о весе устройства и учтем этот момент при дальнейших улучшениях. Благодарим за выбор BananaBook и надеемся и дальше превосходить ваши ожидания!

С наилучшими пожеланиями,

[Название компании]

Очень хорошо! Чат-бот отвечает клиенту дружеским сообщением, ссылаясь на конкретные детали отзыва и делая это в соответствующей манере (подтверждая положительные моменты, упомянутые клиентом, и обещая исправить недостатки). Обратите внимание, что ChatGPT оставляет метку-заполнитель вместо названия компании (отмечено квадратными скобками). Мы еще не сообщали ChatGPT название нашей компании (Banana). Исправим это.

Вы: Название компании — Banana. Вставь его вместо метки-заполнителя в своем ответе.

GPT: Прошу прощения за ошибку в моем предыдущем ответе. Вот обновленный ответ с правильным названием компании:

Уважаемый покупатель,

Мы рады, что вы все еще довольны своим электронным устройством BananaBook спустя год после покупки! Мы разработали BananaBook, стремясь предложить нашим пользователям впечатляющий экран и быструю производительность, и рады видеть, что эти характеристики отвечают вашим потребностям при написании кода и в офисной работе. Мы ценим ваше мнение о весе устройства и учтем это при дальнейших улучшениях. Благодарим за выбор BananaBook и надеемся и дальше превосходить ваши ожидания!

С наилучшими пожеланиями, команда компании Banana

Хорошо, это работает! Основываясь на дополнительной информации, чат-бот смог подставить в предыдущий ответ название компании.

ПРЕДУПРЕЖДЕНИЕ Языковые модели наподобие ChatGPT могут иногда генерировать вредоносный контент, хотя в последних моделях вероятность таких случаев значительно снизилась. Рекомендуется проверять текст, сгенерированный ChatGPT, прежде чем размещать его на публичных форумах. Следовательно, автоматически генерировать ответы на отзывы клиентов без контроля со стороны человека нецелесообразно.

Мы убедились, что ChatGPT можно использовать для решения различных задач, касающихся обработки текстов. С его помощью мы выполняли категоризацию текста согласно заданным критериям и пользовательским классификациям (классификация по цели отзыва и по его содержанию). Затем мы использовали языковую модель для извлечения определенных фрагментов информации из текста и обобщения

документов (то есть отзывов). Наконец, с помощью ChatGPT мы создали текст ответа на отзыв покупателя. Во всех случаях чат-бот был способен следовать инструкциям, касающимся задачи и желаемого формата вывода. Если хотите, то попробуйте написать еще один отзыв и убедитесь, что ChatGPT по-прежнему может решить все эти задачи.

До недавнего времени для решения каждой из задач обработки текста, о которых говорилось в этом разделе, требовалась специализированная языковая модель. Последнее поколение моделей обладает широкими возможностями и может решать самые разные задачи на основе их описания на простом русском (и других естественных языках).

Обратите внимание, что мы обработали только один короткий отзыв. Здесь использование ChatGPT не дает значительных преимуществ. Мы могли бы классифицировать или обобщить отзыв вручную и сделали бы это гораздо быстрее, чем с помощью ChatGPT. Конечно, целью автоматизации является масштабирование обработки на большое количество (возможно, более длинных) отзывов. Если говорить о сотнях или тысячах отзывов, то анализ вручную займет гораздо больше времени, чем настройка ChatGPT для выполнения этой задачи. Однако мы не затрагивали еще один важный элемент: как предоставить ChatGPT все тексты отзывов? Можно ли просто скопировать весь набор отзывов и вставить его в веб-интерфейс ChatGPT?

Такой подход не сработает. Языковые модели обычно имеют ограничения по объему данных, поступающих на обработку одновременно. Следовательно, нужен механизм, который будет автоматически «подавать» в ChatGPT отдельные отзывы (или небольшие наборы отзывов) для обработки. Соответствующие подходы мы рассмотрим в следующих главах. А пока просто хотим убедиться, что ChatGPT в принципе можно использовать

для выполнения разнообразных задач анализа текстовых документов.

2.4. Обработка таблиц с помощью ChatGPT

В предыдущем разделе мы использовали чат-бот непосредственно для обработки данных, предоставляя ему сами данные и описание задачи. Такой подход целесообразен, если мы обрабатываем текст. Это «родной» формат ввода и вывода ChatGPT. Для обработки других типов данных гораздо эффективнее использовать специализированные инструменты в зависимости от типа данных. Возникает вопрос: если мы используем для обработки данных внешние инструменты, то какую пользу приносит ChatGPT? Существует множество инструментов для обработки данных, специализированных для различных типов данных, видов обработки, а также аппаратных или программных платформ. Для работы с такими инструментами пользователям приходится формулировать задачи анализа на формальных языках этих инструментов. Написание кода для анализа данных — трудоемкая задача даже для специалистов, не говоря уже о пользователях, не имеющих соответствующей подготовки в области ИТ. Именно в подобных ситуациях и можно применять языковые модели наподобие ChatGPT, поскольку они понимают как естественный язык, так и формальные языки, используемые инструментами анализа данных. Это означает, что ChatGPT можно задействовать в качестве своеобразного «переводчика» для перевода наших вопросов о данных, сформулированных простым русским языком, в код на различных языках, исполняемый внешними инструментами. Именно этим мы и займемся далее.

Вернемся к примеру с компанией Banana. Вы по-прежнему работаете в ней и успешно используете ChatGPT для анализа

всего многообразия отзывов, оставленных пользователями. Эту информацию логично отобразить в виде таблицы, где каждая строка соответствует одному отзыву, а столбцы представляют собой различные типы данных, извлеченных из отзыва. В табл. 2.1 показаны первые несколько строк.

Таблица 2.1. Пример таблицы с тремя столбцами: идентификатор отзыва, показатель удовлетворенности клиента и категория товара

Идентификатор отзыва	Доволен	Категория
1	1	Ноутбуки
2	0	Телефоны
2	1	Игры
…	…	…

Чтобы не усложнять пример, воспользуемся таблицей с небольшим количеством столбцов, соответствующих подмножеству анализов, описанных в предыдущем разделе. Первый столбец содержит идентификатор отзыва, второй — показатель, сообщающий, доволен ли автор отзыва товаром (1) или недоволен (0). В последнем столбце указывается категория. В рамках текущего примера мы рассматриваем только три категории: ноутбуки, телефоны и игры. В этой усеченной таблице исходные отзывы представлены в гораздо более сжатом виде. Однако полная таблица содержит множество строк (поскольку мы начали с большого количества отзывов), и чтение указанных в ней необработанных табличных данных мало что прояснит. В идеале необходимо агрегировать данные нетривиальными методами и продемонстрировать руководству только общие тенденции. Какие инструменты можно для этого использовать?

2.4.1. Обработка таблиц в веб-интерфейсе

Первый вариант — использовать веб-интерфейс компании OpenAI непосредственно для анализа табличных данных. Сначала нужно скачать пример таблицы с результатами анализа отзыва. Для этого зайдите на сайт книги (**www.dataanalysiswithllms.com**), перейдите в раздел **Resources** (Дополнительные материалы), откройте материалы к главе 2, найдите ссылку **Review Table** и скачайте соответствующий файл. В нем содержится таблица в формате `.csv` со структурой, показанной в табл. 2.1.

> **Что такое формат .csv**
>
> Расширение CSV означает «значения, разделенные запятыми». Оно обозначает особый формат, используемый для представления табличных данных. Каждая строка таблицы хранится в отдельной строке, а значения для разных столбцов в одной строке разделяются запятыми.

Чтобы проанализировать такие данные непосредственно в веб-интерфейсе чат-бота ChatGPT, их сначала нужно загрузить. Нажмите кнопку загрузки, как показано на рис. 2.4 (перед этим можно начать новый чат). Выберите команду **Upload From Computer** (Добавить фото и файлы), а затем — файл, который вы только что скачали (`reviews_table.csv`).

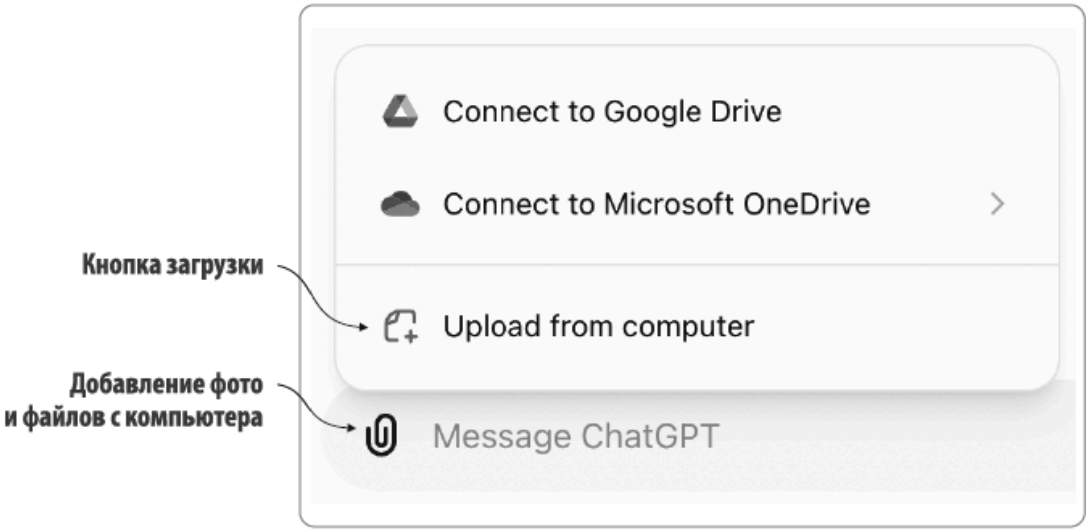


Рис. 2.4. Для загрузки файлов с диска нажмите кнопку загрузки и выберите команду Upload from computer (Загрузить с компьютера)

После загрузки файл должен появиться рядом с полем ввода текста. Теперь в это поле можно вводить какие угодно вопросы о данных. При генерации ответов ChatGPT будет учитывать и анализировать предоставленные вами данные. Например, спросим о количестве отзывов для каждой категории товаров.

> ***Пользователь:*** Сколько отзывов у нас есть для каждой категории товаров?

Отобразившийся результат должен быть таким же, как на рис. 2.5. ChatGPT создает таблицу, содержащую ответ на ваш вопрос (в каждой строке подсчитывается количество отзывов для одной категории товаров) и сопроводительный текст.

Как чат-бот вычислил ответ? Прочитал всю таблицу, чтобы сгенерировать ответ напрямую? Не совсем так. В фоновом режиме ChatGPT генерирует и выполняет код Python на платформе OpenAI, который анализирует входные данные (в данном сценарии механизм выполнения Python — это внешний инструмент, на который мы ссылались вначале). Сгенерированный код можно увидеть, нажав кнопку **Show**

analysis (Показать анализ) (см. рис. 2.5). На самом деле настоятельно рекомендуется проверить сгенерированный код — не стоит слепо верить ответу. В конце концов, несмотря на свои удивительные возможности, языковые модели регулярно допускают ошибки.

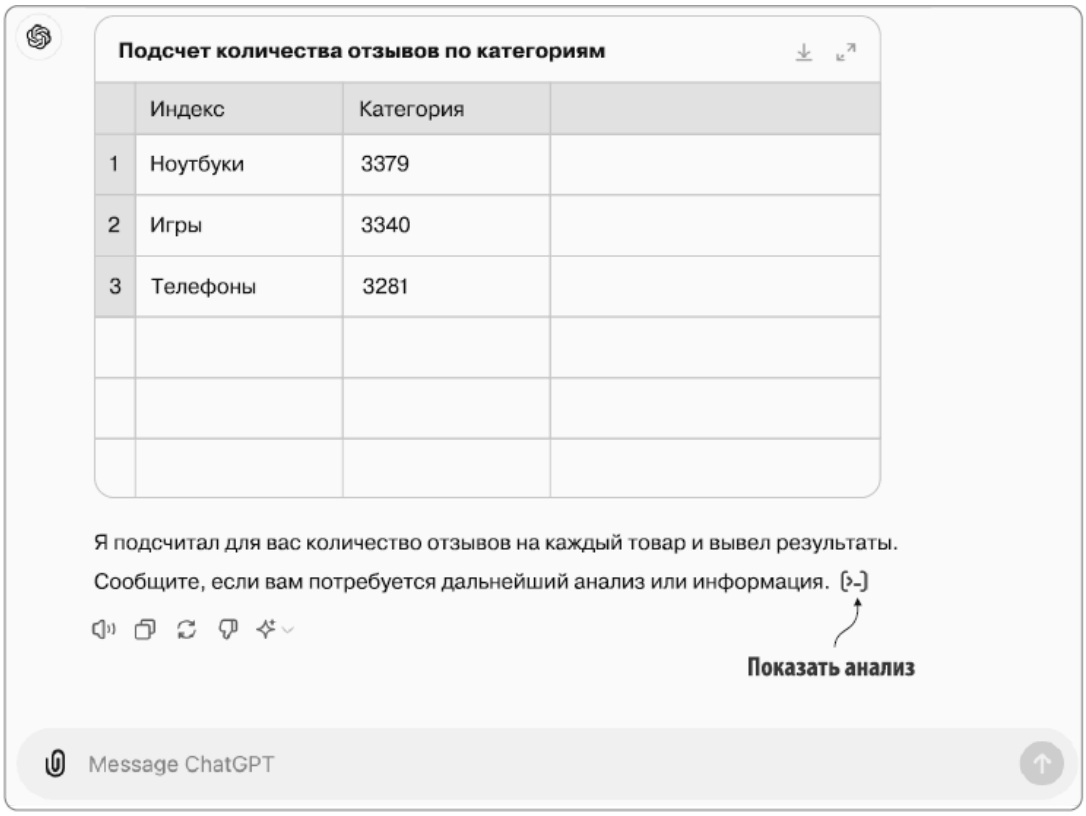


Рис. 2.5. Ответ, сгенерированный ChatGPT на вопрос о входной таблице. Нажмите кнопку Show analysis (Показать анализ), чтобы увидеть, как чат-бот составлял ответ

Задайте еще несколько вопросов, проверьте сгенерированный код и, пожалуй, даже протестируйте модель на еще нескольких разных наборах данных (вы можете загрузить на свой компьютер любые табличные данные, например в формате Excel). Вы увидите, что ChatGPT может обрабатывать самые разные наборы данных и запросы.

Вроде бы все работает отлично! Зачем делать что-то еще? Рассмотреть другие внешние инструменты для анализа данных

стоит по нескольким причинам.

- Начнем с того, что существуют строгие ограничения допустимого размера файлов для загрузки (512 Мбайт на момент написания книги). Загрузка больших наборов данных невозможна.

- Далее, вы могли заметить, что анализ данных с помощью Python может занимать несколько секунд даже для наборов данных умеренного размера (например, таблица с отзывами содержит всего 10 000 строк, что очень мало по современным стандартам). Анализ больших наборов данных занимает непозволительно много времени.

- И наконец, в некоторых сценариях использования загрузить данные в OpenAI не представляется возможным. Чтобы обеспечить конфиденциальность персональных данных, пользователи могут предпочесть анализировать их на собственных платформах.

В следующем подразделе мы рассмотрим варианты использования ChatGPT для анализа данных вне веб-интерфейса OpenAI.

2.4.2. Обработка таблиц на вашей платформе

В некоторых сценариях анализ данных непосредственно в веб-интерфейсе OpenAI невозможен. Вместо этого ChatGPT может помочь применить для анализа данных другие инструменты, которыми можно управлять напрямую. Далее мы задействуем *реляционную систему управления базами данных* (РСУБД) (relational database management system, RDBMS). Такие системы специализируются на работе с данными интересующего нас

типа и, как правило, демонстрируют высокую эффективность обработки. Это лишь пример: предлагаемый подход применим и к другим типам систем анализа данных.

Реляционные системы управления базами данных (РСУБД)

РСУБД специализируются на работе с реляционными данными, то есть наборами данных, содержащими одну или несколько таблиц наподобие приведенной выше (см. табл. 2.1). Большинство таких систем поддерживают варианты языка структурированных запросов (англ. Structured Query Language, SQL), используемого для описания данных и операций с ними.

В примерах ниже предполагается, что вы знакомы с основами SQL и РСУБД. Если это не так, то в главе 5 можете прочитать краткое введение в SQL.

Для получения более подробной информации рекомендую прочитать книгу *Database Management Systems* (авторы Герке и Рамакришнан) [2] или пройти мой онлайн-курс на сайте **www.databaselecture.com**.

Мы будем пользоваться SQLite — одной из наиболее популярных РСУБД. Чтобы не тратить время и силы на установку и настройку этой системы на своем компьютере, зайдите на сайт книги и перейдите по ссылке на ресурс BananaDB. Вы попадете в блокнот Google Colab, в котором можно выполнить описанные ниже действия. На рис. 2.6 показано, как должен выглядеть блокнот после того, как вы перейдете в него по ссылке.

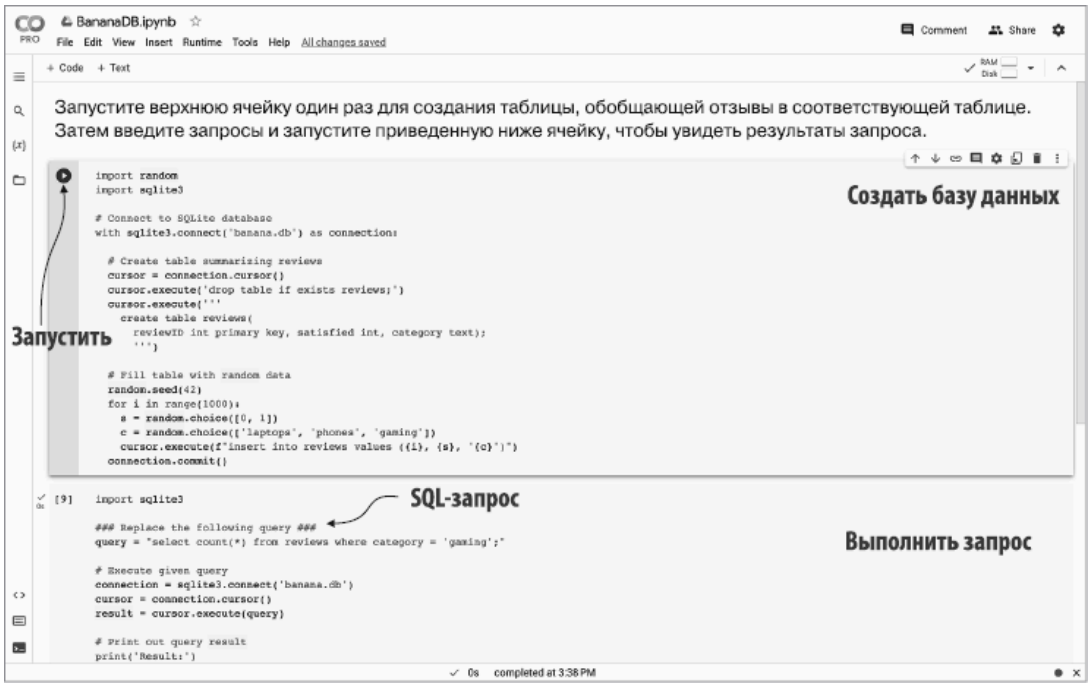


Рис. 2.6. Блокнот Google Colab, позволяющий запрашивать базу данных BananaDB через SQLite. Запустите верхнюю ячейку (Создать базу данных) для создания базы данных, замените заданный SQL-запрос другим запросом на свой выбор, а затем запустите нижнюю ячейку (Выполнить запрос), чтобы увидеть результат запроса. Стрелка указывает на кнопку запуска верхней ячейки

Блокноты Google Colab

Мы будем работать с блокнотами Google Colab, чтобы вы могли протестировать приведенные ниже примеры, не устанавливая программное обеспечение на свой компьютер. Далее в объяснениях предполагается, что вы в общих чертах знакомы с блокнотами Colab. Освежить навыки работы с ними поможет руководство, доступное на сайте **https://colab.research.google.com**.

Блокнот уже содержит код для создания примера базы и анализа данных в ней. Эта глава посвящена знакомству с веб-интерфейсом ChatGPT. Следовательно, в данный момент мы не будем тратить время на подробное обсуждение этого кода. Для работы с остальной частью главы вам достаточно знать, что код

делает и какие именно его фрагменты необходимо будет изменить. Если говорить упрощенно, то код в блокноте создает базу данных с данными примера и позволяет выполнять SQL-запросы, анализирующие эти данные через SQLite.

В блокноте представлены две ячейки, содержащие код (на рис. 2.6 обозначены как **Создать базу данных** и **Выполнить запрос**). В верхней ячейке (**Создать базу данных**) находится код, который создает базу, содержащую расширенную версию табл. 2.1, заполняя ее данными примерно 1000 отзывов. Сначала запустим эту ячейку, нажав кнопку **Run** (Запуск) в левом верхнем углу ячейки (на рис. 2.6 отмечена стрелкой). Через несколько секунд база данных SQLite должна быть создана и заполнена соответствующими данными. Эту ячейку необходимо запускать только один раз за сессию. Если не взаимодействовать с блокнотом в течение достаточно длительного периода времени, то среда будет сброшена и ячейку придется запускать снова.

Попросим ChatGPT написать для нас SQL-запросы, обращающиеся к этой базе данных. Эти запросы к данным можно выполнить с помощью нижней ячейки (см. рис. 2.6) (**Выполнить запрос**). По умолчанию ячейка содержит SQL-запрос, подсчитывающий количество отзывов в категории игр. Выполните его (нажав кнопку **Run** (Запуск), чтобы проверить, все ли работает. После запуска ячейки вы должны увидеть количество отзывов в категории игр (`gaming`): `342`. Замените текущий запрос (`select count(*) from reviews where category = 'gaming';`) на запросы, предложенные ChatGPT, а затем повторно запустите ячейку, чтобы получить результаты для новых запросов.

Итак, пришло время поручить чат-боту выполнить для нас анализ данных! Начните новый диалог, нажав кнопку **New Chat** (Новый чат) в веб-интерфейсе ChatGPT. Важно убедиться, что

предыдущие сообщения в окне диалога не помешают нашей новой задаче.

Чтобы написать запросы с помощью ChatGPT, сначала нужно рассказать чат-боту немного о том, как выглядит наша база данных. В частности, описать структуру таблицы, а именно указать имя таблицы отзывов (`Reviews`) и все связанные с ней столбцы. В данном случае в таблице три столбца, и, пожалуй, стоит добавить текст, описывающий их содержимое. При составлении SQL-запросов следует знать, какие типы данных находятся в каждом столбце, и в идеале иметь представление о возможных значениях и их семантике.

Наконец, еще один важный фрагмент информации — указание конкретной системы, использующейся для анализа данных. Несмотря на то что многие СУБД поддерживают SQL, а сам язык SQL достаточно стандартизирован, конкретный набор поддерживаемых функций SQL в разных системах может различаться. Кроме того, некоторые системы могут поддерживать синтаксис SQL, немного отклоняющийся от стандартного.

Теперь, когда мы выяснили, что важно для написания SQL-запросов, предоставим эту информацию ChatGPT для базы данных BananaDB, служащей в качестве примера.

Вы: У нас есть база данных SQLite 3 с таблицей (с именем `Reviews`), в которой обобщены сведения об отзывах на товары. В таблице три столбца:

- `ReviewID` (целочисленный идентификатор отзыва);
- `Satisfied`, обозначающий удовлетворенность пользователя (1, если автор отзыва доволен, и 0, если недоволен);

- `Category` для обозначения категории («ноутбуки», «телефоны» или «игры»).

GPT: Отлично! Наличие базы данных SQLite с таблицей отзывов `Reviews`, в которой обобщаются отзывы о товарах, может оказаться весьма полезным. Используя три упомянутых вами столбца (`ReviewID`, `Satisfied` и `Category`), можно хранить и извлекать ценную информацию об отзывах.

Чтобы помочь вам в дальнейшем, я предоставлю несколько примеров запросов в формате SQLite, которые можно использовать для работы с базой данных. Пожалуйста, сообщите мне, какие конкретные операции или информацию вы хотите выполнить или получить из базы данных.

Чат-бот ChatGPT с воодушевлением воспринимает перспективу анализа базы данных и выводит в ответе краткое описание структуры базы, которое мы предоставили в исходных данных. Кроме того, ChatGPT уже понимает, к чему все идет, и активно предлагает написать несколько примеров SQL-запросов, хотя мы еще об этом не просили. Посмотрим, что он предлагает.

Вы: Конечно! Пожалуйста, предоставь несколько примеров запросов!

GPT: Разумеется! Вот несколько примеров запросов, которые можно использовать для взаимодействия с таблицей отзывов базы данных в формате SQLite с именем `Reviews`:

Для получения всех отзывов:

```
SELECT * FROM Reviews;
```

Для получения положительных отзывов:

```
SELECT * FROM Reviews WHERE Satisfied = 1;
```

Для получения отрицательных отзывов:

```
SELECT * FROM Reviews WHERE Satisfied = 0;
```

Для получения отзывов о ноутбуках:

```
SELECT * FROM Reviews WHERE Category = 'laptops';
```

Для получения отзывов о телефонах:

```
SELECT * FROM Reviews WHERE Category = 'phones';
```

Для получения отзывов об играх:

```
SELECT * FROM Reviews WHERE Category = 'gaming';
```

Для подсчета общего количества отзывов:

```
SELECT COUNT(*) FROM Reviews;
```

Для подсчета количества положительных отзывов:

```
SELECT COUNT(*) FROM Reviews WHERE Satisfied = 1;
```

Для подсчета количества отрицательных отзывов:

```
SELECT COUNT(*) FROM Reviews WHERE Satisfied = 0;
```

Эти запросы должны послужить отправной точкой для выполнения различных операций с таблицей отзывов Reviews.

Их можно изменять или комбинировать в соответствии с вашими конкретными требованиями.

Неплохо! Запросы охватывают довольно большое количество естественных вопросов; например, о количестве отзывов, относящихся к определенным категориям. Вообще, попросить ChatGPT предложить несколько запросов — неплохой способ начать анализ набора данных.

Кстати, можно заметить, что веб-интерфейс отображает SQL-запросы с помощью специфического представления. На рис. 2.7 показано, как отображается первый запрос.



Рис. 2.7. В веб-интерфейсе ChatGPT для представления кода, в том числе SQL-запросов, используется особая разметка. Скопируйте код, нажав соответствующую кнопку (что позволит вставить этот код в нужный интерфейс для выполнения)

SQL-запрос находится в рамочке, благодаря этому его можно легко скопировать, нажав кнопку **Copy** (Копировать).

Чтобы протестировать какой-либо из предложенных запросов, скопируйте его (как описано выше), а затем вставьте в соответствующую позицию в блокноте BananaDB (метка **SQL-запрос** на рис. 2.6). В завершение запустите эту ячейку в блокноте — и полу́чите результат запроса.

Посмотрим, удастся ли нам заставить GPT перевести подобным образом наши вопросы в запросы.

Вы: Напиши SQL-запрос для получения категории товаров с наибольшим количеством отзывов!

GPT: Чтобы получить категорию товаров с наибольшим количеством отзывов из вашей таблицы `Reviews` в базе данных SQLite, можно использовать следующий SQL-запрос:

```
SELECT Category, COUNT(*) AS ReviewCount
FROM Reviews
GROUP BY Category
ORDER BY ReviewCount DESC
LIMIT 1;
```

В запросе используется выражение `GROUP BY` для группировки отзывов по столбцу `Category`. Затем применяется функция `COUNT(*)` для подсчета количества отзывов в каждой категории. Результаты упорядочиваются по убыванию на основе количества отзывов с помощью выражения `ORDER BY`. Наконец, выражение `LIMIT 1` гарантирует, что будет возвращена только категория с наибольшим количеством отзывов.

Запрос вернет категорию товаров вместе с соответствующим количеством отзывов для этой категории.

Нас интересует, для какой категории товаров собрано больше всего отзывов. ChatGPT переводит наш вопрос в SQL-запрос, который извлекает соответствующий результат. При выполнении в нашем примере базы данных этот запрос должен извлечь категорию `Gaming`, а также количество связанных с ней отзывов.

Как работать с многострочными запросами

Чат-бот ChatGPT сгенерировал первый многострочный SQL-запрос. Чтобы использовать его в блокноте Colab, заключите запрос в тройные кавычки (`"""`), а не в одинарные. Это гарантирует, что весь запрос будет обработан как одна строка.

Строго говоря, мы просили указать только категорию, а не связанное с ней количество. Здесь ChatGPT немного своевольничает, добавляя количество к результатам запроса. В данном случае это может быть приемлемо и даже интересно. В общем случае, используя ChatGPT для перевода вопросов в запросы, убедитесь, что чат-бот не расширяет ваши запросы недопустимым образом (например, при передаче в инструмент со строгими требованиями к формату ввода). Посмотрим, можно ли заставить ChatGPT точнее следовать инструкциям.

Вы: Результат запроса должен возвращать только категорию.

GPT: Извините за упущение. Если вы хотите, чтобы запрос возвращал только категорию с наибольшим количеством отзывов, то можете изменить предыдущий запрос, выбрав только столбец `Category`:

```
SELECT Category
FROM Reviews
GROUP BY Category
ORDER BY COUNT(*) DESC
LIMIT 1;
```

Обновленный запрос извлекает из таблицы отзывов `Reviews` категорию продукта с наибольшим количеством отзывов. Он группирует отзывы по категориям, упорядочивает их по

количеству в порядке убывания, а затем выбирает только верхний результат с помощью выражения `LIMIT 1`. Запрос возвращает категорию с наибольшим количеством отзывов.

Идеально! ChatGPT немного изменил предыдущий запрос, чтобы в выводе содержалась только категория без указания количества отзывов в ней. Можно заметить, что ответы чат-бота содержат запрос и объяснение, как он работает. Подобные объяснения полезны, поскольку позволяют проверить что запросы, сгенерированные ChatGPT, действительно корректны. Теперь немного углубим анализ и посмотрим, сколько авторов отзывов остались довольны своими играми.

Вы: Каково соотношение положительных отзывов в категории игр?

GPT: Рассчитать соотношение положительных отзывов в категории игр в вашей таблице отзывов `Reviews` базы данных SQLite можно с помощью следующего SQL-запроса:

```
SELECT (CAST(SUM(Satisfied) AS REAL) / COUNT(*)) AS PositiveRatio
FROM Reviews
WHERE Category = 'gaming';
```

Этот запрос фильтрует отзывы конкретно для категории игр (`gaming`) с помощью оператора `WHERE`. Затем с помощью функции `SUM()` он вычисляет сумму в столбце удовлетворенности (`Satisfied`), которая представляет количество положительных отзывов. Далее результат делится на общее количество отзывов в категории игр, полученное с

помощью функции COUNT(*). Функция CAST() используется для того, чтобы деление выполнялось с точностью до десятых долей. Запрос возвращает соотношение положительных отзывов (от довольных покупателей) в категории игр в виде десятичной дроби.

Выполнив запрос, скорее всего, вы увидите, что процент положительных ответов будет около 50 % (то есть соответствующий автор отзыва доволен)[6]. Наконец, посмотрим, какую категорию товаров необходимо улучшить в первую очередь.

Вы: У какой категории товаров наименьший процент положительных отзывов?

GPT: Для определения категории товаров с наименьшим процентом положительных отзывов из вашей таблицы отзывов Reviews базы данных SQLite можно выполнить следующий SQL-запрос:

```
SELECT Category, (CAST(SUM(Satisfied) AS REAL)
/ COUNT(*))
AS PositiveRatio
FROM Reviews
GROUP BY Category
ORDER BY PositiveRatio ASC
LIMIT 1;
```

В этом запросе для группировки отзывов по столбцу категорий (Category) используется оператор GROUP BY. Затем он вычисляет соотношение положительных отзывов для каждой

категории, разделив сумму их значений (`Satisfied`) на общее количество отзывов в этой категории. Результаты упорядочиваются в порядке возрастания на основе соотношения положительных отзывов с помощью оператора `ORDER BY`. Наконец, выражение `LIMIT 1` гарантирует, что будет возвращена только категория с наименьшим соотношением положительных отзывов.

Запрос возвращает название категории товаров вместе с соответствующим соотношением положительных отзывов (где автор доволен). Будет представлена категория с наименьшим соотношением положительных отзывов.

При выполнении этого запроса должна быть выведена категория `Phones` вместе с соответствующим количеством положительных отзывов.

В этом разделе мы разобрали второй способ использования ChatGPT для анализа данных: вместо того чтобы применять ChatGPT непосредственно к данным, мы используем его как механизм перевода. В этом случае мы переводим вопросы о данных (сформулированные на обычном русском языке) в SQL-запросы, которые затем можно выполнить с помощью соответствующего инструмента (в этом сценарии — СУБД). В следующих главах мы также будем использовать ChatGPT для перевода на другие формальные языки.

Мы вводили вопросы вручную, а затем копировали их в другой интерфейс. Очевидно, что этот подход не самый удобный. В главе 5 вы узнаете, как создавать платформы, автоматизирующие все эти действия.

Резюме

- Языковые модели могут решать различные задачи, получая инструкции на естественном языке.
- Несмотря на свои широкие возможности, языковые модели могут выдавать неверные результаты.
- Вы можете работать с чат-ботом ChatGPT через веб-интерфейс в своем браузере.
- ChatGPT позволяет пользователям взаимодействовать с различными версиями моделей GPT.
- Для обработки текста с помощью ChatGPT введите текст и описание задачи.
- ChatGPT умеет классифицировать текст по заданным категориям, извлекать информацию из текста по соответствующим вопросам, а также кратко излагать текст в соответствии с указаниями пользователя.
- ChatGPT не подходит для прямой обработки больших наборов данных. Однако модель способна генерировать код, который обрабатывает большие данные с помощью других инструментов. Для создания кода введите описание данных, задачи и формата вывода.

Литература

1. *Ji Z., Lee N., Frieske R. et al.* Survey of Hallucination in Natural Language Generation // ACM Computing Surveys, 2023. — № 55 (12). — Pp. 1–38.

2. *Ramakrishnan R., Gehrke J.* Database Management Systems. 3rd ed. — New York: McGraw Hill, 2002.

---

6 Неудивительно, учитывая, что наши тестовые данные сгенерированы с использованием равномерного случайного распределения (то есть количество положительных и отрицательных отзывов должно быть примерно одинаковым).

Часть II. Анализ данных с помощью языковых моделей

Для работы с языковыми моделями на малых объемах данных достаточно использовать веб-интерфейс. Но что делать, если данных много?

В главе 3 мы рассмотрим мощную библиотеку для Python от OpenAI, которая позволяет вызывать языковые модели напрямую из кода Python. Мы обсудим самые важные функции библиотеки, в том числе различные параметры, с помощью которых можно дообучать языковую модель под ваш конкретный сценарий использования.

В главе 4 мы обсудим несколько мини-проектов, в которых с помощью языковых моделей OpenAI и библиотеки для Python выполняется анализ коллекций текстовых документов. В частности, проекты охватывают классификацию текстовых документов по нескольким предопределенным категориям, извлечение конкретных фрагментов информации из текста и кластеризацию похожих текстовых документов.

В главе 5 показано, как языковые модели помогают в анализе структурированных данных: табличных и графовых. Здесь языковые модели используются как интерфейс, переводящий вопросы о данных (написанные на естественном языке) в формальные запросы, которые можно обработать специализированными инструментами анализа. Например, в главе демонстрируется, как с помощью языковых моделей можно создавать интерфейсы, переводящие вопросы в запросы на языке SQL, которые могут обрабатываться с помощью реляционных систем управления базами данных.

В главе 6 вы увидите, как языковые модели справляются с анализом изображений и даже видеороликов. Среди примеров

проектов — использование языковых моделей для ответа на произвольные вопросы об изображениях, составленные на естественном языке, а также для поиска пар изображений, на которых запечатлен один и тот же человек. Кроме того, в главе показано, как языковые модели позволяют автоматически создавать подписи к видео на основе их содержания.

Глава 7 посвящена аудиоданным: в ней рассматриваются распознавание и генерация речи. Например, вы увидите, как использовать модели OpenAI для создания интерфейса голосовых запросов, отвечающего на голосовые вопросы о табличных данных. Кроме того, мы создадим инструмент для перевода голосового ввода на одном языке в голосовой вывод на другом языке в соответствии с указаниями пользователя.

3. Библиотека OpenAI для Python

В этой главе

- Установка библиотеки OpenAI.
- Вызов моделей GPT с помощью Python.
- Параметры конфигурации.

В предыдущей главе мы работали с моделями GPT через веб-интерфейс компании OpenAI. Такой способ эффективен, если мы просто пытаемся вести диалог с чат-ботом или классифицировать и кратко излагать содержание отдельных отзывов. Однако представьте, что требуется классифицировать сотни отзывов. В этом случае использовать веб-интерфейс вручную для каждого отзыва очень трудоемко (мягко говоря).

Кроме того, возможно, нам потребуется применить языковую модель в сочетании с другими инструментами. Например, нужно будет с помощью модели GPT перевести вопросы в формальные запросы, а затем беспрепятственно запустить их выполнение в соответствующем инструменте (чтобы не копировать вручную запросы из одного интерфейса в другой). Для всех подобных сценариев нам потребуется другой интерфейс.

В этой главе мы обсудим библиотеку для Python от компании OpenAI, позволяющую вызывать языковые модели OpenAI напрямую из кода Python. Благодаря этому вызовы языковых моделей можно интегрировать в свой код как подфункцию. Мы будем работать с этой библиотекой в

большинстве глав книги. Поэтому рекомендую прочесть эту главу (хотя бы бегло), прежде чем переходить к следующим.

Глава посвящена библиотеке для Python от OpenAI, но библиотеки, предлагаемые другими поставщиками языковых моделей (такими как Anthropic, Cohere и Google), схожи с ней.

3.1. Предварительные требования

Сначала вам нужно убедиться, что у вас есть среда, подходящая для работы с библиотекой Python от компании OpenAI. Мы будем использовать язык программирования Python, поэтому убедитесь, что он установлен. Для этого откройте терминал и введите следующую команду (она должна работать в терминалах Linux, macOS и Windows):

```
python --version
```

Если после ввода команды появится сообщение об ошибке, то попробуйте заменить в ней `python` на `python3` и запустить снова. Если Python в вашей системе установлен, то в ответе должен отобразиться номер версии (например, Python 3.10.13). В противном случае будет выведено сообщение об ошибке. Для следующих примеров вам потребуется как минимум версия Python 3.9 (или более поздняя). Если Python не установлен в вашей системе или его версия ниже требуемой, то перейдите на сайт **www.python.org**, нажмите кнопку **Downloads** (Скачивания) и следуйте инструкциям по установке Python. Возможно, вы также захотите установить интегрированную среду разработки (integrated development environment, IDE). PyDev (**www.pydev.org**) и PyCharm (**www.jetbrains.com/pycharm**) — две из множества IDE, доступных для Python.

Наряду с самим языком Python вам понадобится pip — система управления пакетами, используемая для установки

пакетов Python (библиотека OpenAI поставляется в форме такого пакета). В последних версиях языка (которые вам в любом случае понадобятся) эта программа установлена по умолчанию. Тем не менее не помешает убедиться:

```
pip --version
```

Опять же, если все установлено правильно, должен отобразиться номер версии. Давайте убедимся, что система pip обновлена. Следующая команда должна работать в операционных системах Linux, macOS и Windows:

```
python -m pip install--upgrade pip
```

Вот и все! Ваша система готова к установке клиента OpenAI для Python.

Что делать, если что-то из вышеописанного не работает

Не паникуйте! Если какое-либо из перечисленных действий завершится неудачей, то, скорее всего, вам не удастся выполнить рассматриваемый далее код на своем локальном компьютере. Но если у вас есть доступ в Интернет, то вы можете использовать облачную платформу. Например, платформа Google Colab, доступная по адресу **https://colab.research.google.com**, позволяет создавать блокноты, в которых выполняются все примеры кода, которые будут приводиться ниже. На рис. 3.1 показан интерфейс после создания ячейки, устанавливающей библиотеку OpenAI (верхняя ячейка), и начала соответствующей программы на языке Python (нижняя ячейка). Установку и использование библиотеки мы обсудим в следующих разделах.

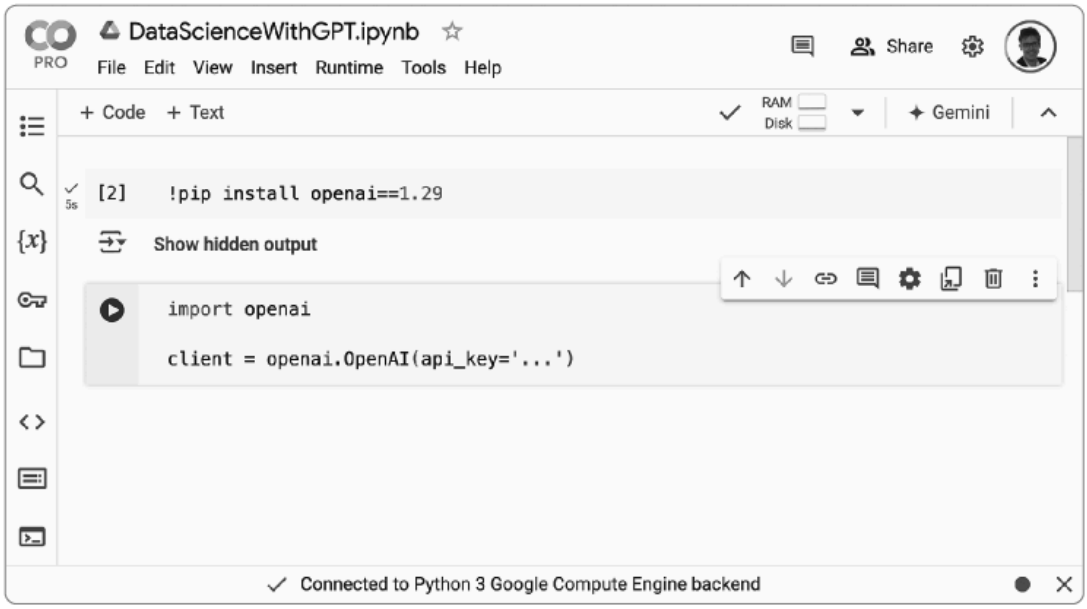


Рис. 3.1. Для запуска следующих примеров можно использовать платформу Google Colab

3.2. Установка библиотеки OpenAI для Python

Пора начинать профессионально работать с GPT! Веб-интерфейс модели ChatGPT, который был представлен в главе 2, полезен для ведения диалогов и тестирования новых запросов, однако непригоден для реализации сложных конвейеров обработки данных. Для этой цели гораздо лучше подходит библиотека Python от OpenAI, поскольку позволяет вызывать языковые модели напрямую из кода Python. Сначала установим эту библиотеку. Введите в терминале следующую команду:

```
pip install openai==1.29
```

Можно ли использовать другую версию библиотеки

Полагаю, вы заметили ссылку на конкретную версию (1.29) библиотеки от OpenAI. Код, представленный в текущей и следующих главах, был протестирован в этой версии. В

разных версиях библиотеки синтаксис немного различается, так что если вы не хотите адаптировать код, то установите именно эту версию.

При каждом использовании библиотеки OpenAI требуется предоставлять ключ, дающий доступ к моделям OpenAI (это необходимо для выставления счетов). Если вы еще не создали учетную запись OpenAI, то перейдите по адресу **https://platform.openai.com**, нажмите кнопку **Sign Up** (Зарегистрироваться) и следуйте инструкциям. Если у вас есть учетная запись, но вы в данный момент не вошли в систему, то введите свои учетные данные. Обязательно добавьте способ оплаты в разделе **Billing** (Платежи) и пополните счет несколькими долларами. Теперь самое время сгенерировать секретный ключ, если вы еще этого не сделали.

Перейдите по адресу **https://platform.openai.com/account/api-keys**. Должен отобразиться сайт, показанный на рис. 3.2.

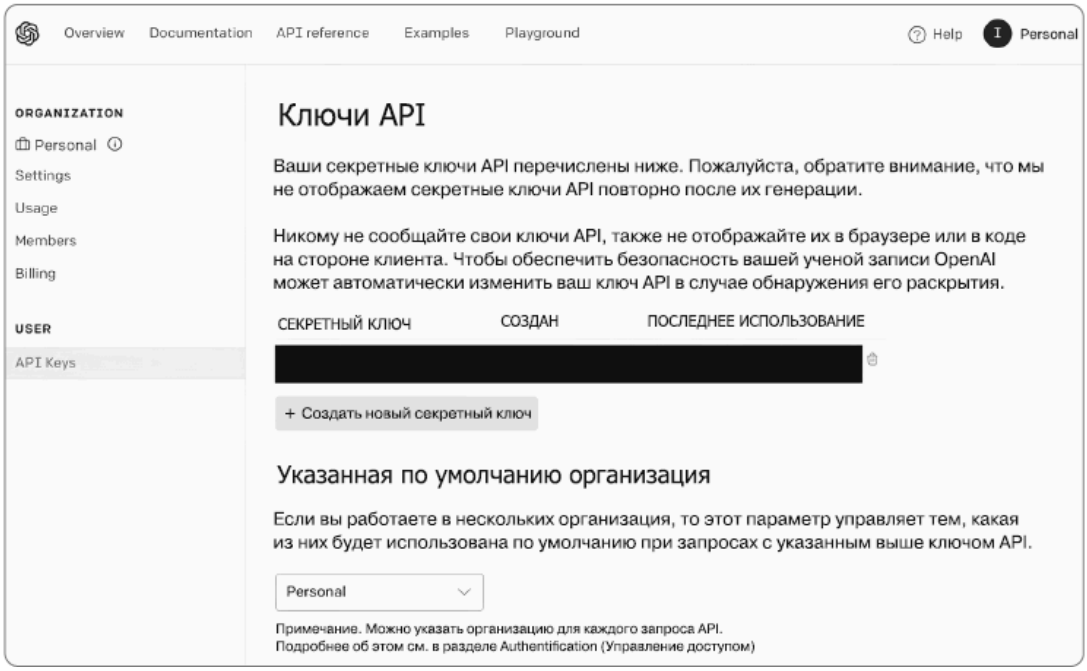


Рис. 3.2. Управление секретными ключами для доступа к учетной записи API OpenAI

Нажмите кнопку **Create New Secret Key** (Создать новый секретный ключ). В интерфейсе появится текстовая строка, представляющая собой ключ. Обязательно скопируйте и сохраните его! После закрытия этого окна вы не сможете повторно получить полный ключ.

Всякий раз при использовании библиотеки Python необходимо предоставлять секретный ключ, чтобы связать запросы с соответствующей учетной записью. Самый простой способ сделать это — сохранить секретный ключ в переменной окружения `OPENAI_API_KEY`. При наличии такой переменной OpenAI автоматически извлечет из нее ключ. Конкретная команда, используемая для установки переменных окружения, зависит от операционной системы. Например, следующая команда работает в операционных системах Linux и macOS (вместо трех точек укажите свой ключ):

```
export OPENAI_API_KEY=...
```

Другой способ — задавать ключ для каждого вызова, присваивая соответствующие значения переменной перед вызовами Python. Например, используйте такую команду, чтобы вызвать листинг кода, представленный в следующем разделе, одновременно задавая ключ (опять же, укажите вместо трех точек свой ключ):

```
OPENAI_API_KEY=... python listing1.py
```

Наконец, если никакой другой вариант на сработает, можно указать ключ доступа напрямую в коде Python. Точнее, сразу после импорта библиотеки Python от компании OpenAI можно передать ключ доступа к API как параметр при создании объекта клиента (который мы подробно обсудим позже):

```
import openai
```

```
client = openai.OpenAI(api_key='...')
```

Как и раньше, вместо трех точек укажите свой ключ доступа OpenAI. В приведенных ниже примерах кода предполагается, что ключ доступа указан в переменной окружения, поэтому в них этот параметр опущен. Если переменные окружения у вас не работают, то измените коды, передав ключ доступа как параметр.

ПРЕДУПРЕЖДЕНИЕ Никогда не публикуйте свой код, если он содержит ключ доступа OpenAI. Помимо прочего, наличие вашего ключа позволит другим вызывать модели OpenAI, а вам придется за это платить.

Итак, вы указали ключ доступа тем или иным способом. Теперь мы готовы начать вызывать модели GPT с помощью библиотеки OpenAI для Python.

3.3. Список доступных моделей

Обратимся к библиотеке Python для получения списка доступных моделей OpenAI. В листинге 3.1 показан соответствующий код на Python. (Этот код и все последующие доступны по ссылкам, размещенным на сайте книги, в разделе дополнительных материалов.)

Листинг 3.1. Список доступных моделей OpenAI

```
import openai                          ❶ Импортирует библиотеку OpenAI для Python
client = openai.OpenAI()               ❷ Создает клиент OpenAI

models = client.models.list()          ❸ Получает доступные модели OpenAI

for model in models.data:              ❹ Выводит полученные модели
    print(model)
```

Сначала импортируем библиотеку OpenAI ❶ . Затем создадим объект клиента, позволяющий обращаться к

функциям библиотеки ❷ . Далее запросим все доступные модели OpenAI ❸ и выведем результат ❹ , который должен быть похож на приведенный ниже:

```
Model(id='dall-e-3', created=1698785189,
    object='model', owned_by='system')
Model(id='whisper-1', created=1677532384,
    object='model', owned_by='openai-internal')
Model(id='GPT-4o-2024-05-13', created=1715368132,
    object='model', owned_by='system')
Model(id='davinci-002', created=1692634301,
    object='model', owned_by='system')
Model(id='GPT-4o', created=1715367049, ❶ Модель GPT-4o
    object='model', owned_by='system')
...
❷ Дообученная версия модели
Model(id='curie:ft-personal-2022-01-10-16-52-53',
    created=1641833573, object='model', owned_by='trummerlab')
Model(id='davinci:ft-personal-2022-01-13-19-59-51',
    created=1642103991, object='model', owned_by='trummerlab')
Model(id='ft:gpt-3.5-turbo-0613:trummerlab::8qlJH6bV',
    created=1707585607, object='model', owned_by='trummerlab')
...
```

Каждая модель имеет идентификатор (ID) (например, `GPT-4o` ❶). Мы будем использовать его, чтобы указывать OpenAI, какую модель использовать для обработки наших запросов. Помимо идентификатора, каждая модель содержит временную метку создания и информацию о владельце модели (поле `owned_by`). В большинстве случаев модели принадлежат компании OpenAI (помечены, например, как `system` или `openai-internal`). Однако иногда модели принадлежат `trummerlab` ❷ — это имя учетной записи, которую использует автор этой книги. Такие модели не являются общедоступными, они принадлежат владельцу учетной записи. Вы не увидите их, если будете выполнять код с использованием своей записи. Они создаются из общедоступных базовых моделей в ходе процесса, называемого *дообучением*.

Что такое дообучение, или тонкая настройка (fine-tuning)

По умолчанию языковые модели, такие как GPT-4o, обучаются быть универсальными, то есть способными выполнять любую

задачу (теоретически). Но иногда нам нужна не универсальная модель, а та, которая будет очень хорошо справляться с одной конкретной задачей. Дообучение позволяет специально настроить модель для решения интересующей нас задачи. Более подробно дообучение мы обсудим в главе 9.

3.4. Генерация ответов в диалоге

Почти везде в нашем коде используется одна и та же функция библиотеки Python от OpenAI: *генерация ответов в диалоге*, или *диалоговый режим* (chat completion). При этом модель генерирует ответ на основе предыдущего диалога. Входные данные могут быть различных типов, например текстом и изображениями. Мы используем эти возможности в следующих главах, но пока ограничимся текстом. Генерация ответов в диалоге выполняется и в фоновом режиме веб-интерфейса ChatGPT от OpenAI. При использовании в качестве входных данных истории диалога (в которую входит последнее сообщение, а также предыдущие, возможно содержащие релевантный контекст) модель генерирует наиболее подходящий ответ.

Чтобы использовать возможность генерации ответов в диалоге из кода Python, сначала вам потребуется формат для описания истории чата. Это часть входных данных, предоставляемых для общения с моделью. В библиотеке Python от OpenAI чаты представлены в виде списка сообщений. Каждое из них, в свою очередь, представлено в виде словаря Python, который задает значения для нескольких важных свойств сообщения. Для каждого сообщения необходимо указать как минимум два важных атрибута:

- `role`, определяющий источник сообщения;
- `content`, содержащий текст сообщения.

Начнем с обсуждения атрибута `role`. Как вы знаете из предыдущей главы, чат с моделями GPT — это обмен множественными сообщениями между пользователем и моделью. Соответственно, мы можем указать значение `user` для атрибута `role`, чтобы идентифицировать сообщение как написанное пользователем. Как вариант, можно указать значение `assistant`, чтобы пометить сообщение как сгенерированное языковой моделью. Третье возможное значение для атрибута `role` — `system`. Такие сообщения обычно используются в начале истории чата. Они предназначены для передачи общих указаний модели, независимо от конкретных задач, поставленных пользователями. Например, типичное системное сообщение может иметь содержание «Ты полезный помощник», но возможны и более специализированные варианты (например, «Ты — помощник, который переводит вопросы о наборах данных в SQL-запросы»). В этой книге системные сообщения не используются, но вы можете поэкспериментировать и добавить собственные системные сообщения, чтобы выяснить, повлияют ли они на вывод модели.

Атрибут `content` указывает содержание сообщения. В этой главе мы ограничимся текстовым содержанием, а в следующих обсудим, как использовать языковые модели для обработки более разнообразных типов контента. В приведенных ниже примерах кода нужно будет указать в истории чата только одно сообщение. Оно содержит инструкции, описывающие задачу, которую должна решить языковая модель, а также релевантную

контекстную информацию. Например, вот такая история чата побуждает модель сочинить для нас историю:

```
[{
    'role':'user',                  ❶ Сообщение от пользователя
    'content':'Расскажи историю!'   ❷ Спецификация задачи
}]
```

В списке сообщений содержится только одно сообщение. Оно помечено как исходящее от пользователя ❶ , и в его содержании описывается упомянутая ранее задача ❷ . В ответ мы ожидаем, что модель сгенерирует историю, следуя инструкциям.

Как вызвать модель для генерации ответов в диалоге? Для этого можно использовать всего несколько строк кода на Python. Прежде всего необходимо импортировать библиотеку Python от OpenAI ❶ и создать объект клиента ❷:

```
import openai                ❶ Импортирует библиотеку OpenAI для Python
client = openai.OpenAI()     ❷ Создает клиент OpenAI
```

Мы используем объект `client` для всех последующих вызовов языковой модели. Приведенный выше код есть почти во всех наших примерах кода. Помните, что при создании клиента, возможно, придется вручную передавать ключ доступа OpenAI как параметр (если вы не укажете свой ключ доступа в переменной окружения, как рекомендуется). После создания объекта `client` можно отправлять запросы на генерацию ответов в диалоге, как показано ниже:

```
result = client.chat.completions.create(
    model='GPT-4o',                 ❶ Выбирает модель
    messages=[{                     ❷ Указывает входные сообщения
        'role':'user',
        'content':'Расскажи историю!'
        }])
```

С помощью функции `client.chat.completions.create` создадим новый запрос. Параметр `model` ❶ содержит название модели, которую необходимо использовать для диалога. В нашем случае мы выбираем модель GPT-4o компании OpenAI,

способную обрабатывать мультимодальные данные. Мы будем использовать ее для большинства примеров кода в книге. Далее в качестве входных данных укажем историю чата, добавив параметр `messages` ❷. Это уже обсуждавшаяся ранее история чата, которая дает модели указание сочинить историю.

Теперь объединим все вместе. В листинге 3.2 модель GPT-4o используется для создания истории. (Файл с кодом доступен по ссылке **Listing 2**, которую можно найти на сайте книги, в разделе дополнительных материалов к этой главе.)

Листинг 3.2. Использование модели GPT-4o для генерации ответов в диалоге

```
import openai     ◄── Импортирует библиотеку OpenAI для Python
client = openai.OpenAI()  ◄┐ Создает клиент OpenAI

result = client.chat.completions.create(  ◄── Вызывает диалоговый режим
    model='GPT-4o',              ◄── Выбирает модель
    messages=[{          ◄┐ Определяет входные сообщения
        'role':'user',
        'content':'Расскажи историю!'
        }])
print(result)
```

В результате выполнения кода должен получиться ответ, подобный приведенному ниже (из-за случайного характера генерации ваша история может быть другой):

```
ChatCompletion(
    id='chatcmpl-9YKmJCE8SITsKyI557T8KTuX3IxWN',
    choices=[                                    ❶ Список вариантов ответов
        Choice(
            finish_reason='stop',                ❷ Условие остановки
            index=0,
            logprobs=None,
            message=ChatCompletionMessage(       ❸ Ответ модели
                content="Конечно! Вот история, которая... ",
                role='assistant',
                function_call=None,
                tool_calls=None)
            )
        ],
        created=1717970051,
        model='GPT-4o-2024-05-13',
        object='chat.completion',
        system_fingerprint='fp_319be4768e',
        usage=CompletionUsage(                   ❹ Использование токенов
            completion_tokens=810,
            prompt_tokens=12,
            total_tokens=822
        )
    )
```

Обсудим компоненты результата. Начнем с того, что у нас есть список альтернативных вариантов ответа ❶ (объекты `Choice`). В нашем случае в списке пока только одна запись. Такое поведение задано по умолчанию, хотя мы можем запросить несколько альтернативных вариантов, установив соответствующие параметры конфигурации (обсуждаются в следующем разделе). Параметр `finish_reason` ❷ указывает для каждого ответа условие остановки генерации ответа. Например, это может быть достижение лимита длины генерируемого текста. Значение `stop` указывает, что языковая модель смогла сгенерировать вывод полностью (а не достигла лимита длины). Фактическое содержание сообщения ❸ сокращено, и, скорее всего, вызвав код повторно, вы получите другую историю.

Помимо самих ответов, результат содержит метаданные и статистику использования ❹ . Точнее, мы находим значения для следующих свойств:

- `completion_tokens` — количество сгенерированных токенов;

- `prompt_tokens` — количество токенов во входных данных;
- `total_tokens` — количество прочитанных и сгенерированных токенов.

Почему нужно учитывать количество токенов? Потому что в тарифах большинства моделей OpenAI учитывается количество прочитанных и сгенерированных токенов. Например, на момент написания книги использование модели GPT-4o стоило 5 долларов за миллион прочитанных токенов и 15 — за миллион сгенерированных. Обратите внимание на разницу в цене. Обычно, как в нашем случае, генерация токенов обходится дороже, чем их чтение. Цена зависит не только от количества токенов, но и от используемой модели. Например, замена модели GPT-4o на GPT-3.5 Turbo (чуть менее мощную версию GPT) снижает затраты в десять раз. Прежде чем анализировать большие объемы данных с помощью языковых моделей, выберите размер модели, подходящий под вашу задачу и кошелек.

3.5. Настройка поведения модели

Влиять на то, как модель отвечает на ваши входные данные, можно с помощью разных параметров. Их можно указать в дополнение к параметрам `model` и `messages` при вызове функции `chat.completions.create`. В этом разделе мы обсудим различные категории параметров, классифицируя их в зависимости от того, на что именно в поведении модели они влияют.

3.5.1. Настройка условий остановки

При вызове в диалоговом режиме модель продолжит генерировать вывод до тех пор, пока не будет выполнено условие остановки. Управлять моментом остановки генерации текста позволяют два параметра: `max_tokens` и `stop`.

Параметр `max_tokens` указывает максимальное количество токенов (то есть атомарных единиц, с помощью которых языковые модели представляют текст), генерируемых во время диалога. Токен соответствует примерно четырем символам, а в абзаце, как правило, содержится около 100 токенов. Максимально допустимое значение для этого параметра определяется используемой моделью. Например, ada, одна из самых маленьких версий модели GPT, позволяет использовать до 2049 токенов, а GPT-4o поддерживает до 128 000. Имейте в виду, что в максимальное количество токенов, поддерживаемое моделью, входят как прочитанные, так и сгенерированные токены. Параметр `max_tokens` относится только к количеству сгенерированных токенов, поэтому не следует задавать его значение выше, чем максимальное количество токенов, поддерживаемое используемой моделью, *минус количество токенов в промпте*.

Как правило, почти всегда рекомендуется задавать для параметра `max_tokens` разумное значение. В конце концов, мы платим за каждый сгенерированный токен, и установка ограничения на их количество позволяет контролировать расходы на один вызов модели.

В некоторых сценариях определенные шаблонные выражения обозначают конец желаемого вывода. Например, при генерации кода это может быть строка, характерная для соответствующего языка программирования и указывающая на конец программы. С другой стороны, если модель сочиняет историю, это может быть строка «и жили они долго и счастливо!». В таких сценариях можно использовать параметр

stop, чтобы настроить библиотеку OpenAI на прекращение генерации вывода всякий раз при появлении определенной последовательности токенов. В некоторых случаях только одна последовательность токенов указывает на остановку. В таких сценариях можно напрямую присвоить параметру stop соответствующее строковое значение. В других сценариях есть несколько последовательностей, потенциально указывающих на остановку. Тогда параметру stop можно задать список, содержащий до четырех последовательностей. Генерация текста будет прекращаться при появлении любой из них.

Обратите внимание: параметры max_tokens и stop можно использовать вместе. В таких случаях генерация вывода будет прекращаться при достижении лимита длины или появлении одной из стоп-последовательностей (в зависимости от того, что произойдет раньше).

3.5.2. Настройка генерации вывода

Параметры, которые мы обсуждали в предыдущем подразделе, позволяют выбирать условие остановки вывода. Но как повлиять на результаты вывода, генерируемые до самого момента остановки? Компания OpenAI предоставляет несколько параметров, с помощью которых можно смещать вероятности выбора токенов при генерации текста моделями GPT.

Некоторые параметры позволяют влиять на то, насколько «повторяющимся» должен быть генерируемый вывод. Точнее, с помощью этих параметров можно указать, желательно ли многократное повторение одних и тех же токенов.

Параметр presence_penalty позволяет вам штрафовать модель за повторное использование в сгенерированном ответе одних и тех же токенов. Штраф может принимать значения в

диапазоне от `-2` до `+2` (со значением по умолчанию `0`). Положительное значение штрафа побуждает модель избегать повторного использования одних и тех же токенов. При отрицательном значении модель, напротив, стремится многократно их использовать. Чем выше абсолютное значение, тем сильнее соответствующий эффект.

Параметр `frequency_penalty` связан с предыдущим параметром, но позволяет использовать более детализированную схему штрафов. Параметр `presence_penalty` учитывает сам факт *наличия* токена. То есть нет разницы, сколько раз повторяется токен: дважды или сотни раз. Штраф за частоту использования принимается как коэффициент, умножающий количество предыдущих появлений токена при суммировании его оценки (которая определяет, должен ли токен появиться следующим). Следовательно, чем чаще токен использовался ранее, тем меньше вероятность, что он появится снова. Как и у параметра `presence_penalty`, значения параметра `frequency_penalty` варьируются в диапазоне от `-2` до `+2` с заданным по умолчанию значением `0`. Положительное значение штрафа побуждает модель GPT избегать повторения одного и того же токена, тогда как отрицательное значение способствует его повторам.

Иногда нас интересует только один элемент из ограниченного набора допустимых токенов. Например, при классификации текста набор классов обычно определяется заранее. Если так, то сообщим об этом модели! Параметр `logit_bias` позволяет сопоставлять идентификаторы токенов с коэффициентом смещения. Высокий коэффициент смещения побуждает модель рассматривать соответствующий токен как вывод. Если установить достаточно низкое смещение, то модель практически перестанет использовать такой токен.

Достаточно высокий показатель практически гарантирует, что соответствующий токен появится в выводе.

Использование параметра `logit_bias` позволяет избежать генерации бесполезных результатов вывода в ситуациях, когда можно сузить набор логично применяемых токенов. Значение этого параметра представляет собой словарь Python, который сопоставляет идентификаторы токенов со значениями в диапазоне от `-100` до `+100`. Более типичны значения в интервале от `-1` до `+1`, которые позволяют модели учитывать токены с низким весом (или избегать токенов с высокими значениями). Но как найти идентификаторы токенов, связанные с релевантными словами? Для этого можно воспользоваться инструментом токенизатора GPT, который доступен по адресу **https://platform.openai.com/tokenizer?view=bpe**. Просто введите слова, использование которых хотите поощрить (или запретить), — и отобразятся связанные с ними идентификаторы токенов. Обратите внимание, что доступно несколько вариантов токенизатора для разных моделей. Выберите подходящий для своей модели (иначе идентификаторы токенов могут быть некорректными).

3.5.3. Настройка рандомизации

Как модели GPT выбирают следующий выходной токен? Если говорить упрощенно, то мы вычисляем оценки для всех возможных выходных токенов, а затем на основе этих оценок выбираем токен. У токенов с более высокими оценками, как правило, больше шансов быть выбранными, однако нам не всегда требуется токен с максимальной оценкой. Например, вспомните, как в главе 2 можно было заново генерировать ответы на одни и те же входные данные, что потенциально приводило к разным результатам. Это полезно, когда первый вывод не совсем удовлетворяет нашим требованиям. Если

всегда выбирать токены с наивысшими оценками, то при повторной генерации ответа вывод вряд ли изменится. Следовательно, чтобы пользователи могли получать разные ответы, необходимо ввести определенную степень рандомизации при преобразовании оценок в токены вывода.

Конечно, в результате слишком сильного отрыва вывода от оценок токенов — то есть чрезмерной рандомизации — вывод может оказаться бесполезным (в крайнем случае не связанным с входными данными и не соответствующим нашим инструкциям). С другой стороны, слишком малая рандомизация может привести к тому, что выводы будут менее разнообразны, чем нам хотелось бы. Пожалуй, чтобы выбрать верную степень рандомизации для конкретного сценария, придется поэкспериментировать. В каждом случае OpenAI предлагает несколько параметров, позволяющих дообучить модель тонкостям преобразования оценок токенов в выходные токены. Давайте обсудим эти параметры.

Один из наиболее часто используемых параметров для настройки рандомизации — `temperature`. Повышение значений температуры увеличивает степень рандомизации, а понижение — уменьшает ее. При низкой степени рандомизации, скорее всего, будет выбран токен с наивысшей оценкой. При очень высокой степени рандомизации токены выбираются (почти) с равной вероятностью, независимо от оценок, присвоенных моделью. Параметр `temperature` позволяет найти золотую середину между этими двумя крайностями. Его значения выбираются в диапазоне от `0` до `2`, но по умолчанию задано `1`.

Температура — один из вариантов выбора степени рандомизации. Альтернативный подход — использовать параметр `top_p`. (Не рекомендуется в одном вызове языковой модели изменять одновременно параметры `temperature` и

top_p.) По этим оценкам можно рассчитать, какова вероятность того, что тот или иной токен окажется «правильным». Теперь представьте, что мы сортируем токены в порядке убывания вероятности. Можно уменьшить степень рандомизации и сосредоточиться только на первых нескольких токенах, но тогда мы пренебрегаем токенами с меньшей вероятностью. Сколько токенов следует рассматривать? Вместо того чтобы напрямую фиксировать количество допустимых токенов, параметр top_p показывает *суммарную вероятность* этих токенов. Другими словами, токены добавляются в набор допустимых в порядке убывания их вероятности. Всякий раз, когда сумма значений вероятности всех выбранных токенов (суммарная вероятность) превышает значение top_p, добавление токенов прекращается. В итоге следующий выходной токен выбирается из этих допустимых токенов.

Поскольку параметр top_p описывает вероятность, то его значения берутся из интервала от 0 до 1. Как и в случае с температурой, выбор более высокого значения приводит к большей рандомизации (ведь допустимыми становятся даже токены с меньшей вероятностью).

При наличии ненулевой рандомизации целесообразно запрашивать несколько ответов на один и тот же входной промпт. После этого можно выбрать предпочтительный ответ с помощью последующей обработки. Например, предположим, что мы генерируем несколько SQL-запросов для одного и того же входного промпта. Чтобы выбрать предпочтительный ответ, можно попробовать выполнить их в целевой базе данных и отбросить запросы, которые приводят к сообщению об ошибке синтаксиса. Конечно, можно просто несколько раз вызывать языковую модель, вводя один и тот же промпт. Однако эффективнее вызвать модель один раз и настроить количество генерируемых ответов. Параметр n определяет это количество.

По умолчанию данный параметр установлен в значение 1 (то есть генерируется только один ответ). Для получения большего количества ответов можно выбрать большее значение. Обратите внимание: чем выше значение этого параметра, тем дороже обходится вызов модели (ведь вы платите за каждый сгенерированный токен, учитывая токены в разных ответах).

3.5.4. Пример настройки

Применим некоторые параметры в коде. В листинге 3.3 мы даем модели GPT-4o задание написать историю, причем на этот раз используем некоторые из описанных выше параметров для настройки диалогового режима.

Сначала, чтобы история не получилась слишком длинной, зададим максимальное количество токенов равным 512 ❶. Этого должно хватить на несколько абзацев текста. Чтобы избежать генерации более длинного контента, чем необходимо, мы также определяем стоп-последовательность: «долго и счастливо» ❷. Мы делаем это в надежде, что любая логичная история, конечно же, закончится популярным выражением «и жили они долго и счастливо». Разумеется, это допущение.

Листинг 3.3. Использование модели GPT-4o для генерации ответов в диалоге с применением пользовательских настроек параметров

```
import openai
client = openai.OpenAI()

result = client.chat.completions.create(
    model='GPT-4o',
    messages=[{
        'role':'user',
        'content':'Расскажи историю!'
    }],
    max_tokens=512,             ❶ Ограничивает длину вывода
    stop='долго и счастливо',   ❷ Задает условие остановки
    temperature=1.5,            ❸ Задает температуру
    presence_penalty=0.5,       ❹ Штрафует за повторения
    logit_bias={14844:-100})    ❺ Добавляет смещение
print(result.choices[0].message.content)
```

Далее мы задаем значение температуры `1.5` ❸, разрешая определенную степень рандомизации. Если в истории будет слишком много повторений, она будет скучной. Поэтому мы устанавливаем положительное значение штрафа за наличие токенов (`presence_penalty=0.5`) ❹, которое не позволит модели GPT-4o повторно использовать одни и те же токены. Наконец, нам не нужна страшная история (вдруг мы будем читать ее детям), поэтому мы штрафуем за слово «дракон» (получив соответствующий идентификатор токена, как было описано в подразделе 3.5.2) ❺.

Посмотрим на результат! Например, при одном из обращений к модели GPT-4o они сгенерировала такую историю.

> Давным-давно в живописной деревушке, затерянной среди холмов и густых лесов, жила девочка по имени Элара. Любознательность и жажда приключений были ее отличительными чертами — они-то и заставляли ее то и дело исследовать самые дальние закоулки деревни. Однажды погожим осенним утром Элара решила отправиться в Шепчущий Лес, место, окутанное тайной и легендами. Жители деревни часто говорили, что в лесу живет магия, и никто не осмеливался заходить дальше окраины. Взяв свою любимую

котомку и трепеща от предвкушения, Элара на рассвете отправилась в путь. Чем глубже она заходила в лес, тем сильнее, казалось, становится шелест листьев, словно приветствующих ее, и возникло странное чувство, будто все это ей знакомо. Она пробиралась сквозь заросли и перелезала через упавшие бревна, пока не наткнулась на уединенную рощицу, залитую мерцающим светом. В центре рощи стояло древнее дерево с серебристыми листьями, его ствол перекручивался, подобно струйкам дыма. Элара была очарована; оно не было похоже ни на одно дерево из когда-либо виденных ею. Девочка осторожно приблизилась к нему и заметила маленький, искусно украшенный сундучок, спрятавшийся среди корней. Сердце девочки забилось чаще. Элара открыла сундучок и обнаружила внутри изящный золотой кулон. Как только она его коснулась, лес будто ожил. Шепот стал громче, складываясь в слова, которые она могла понять. «Храброе сердце, ты нашла Кулон Этернии, — прошептал голос. — Он дарует носителю одно желание». В голове у Элары проносились разные возможности, но она знала, чего ее сердце желает больше всего: принести радость и процветание любимой деревне. Сжимая в руке кулон, Элара прошептала желание. В ту же секунду теплое сияние окутало девочку, и она почувствовала прилив энергии. Роща на мгновение замерцала, а затем снова стала безмятежно спокойной. Элара знала, что произошло нечто необычайное. Возвращаясь домой, она заметила небольшие изменения: урожай выглядел более обильным, животные здоровее, а сама деревня словно купалась в атмосфере покоя и довольства. Жители деревни радостно встретили ее возвращение, их лица

светились счастьем. Элара никогда никому не раскрывала тайну Шепчущего Леса или Кулона Этернии. Она лишь улыбалась всякий раз, когда кто-то заговаривал о неожиданном процветании деревни. Девочка знала, что иногда величайшая магия заключается не в заметных действиях, а в мужестве следовать зову сердца. С той поры деревушка Элары зажила в довольстве и мире, превратившись в истинный символ радости и благополучия. Все потому, что одна храбрая девочка не побоялась прислушаться к шепоту леса. И она жила

Долго и счастливо! Оказывается, наша стоп-последовательность, выражение «долго и счастливо», действительно использовалась в конце сказки (и поэтому опущена в выводе, который вернула модель GPT-4o). Протестируйте модель на других настройках параметров и посмотрите, как будет меняться результат в зависимости от конфигурации.

3.5.5. Дополнительные параметры

Мы обсудили наиболее важные параметры для целей анализа данных. Их все можно указывать при обращении к моделям GPT от OpenAI для генерации ответа в режиме диалога. Однако, помимо упомянутых в главе, существуют и другие параметры. Они подробно описаны в справочной документации по API OpenAI (**https://platform.openai.com/docs/api-reference/completions**).

Резюме

- Языковые модели OpenAI можно использовать через Python API. Другие поставщики предлагают похожие библиотеки для доступа к своим моделям.

- Чтобы использовать библиотеку OpenAI, создайте объект клиента.

- Модели OpenAI можно использовать для генерации ответов в режиме диалога. Для этого нужно передать историю переписки в виде списка сообщений.

- Каждое сообщение чата характеризуется содержанием (`content`) и ролью (`role`). Варианты ролей: `user` (пользователь), `assistant` (ассистент) или `system` (система).

- Для получения ответов в диалоговом режиме предназначена функция `chat.completions.create`.

- Модели можно настраивать, используя различные параметры:

    - с помощью `max_tokens` можно ограничить количество генерируемых токенов;

    - `stop` позволяет определять фразы, которые останавливают генерацию текста;

    - с помощью `logit_bias` можно штрафовать или поощрять модель за использование определенных токенов;

    - `presence_penalty` позволяет штрафовать модель за повторение токенов независимо от их частоты;

- с помощью `frequency_penalty` тоже можно штрафовать модель за повторение токенов, но уже пропорционально их частоте;
- `temperature` позволяет выбрать степень рандомизации;
- с помощью `top_p` можно определить количество рассматриваемых токенов вывода;
- `n` позволяет выбрать количество генерируемых ответов.

4. Анализ текстовых данных

В этой главе

- Классификация текста.
- Извлечение информации.
- Кластеризация документов.

Текстовые данные используются повсеместно и содержат ценную информацию. Пример таких данных — газетные статьи, электронные письма, отзывы или книга, которую вы читаете. Однако до недавнего времени анализировать текст с помощью компьютера было сложно. В конце концов, в отличие от формальных языков наподобие Python, естественный язык изначально не был рассчитан на то, чтобы компьютеры могли легко его обрабатывать. Новейшее поколение языковых моделей позволяет для решения многих популярных задач выполнять анализ текста почти так же хорошо, как это сделал бы человек. В некоторых случаях было показано, что производительность языковых моделей в области анализа и генерации текста в среднем даже превосходит возможности людей [1].

В этой главе мы рассмотрим, как использовать большие языковые модели для анализа текста. В определенном смысле анализ текстовых данных — весьма естественное применение языковых моделей. Они обучались на больших объемах текста, и их можно использовать непосредственно для анализа текста (то есть не обращаясь к внешним инструментам для фактического анализа данных). В этой главе описываются

несколько популярных видов анализа текста: классификация документов, извлечение данных из текста и кластеризация — объединение документов в группы на основе смыслового сходства. Для каждого из этих сценариев использования мы рассмотрим пример кода и обсудим варианты и дополнительные возможности.

Классификация, извлечение информации и кластеризация важны, но это далеко не единственные виды анализа текста, которые могут вам понадобиться на практике. Тем не менее, поработав с примерами из этой главы, вы сможете создавать собственные конвейеры обработки текстовых данных на основе языковых моделей.

4.1. Подготовка

Сначала убедитесь, что ваша система правильно настроена для работы с примерами проектов. В примерах этой главы используется серия моделей GPT от OpenAI, доступ к которым осуществляется через библиотеку Python от OpenAI. Мы подробно рассматривали ее в главе 3. Удостоверьтесь, что выполнили инструкции из главы 3 и имеете возможность запускать примеры кода.

> **ПРЕДУПРЕЖДЕНИЕ** Библиотека Python от OpenAI быстро меняется. Код, представленный в этой главе, был протестирован в версии 1.29 этой библиотеки, но, возможно, он не будет работать с другими версиями.

Помимо библиотеки OpenAI, мы будем пользоваться популярной библиотекой `pandas`. Она широко применяется для работы с табличными данными (именно в этом формате у нас будет выполняться ввод и вывод данных). Мы будем работать только с базовой функциональностью этой библиотеки и

разбирать соответствующие команды по мере их появления в коде. Убедитесь, что библиотека `pandas` установлена (например, попробуйте выполнить команду `import pandas` в интерпретаторе Python). В противном случае установите ее, введя в терминале следующую команду:

```
pip install pandas==2.2
```

Наконец, для работы с последним разделом главы вам понадобятся алгоритмы кластеризации из библиотеки `scikit-learn`. Чтобы установить соответствующую ее версию, наберите в терминале команду:

```
pip install scikit-learn==1.3
```

В следующих разделах содержится код трех мини-проектов, в которых языковые модели используются для анализа текста. Вводить код вручную не нужно — его можно найти по ссылкам, размещенным на сайте книги, в разделе дополнительных материалов для этой главы. Конечно, вы можете выполнять код на собственных данных, но в книге есть несколько наборов, которые используются в примерах (ссылки на наборы также есть на сайте книги). А теперь пришло время применить языковые модели для классификации текста!

4.2. Классификация

Итак, вы строите планы на субботний вечер и думаете, не посмотреть ли новую часть любимой кинофраншизы. Но стоит ли тратить время? В ваших лентах в социальных сетях появляется все больше комментариев друзей (и их друзей), которые делятся своими впечатлениями от фильма. Эти отзывы можно просмотреть вручную, читая каждый, чтобы

лучше понять, каких мнений больше: положительных или отрицательных. Вот только у кого есть на это время? Не могут ли языковые модели помочь автоматизировать эту задачу?

Вообще-то, могут. Перед нами классический случай, одна из самых типичных задач по обработке текста: есть текст и его нужно классифицировать, сопоставив с категорией из фиксированного набора вариантов. В данном случае текст для классификации — это отзыв о фильме. Его нужно определить как положительный (то есть автор отзыва считает, что фильм отличный и его стоит посмотреть!) или отрицательный («Сэкономьте деньги!»). Значит, у нас есть две категории. В табл. 4.1 приведены фрагменты нескольких примеров отзывов с соответствующими метками классов. Отзыв, восхваляющий фильм как «хорошо снятый», явно положительный, тогда как отзыв, описывающий его как «явно слабый, дешевый», — отрицательный. Эти и другие отзывы можно найти в соответствующем файле, доступном по ссылке **Reviews.csv**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе.

Таблица 4.1. Фрагменты отзывов о фильмах и соответствующие метки классов

Отзыв	Класс
Прежде всего этот фильм — кусочек реальности, мастерски воплощенный художественно	Положительный
Переименованный «Банды Инкорпорейтед» — откровенно слабая, дешевая мелодрама о гангстерах	Отрицательный

Это лишь один из многих сценариев использования классификации текста.

Еще один пример: представьте, что вы пытаетесь разобраться во входящих электронных письмах. Не правда ли, было бы замечательно автоматически классифицировать их на

основе содержания (например, используя пользовательские категории, такие как «Работа», «Хобби», «Уход за детьми» и т.д.)? И снова перед нами задача классификации текста, на этот раз с несколькими категориями.

И последний пример: представьте, что вы создаете сайт, который позволяет пользователям оставлять текстовые комментарии в свободной форме. Конечно, вы не хотите показывать потенциально оскорбительные комментарии, и хорошо бы их автоматически отфильтровывать. Опять же, это означает, что текстовые комментарии классифицируются как относящиеся к одной из двух категорий («Оскорбительные» и «Не оскорбительные»). Теперь посмотрим, как легко языковые модели справляются с каждым из этих сценариев.

4.2.1. Обзор

Здесь мы сосредоточимся на разделении отзывов о фильмах (или, по сути, любых отзывов) на положительные («Отличный фильм!») и отрицательные («Оставайтесь дома!»). Для этого используем языковые модели OpenAI. Предположим, что мы собрали необходимые отзывы и поместили в файл на диске. Разработанный нами код будет перебирать все отзывы, классифицировать каждый с помощью языковой модели и возвращать результат классификации для каждого отзыва.

Но как классифицировать отзывы? С помощью библиотеки Python от OpenAI, представленной в главе 3. Для каждого отзыва, который нужно классифицировать, сначала сгенерируем промпт. Он описывает задачу для языковой модели. В случае нашего примера эта задача такова: модель должна отнести отзыв к одной из двух категорий («Положительный» или «Отрицательный»). Например, рассмотрим следующий промпт:

```
Этот фильм — кусочек реальности, мастерски воплощенный...  ❶ Отзыв
Отзыв положительный или отрицательный?                     ❷ Вопрос
Ответ ("Положительный"/"Отрицательный"):                   ❸ Формат вывода
```

В этом промпте содержатся: отзыв для классификации ❶, вопрос, описывающий задачу классификации ❷, и окончательное утверждение, описывающее желаемый формат вывода ❸. Мы составим промпты такого типа для каждого отзыва, отправим промпт языковой модели и (надеюсь) получим обратно один из двух возможных ответов («Положительный» или «Отрицательный»). На рис. 4.1 показана общая схема процесса классификации для каждого отзыва.

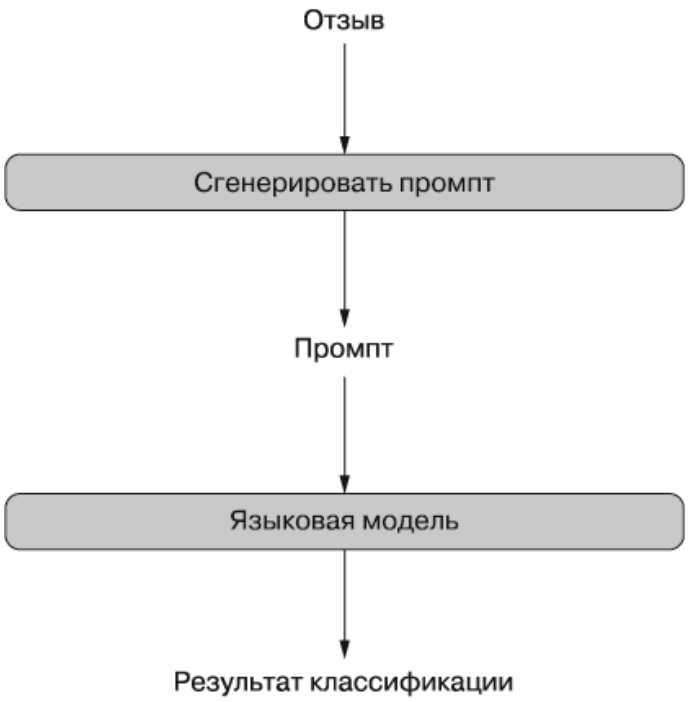


Рис. 4.1. Для каждого отзыва мы генерируем промпт, который содержит отзыв вместе с инструкциями, описывающими задачу классификации. Получив промпт в качестве входных данных, языковая модель выводит для отзыва метку класса

4.2.2. Составление промптов

При наличии отзыва мы генерируем промпт, дающий языковой модели указание классифицировать этот отзыв. Все

генерируемые для классификации промпты строятся по одному шаблону.

> **Напоминание: что такое шаблон промпта**
> Шаблоны промптов упоминались в главе 1. Шаблон промпта — это текст, который содержит метки-заполнители. Подставляя вместо заполнителей реальный текст, мы получаем промпт, который можно отправить языковой модели. Мы также говорим, что *создается экземпляр* промпта по шаблону, если его можно получить путем подстановки значений в метки-заполнители шаблона.

Тестовый промпт, рассмотренный в предыдущем подразделе, является экземпляром следующего шаблона промпта:

```
[Отзыв]                                          ❶ Отзыв (заполнитель)
Отзыв положительный или отрицательный?           ❷ Вопрос
Ответ ("Положительный"/"Отрицательный"):         ❸ Формат вывода
```

Шаблон содержит только одну метку-заполнитель: текст отзыва для классификации ❶. Для каждого отзыва эта метка-заполнитель заменяется фактическим текстом отзыва. Кроме того, мы даем модели инструкцию, что делать с текстом отзыва ❷ (проверить его тональность: положительный он или отрицательный) и определяем формат вывода ❸. Последний шаг важен, ведь способов обозначить тональность отзыва бывает множество: например, использовать для положительного отзыва букву «П», а для отрицательного — «О» или дать более развернутый ответ, скажем «Отзыв положительный». Если мы явно не укажем языковой модели, что нужно использовать определенный формат вывода, то она может выбрать любой из этих вариантов! В нашем сценарии в конечном счете нужно будет агрегировать результаты классификации, чтобы узнать мнение большинства (нравится

ли фильм большинству людей?), а агрегировать результаты каждого отзыва намного проще, если все классификации соблюдают один формат вывода.

Приведенная ниже функция генерирует промпт для заданного отзыва (указанного как входной параметр `text`), используя описанный шаблон:

```
def create_prompt(text):
    task = 'Отзыв положительный или отрицательный?'
                    answer_format     =     'Ответ
("Положительный"/"Отрицательный")'
    return f'{text}\n{task}\n{answer_format}:'
```

Функция возвращает готовый промпт — это конкретный экземпляр шаблона, заполненный данными из отзыва.

4.2.3. Вызов модели

Далее отправим сгенерированные промпты языковой модели, чтобы получить решение. А именно, используем модель GPT-4o компании OpenAI — последнюю модель на момент написания книги. Поскольку это одна из чат-моделей OpenAI, оптимизированная для многоэтапного взаимодействия с пользователями, то для связи с ней мы используем эндпоинт `chat.completions`. Как подробно рассказывалось в главе 3, этот эндпоинт ожидает в качестве входных данных историю предыдущих сообщений (в дополнение к конкретному имени модели). Здесь у нас есть только одно предыдущее «сообщение»: промпт. Мы классифицируем его как сообщение пользователя, побуждая модель решить любую задачу, описанную в сообщении. Например, мы можем отправлять промпты языковой модели и собирать ответы с помощью

следующего фрагмента кода (при условии, что параметр `prompt` содержит ранее сгенерированный текст промпта):

```
import openai
client = openai.OpenAI()

response = client.chat.completions.create(
    model='gpt-4o',
    messages=[
        {'role':'user', 'content':prompt}
        ]
    )
```

Однако использовать этот код напрямую проблематично. Модели GPT от OpenAI размещены онлайн и доступны удаленно. Из-за этого могут возникнуть сбои при попытке достичь соответствующего эндпоинта: например, из-за временного разрыва соединения. В связи с этим при вызове модели рекомендуется разрешать выполнение нескольких повторных попыток. В частности, при обработке больших наборов данных, требующих множества последовательных вызовов моделей OpenAI, шансы, что как минимум один вызов завершится неудачей, возрастают. Вместо того чтобы прерывать вычисления, выводя сообщение об исключении, лучше подождать несколько секунд, прежде чем делать следующую попытку. Ниже приведена завершенная версия предыдущего кода — функция, которая вызывает языковую модель с автоматическими повторными попытками:

```
import openai
client = openai.OpenAI()

def call_llm(prompt):
```

```
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                    {'role':'user',
'content':prompt}
                    ]
                )
                                                return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')
```

Функция `call_llm` позволяет выполнить до трех повторных попыток с увеличивающимся периодом задержки между ними. Эта задержка реализуется путем вызова функции `time.sleep` (с использованием модуля `time` из Python-библиотеки) всякий раз при возникновении исключения (указывающего, например, на временный разрыв соединения). После трех повторных попыток функция завершается исключением (пессимистично предполагая, что, какая бы проблема ни мешала нам связаться с OpenAI, она не будет решена в ближайшее время). При успешном вызове функция каждый раз возвращает соответствующий результат.

4.2.4. Полный рабочий код для классификации данных

Пришло время все объединить! В листинге 4.1 показан код, соответствующий процессу классификации, который мы обсуждали в предыдущем подразделе. Кроме того, этот код содержит функцию для генерации промптов ❷ и функцию для вызова языковой модели ❸.

Листинг 4.1. Классификация входного текста по тональности отзыва (положительный, отрицательный)[7]

```
import argparse                                          ❶
```

Импортирует библиотеки

```
import openai
import pandas as pd
import time

client = openai.OpenAI()


def create_prompt(text):                                 ❷
```

Генерирует промпты для классификации

```
    """Сгенерировать промпт для классификации по
тональности.

    Аргументы:
        text: классифицируй текст.

    Выводит:
        Входные данные для LLM.
    """
    task = 'Отзыв положительный или отрицательный?'
                    answer_format     =     'Ответ
("Положительный"/"Отрицательный")'
    return f'{text}\n{task}\n{answer_format}:'
```

```
def call_llm(prompt):                                    ❸
```

Вызывает большую языковую модель

```
    """Выполнить запрос к большой языковой модели и
вернуть ответ.

    Аргументы:
        prompt: введите запрос к языковой модели.

    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                    {'role':'user',
'content':prompt}
                    ]
                )
                                                 return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')
```

```
def classify(text):                                    ❹
```

Классифицирует один текстовый документ

```
    """Классифицировать входной текст.

    Аргументы:
            text: присвоить данному тексту метку
класса.

    Выводит:
        Имя класса.
    """
    prompt = create_prompt(text)
    label = call_llm(prompt)
    return label

if    __name__    ==    '__main__':                  ❺
```

Читает текст, выполняет классификацию и записывает результат

```
     parser = argparse.ArgumentParser()              ❻
```

Определяет аргументы командной строки

```
        parser.add_argument('file_path', type=str,
help='Путь к входному файлу')
    args = parser.parse_args()

     df = pd.read_csv(args.file_path)                ❼
```

Читает входные данные

```
       df['class'] = df['text'].apply(classify)      ❽
```

Классифицирует текст

```
    statistics = df['class'].value_counts()    ❾
```

Генерирует вывод

```
print(statistics)
df.to_csv('result.csv')
```

Сначала обсудим библиотеки, используемые в этом листинге ❶. Мы будем повторно использовать их в следующих проектах, поэтому имеет смысл рассмотреть их подробнее (и понять, зачем они здесь нужны). Мы собираемся запускать код из командной строки, указывая в качестве аргументов соответствующие параметры (например, путь к входным данным). В модуле `argparse` есть удобные средства для описания и считывания подобных аргументов, переданных через командную строку. Далее нам нужна библиотека `openai` (см. главу 3), чтобы вызывать языковую модель OpenAI из Python. Библиотека `pandas` поддерживает стандартные операции с табличными данными. Конечно, в рамках этой главы табличные данные нас мало интересуют. Однако мы будем хранить текстовые документы и связанные метаданные в виде строк в таблицах, поэтому будет удобно использовать библиотеку `pandas`. Наконец, как обсуждалось ранее, мы используем модуль `time` для реализации повторных попыток в случае задержки при вызове языковой модели.

4.2.5. Классификация документов

Процесс классификации одного текстового документа ❹ объединяет две функции, которые мы обсуждали выше. При наличии входного текста для классификации код сначала создает соответствующий промпт (вызов функции `create_prompt`), а затем генерирует подходящий ответ, вызывая языковую модель (вызов функции `call_llm`).

Предполагается, что в результате пользователь получит метку класса.

Теперь объединим все действия ❺ . Эта часть кода выполняется при вызове модуля Python из командной строки и использует введенные нами функции. Начальное условие `if` ❺ гарантирует, что следующий код выполняется только при прямом вызове модуля (а не при его импорте другого модуля).

Сначала ❻ определим аргументы командной строки. Здесь нужен только один аргумент: путь к файлу в формате `.csv`, содержащему данные для классификации. Мы предполагаем, что каждая строка содержит один текстовый документ и что текст для классификации находится в столбце `text`. Мы разбираем аргументы командной строки и делаем их значения доступными в переменной `args`.

Затем загрузим входные данные с диска ❼ . Мы предполагаем, что данные хранятся в виде файла в формате `.csv` (значения, разделенные запятыми): есть строка заголовка с именами столбцов, за которой следуют строки, содержащие данные (поля разделены запятыми, как следует из названия). Для этого удобно использовать библиотеку `pandas`, которая позволяет загружать такие данные одной командой. После этого в переменной `df` окажется объект DataFrame библиотеки `pandas`, содержащий данные из входного файла. Мы извлекаем столбец `text` из DataFrame ❽ и применяем ранее определенную функцию `classify` к каждой строке (используя метод `apply` библиотеки `pandas`). Наконец ❾, мы генерируем и выводим агрегированную статистику (количество вхождений для каждого ответа, сгенерированного моделью) и записываем полученные классификации в файл (`result.csv`).

4.2.6. Запуск кода

Используя ссылку **Reviews.csv** (см. раздел дополнительных материалов к этой главе), скачайте файл `reviews.csv`. В нем содержится несколько отзывов о фильмах, которые можно использовать для классификации. В файле два столбца: текст отзыва и его тональность (`neg` — отрицательный и `pos` — положительный). Нашей целью, безусловно, является автоматическое определение тональности. Однако наличие эталонных данных также позволяет нам оценить качество классификаций.

Вы можете протестировать код для классификации, как описано ниже (команды были проверены в операционной системе Linux). В терминале перейдите в каталог, содержащий модуль Python (`listing1.py`) с кодом из листинга 4.1. Затем выполните следующую команду (заменив `python` на имя вашего интерпретатора Python, например `python3`, если необходимо):

```
python listing1.py reviews.csv
```

Здесь мы предполагаем, что входной файл (`reviews.csv`) хранится в том же репозитории, что и код (в противном случае необходимо прописать соответствующий путь к имени файла). Обычно выполнение кода не должно занимать более нескольких секунд (немного дольше, если соединение нестабильно и требуются повторные попытки). При успешном выполнении в выводе вы увидите только сумму меток, присвоенных каждому из двух возможных классов.

После выполнения кода файл `result.csv` будет сохранен в тот же репозиторий. Помимо столбцов входного файла, итоговый файл содержит новый столбец `class`. В нем хранятся результаты классификации (положительные и отрицательные). Сравните метку, присвоенную классификатором, с эталонным

вариантом. Вы увидите, что в большинстве случаев классификации совпадают. Неплохо для нескольких строк кода на Python, не правда ли?

4.2.7. Проверка вариантов в работе

На этом этапе можно немного поэкспериментировать с кодом и данными, чтобы лучше понять, как он работает. Например, попробуйте сами написать несколько отзывов о фильмах! При каких отзывах классификация надежна, а при каких возникают затруднения? Кроме того, попробуйте ввести несколько вариантов промптов. Какие инструкции повышают точность, а какие снижают производительность? Для примера попробуйте удалить часть промпта, которая точно определяет формат вывода (строку `Ответ ("Положительный"/"Отрицательный")`). Теперь запустите программу с измененным промптом. Что произойдет? Скорее всего, в результате классификации (в выводе программы) вы получите больше двух вариантов меток, включая сокращения (например, «П» и «О»), а также излишне подробные ответы (к примеру, во время тестирования модель GPT-4o генерировала ответы типа «Тон этого отзыва положительный»). В главе 9 мы оценим влияние различных промптов на качество вывода модели.

Кроме того, можно попробовать использовать для извлечения другую модель. Что, если применить версию модели меньшего размера, скажем GPT-3.5 (стоимость одного обработанного ею токена гораздо ниже)? А как насчет конфигурации модели? В листинге 4.1 используются только два параметра (имя модели и история сообщений), и оба — обязательные. Однако в главе 3 описаны и другие параметры конфигурации, которые можно здесь применить. Например, попробуйте изменить параметр `temperature` (так, задав значение `temperature`, равное `0`, вы получите более

предсказуемые результаты) или ограничьте длину желаемого вывода. В редких случаях модели GPT способны генерировать текст вывода, превышающий по длине желаемый результат классификации (состоящий из одного токена). Этого можно избежать, ограничив длину вывода с помощью параметра `max_tokens`. Как вариант, можно не ограничивать формат вывода только инструкциями в промпте, а повысить вероятность двух возможных результатов (положительного и отрицательного) с помощью параметра `logit_bias`. Подробнее настройку модели мы обсудим в главе 9.

Еще один вариант — изменить задачу классификации. Например, относительно легко создать классификацию с использованием другого набора категорий. Потребуется всего лишь изменить инструкции в промпте (перечислив все варианты ответов, как и раньше). К тому же, переписав несколько строк кода, можно получить универсальный инструмент классификации, позволяющий пользователям указывать задачу классификации и соответствующие классы в качестве дополнительных аргументов командной строки. Например, помимо отзывов о фильмах, можно использовать этот инструмент для категоризации газетных статей по нескольким тематическим категориям или для разделения электронных писем на «Срочные» или «Несрочные». К моменту чтения этих строк, надеюсь, вы убедились, что языковые модели способны выполнять классификацию текста, выдавая относительно высокое качество, требуя при этом умеренных затрат на реализацию. А теперь перейдем к другим задачам.

4.3. Извлечение данных из текста

Этот вид анализа мы рассмотрим на примере уже знакомой вам компании Banana, производящей бытовую электронику.

Напомню, что в главе 2 вы уже выступали в роли ее сотрудника. Добавлю, что работу в ней (очень престижную!) вы получили благодаря своему опыту в области анализа данных с помощью языковых моделей.

Представьте, что вам начали поступать электронные письма от студентов, которых интересует летняя стажировка. Обзавестись стажером на лето было бы неплохо, но как выбрать лучшего кандидата? В идеале вы хотели бы составить таблицу, сравнивающую всех кандидатов по их среднему баллу, степени, названию компании, в которой они проходили последнюю стажировку (если таковая была), и т.д. Но читать электронные письма самому, чтобы составить такую таблицу, — весьма трудоемкое занятие. Нельзя ли автоматизировать процесс?

Конечно, можно. Применим для анализа электронных писем языковые модели. Пусть они извлекут все релевантные факторы, которые позволят выбрать стажера. Здесь мы снова сталкиваемся со стандартной задачей анализа текста — извлечением информации. В рамках извлечения информации из текста обычно можно получить структурированную информацию (например, табличные данные). В нашем примере мы рассматриваем электронные письма (от соискателей) как текстовые документы. Из каждого письма требуется извлечь ряд атрибутов: имя, средний балл и научную степень (получаемую в данный момент или последнюю полученную). Возьмем такой фрагмент письма одного из соискателей:

```
Привет!
Меня зовут Мартин, я хотел бы пройти летнюю стажировку в Banana!
Немного о себе: в настоящее время я получаю степень бакалавра компьютерных наук в Стэнфордском университете, мой средний балл — 4,0.
```

Рассматривая три упомянутых ранее атрибута, мы можем извлечь имя соискателя (Мартин), его средний балл (4,0) и научную степень (бакалавр компьютерных наук). Проанализировав письма от нескольких соискателей, можно представить результат, как показано в табл. 4.2.

Таблица 4.2. Извлеченная информация о соискателях на летние стажировки

Имя	Средний балл	Научная степень
Мартин	4,0	Бакалавр компьютерных наук
Алиса	4,0	Магистр программной инженерии
Боб	3,7	Бакалавр дизайна

Далее мы обсудим, как выполнить извлечение информации с помощью языковых моделей.

4.3.1. Обзор

И снова предположим, что электронные письма хранятся на диске (в файле табличных данных, где в каждой строке содержится по одному письму). С помощью языковой модели мы будем перебирать письма и извлекать все релевантные атрибуты. Вместо того чтобы строго прописывать релевантные атрибуты в коде, позволим пользователям указывать их в командной строке (тогда код можно будет использовать повторно, если критерии для отбора претендентов на летнюю стажировку изменятся). Поскольку для анализа текста используются языковые модели (которые хорошо интерпретируют естественный язык), то нет необходимости прописывать атрибуты на формальном языке. Просто перечислите имена атрибутов (или, как вариант, их краткие описания на естественном языке), а языковая модель должна

суметь разобраться, что извлекать. Результатом вывода кода станет файл табличных данных (в формате `.csv`), содержащий информацию, аналогичную указанной в табл. 4.2: в таблице вывода будет по одному столбцу для каждого извлеченного атрибута и по одной строке для каждого проанализированного письма.

Как же извлечь атрибуты из заданного письма? Нам снова нужно сгенерировать промпт, который будет описывать языковой модели задачу извлечения. Например, вот такой промпт должен помочь нам извлечь все релевантные атрибуты из предыдущего письма:

```
❶ Описание задачи
Извлечь в таблицу следующие свойства:
имя, средний балл, степень

❷ Текст для анализа
Текстовый источник: Меня зовут Мартин, я хотел бы пройти летнюю стажировку в
Banana!
Немного о себе: в настоящее время я получаю степень бакалавра компьютерных наук
в Стэнфордском университете, мой средний балл — 4,0.

❸ Формат вывода
Пометь начало таблицы тегом <BeginTable>, а конец — тегом <EndTable>. Раздели
строки символами новой строки, а поля — вертикальной чертой (|). Опусти
заголовок таблицы и вставляй значения в порядке следования атрибутов, начиная
сверху. Используй заполнитель <NA>, если значение атрибута недоступно.
```

Промпт состоит из трех частей:

- описания задачи, содержащего спецификацию извлекаемых атрибутов ❶;
- исходного текста, который необходимо извлечь ❷;
- желаемого формата вывода, включая значения для использования, если исходный текст не содержит информации по определенным атрибутам ❸.

Если передать такой промпт модели, то она должна вернуть текст с нужными нам извлеченными данными.

Вывод, который возвращает языковая модель, — это прежде всего текстовая строка. В конечном счете нам в выводе нужны структурированные табличные данные. Следовательно, потребуется обработка для выделения из ответа значений всех интересующих атрибутов (имя, средний балл, научная степень). На рис. 4.2 показаны этапы процесса извлечения (для одного текстового документа).

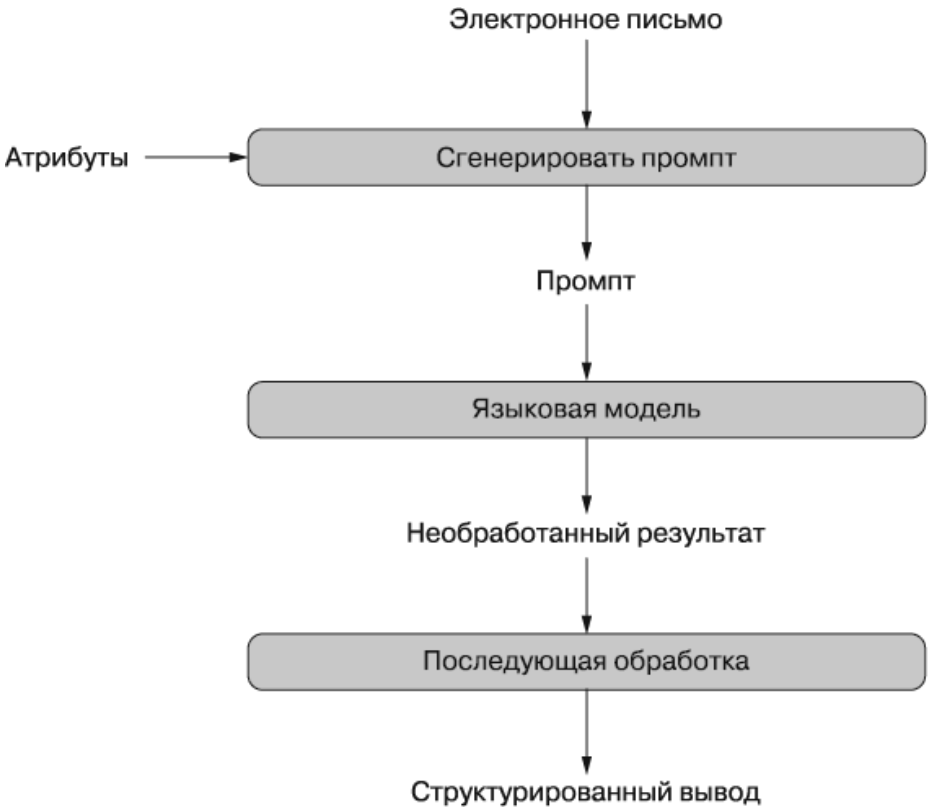


Рис. 4.2. Для каждого электронного письма мы генерируем промпт, который содержит само письмо и описание задачи извлечения. Это описание ссылается на атрибуты для извлечения, указанные пользователем. Получив промпт в качестве входных данных, языковая модель генерирует текстовый ответ, содержащий извлеченные значения атрибутов

Применив последующую обработку, мы извлечем эти значения из необработанного текста ответа.

4.3.2. Генерация промптов

Необходимо генерировать промпты, являющиеся экземплярами следующего шаблона:

```
❶ Описание задачи
Извлеки следующие свойства в таблицу:
[Список атрибутов]

❷ Текст для анализа
Источник текста: [Электронное письмо]

❸ Формат вывода
Пометь начало таблицы тегом <BeginTable>, а конец — тегом <EndTable>. Раздели
строки символами новой строки, а поля — вертикальной чертой (|). Опусти
заголовок таблицы и вставляй значения в порядке следования атрибутов, начиная
сверху. Используй заполнитель <NA>, если значение атрибута недоступно.
```

Шаблон промпта содержит описание задачи ❶ , исходный текст, который требуется извлечь ❷ , и описание формата вывода ❸ . Обратите внимание, что в промпте теперь содержатся две метки-заполнителя (шаблон из предыдущего подраздела содержал только одну): список атрибутов, которые нужно извлечь, и исходный текст для извлечения.

Мы будем генерировать промпты с помощью следующего кода:

```
def create_prompt(text, attributes):
    parts = []

    ❶ Генерирует описание задачи
    parts += ['Извлеки следующие свойства в таблицу:']
    parts += [','.join(attributes)]

    parts += [f'Источник текста: {text}']    ❷ Добавляет исходный текст

    ❸ Добавляет описание формата вывода
    parts += [
        ('Пометь начало таблицы тегом <BeginTable> '
        'а конец — тегом <EndTable>.')]
    parts += [
        ('Раздели строки символами новой строки '
        'а поля — вертикальной чертой (|).')]
    parts += [
        ('Опусти заголовок таблицы и вставляй значения в порядке '
        'следования атрибутов, начиная сверху.')]
    parts += [
        ('Используй заполнитель <NA>, если значение '
        'атрибута недоступно.')]
    return '\n'.join(parts)
```

Эта функция принимает в качестве входных данных текст для анализа (который мы, безусловно, хотим включить в промпт) вместе со списком атрибутов, которые требуется извлечь. После генерации описания задачи ❶ , в том числе списка атрибутов для извлечения, функция добавляет исходный текст ❷ , а также описание желаемого формата вывода ❸. Промпт объединяет эти части.

4.3.3. Последующая обработка

По сравнению с предыдущим проектом (классификация текста) наш промпт изменился, адаптировавшись к новой задаче (извлечение текста). Даже при наличии другого промпта мы все еще можем повторно использовать ту же функцию, что и в предыдущем подразделе, чтобы получить ответ от языковой модели. С другой стороны, нам придется приложить чуть больше усилий, чем раньше, чтобы обработать сырой ответ с помощью языковой модели. В случае классификации мы брали ответ языковой модели и сразу считали его конечным результатом. В текущем сценарии (извлечение текста), как правило, требуется извлечь значения нескольких атрибутов из одного входного текста. Поскольку текст, полученный от языковой модели, содержит значения всех извлеченных атрибутов, нужно будет извлечь значения для конкретных атрибутов из необработанного текста ответа.

Например, в качестве ответа модели мы можем получить следующий необработанный текст:

```
| Мартин | 4.0 | бакалавр компьютерных наук |
```

Для извлечения значений каждого атрибута этот текст можно разделить, используя в качестве разделителей полей вертикальные черты (при этом удалив в ответе первую и

последнюю вертикальную черту). В идеале решение нужно сделать универсальным, выйдя за рамки конкретного сценария использования, рассматриваемого в данный момент (извлечение из электронных писем информации о соискателях). Некоторые сценарии позволяют извлечь из одного и того же текста несколько строк (представьте сценарий, когда несколько соискателей вместе отправляют групповое письмо, но это маловероятно). Для поддержки подобных сценариев также может потребоваться разделить необработанный ответ на текст, связанный с разными строками. Для этого в качестве разделителя строк можно использовать символ новой строки (так как строки разделяются именно этим символом).

Все перечисленное можно выполнить с помощью следующей функции:

```
import re

def post_process(raw_answer):
    table_text = re.findall(                    ❶ Извлекает табличные данные
        '<BeginTable>(.*)<EndTable>',
        raw_answer, re.DOTALL)[0]

    results = []
    for raw_row in table_text.split('\n'):      ❷ Разделяет по строкам
        if raw_row:                             ❸ Разделяет по полям
            row = raw_row.split('|')
            row = [field.strip() for field in row]
            row = [field for field in row if field]
            results.append(row)
    return results
```

Входными данными для функции служит необработанный текст, созданный языковой моделью для одного текстового документа. Вывод — это список строк (где каждая строка результата, в свою очередь, представлена в виде списка). Чтобы перейти от входных данных к выводу, сначала нужно извлечь часть необработанного ответа, содержащую фактические табличные данные ❶ . Ответы, сгенерированные языковой моделью GPT-4o, помимо извлеченной таблицы, могут содержать вступление или дополнительные пояснения

(например, «Конечно, вот таблица, которая вам нужна: ...»). Необходимо отделить интересующие нас данные. К счастью, это несложно сделать при условии, что модель GPT-4o придерживается наших инструкций (а обычно так и происходит): нужные нам данные должны содержаться между двумя маркерами (`<BeginTable>` и `<EndTable>`). Следовательно, регулярное выражение `'<BeginTable>(.*)<EndTable>'` точно соответствует интересующей нас части вывода. Мы извлекаем ее с помощью функции Python `re.findall`, которая, получив в качестве входных данных строку и регулярное выражение, возвращает список соответствующих подстрок. Использование флага `re.DOTALL` позволяет гарантировать, что точка в регулярном выражении будет учитывать любые символы, в том числе символы разрыва строк (поскольку таблица может занимать несколько строк). Из полученных совпадений выбираем первое. Обратите внимание: предполагается, что в выводе модели GPT имеется хотя бы одна таблица. Обычно так и есть, однако стоит подумать, как сделать функцию более устойчивой к ответам языковой модели, не соответствующим инструкциям в промпте.

После извлечения табличных данных в текстовом представлении мы сначала разделяем их на данные, связанные с конкретными строками ❷ , и данные, связанные с конкретными ячейками ❸ . После очистки (функция Python `strip` удаляет символы пробела) мы добавляем полученные значения ячеек в список результатов. Возвращается список строк (где каждая строка, в свою очередь, представлена в виде списка).

4.3.4. Полный рабочий код извлечения данных

В листинге 4.2 показан полный код на языке Python. Структура похожа на представленную в листинге 4.1, и некоторые функции являются общими для обоих листингов (вместо того чтобы опускать повторяющиеся функции, я стараюсь предоставить готовый код, чтобы вам не пришлось собирать его по фрагментам с нескольких страниц). В частности, код использует те же библиотеки, что и раньше ❶ , и вызывает языковую модель через ту же функцию ❸ . Вы узнаете уже знакомые вам функции: для создания промптов ❷ и последующей обработки необработанного вывода языковой модели ❹.

Листинг 4.2. Извлечение определенных пользователем атрибутов из текста

```
import argparse                                          ❶
```

Импортирует необходимые библиотеки

```
import openai
import pandas as pd
import re
import time

client = openai.OpenAI()


def create_prompt(text, attributes):                     ❷
```

Генерирует промпты

```
        """Сгенерировать промпт для извлечения
информации.

    Аргументы:
        text: извлечь информацию из текста.
        attributes: список атрибутов.
```

```
    Выводит:
        Входные данные для LLM.
    """
    parts = []
        parts += ['Извлеки следующие свойства в
таблицу:']
    parts += [','.join(attributes)]
    parts += [f'Источник текста: {text}']
    parts += [
         ('Пометь начало таблицы тегом <BeginTable>
'
        'а конец — тегом <EndTable>.')]
    parts += [
        ('Раздели строки символами новой строки '
        'а поля — вертикальной чертой (|).')]
    parts += [
            ('Опусти заголовок таблицы и вставляй
значения в порядке '
        'следования атрибутов, начиная сверху.')]
    parts += [
        ('Используй заполнитель <NA>, если значение
'
        'атрибута недоступно.')]
    return '\n'.join(parts)


def call_llm(prompt):                                ❸
```

Вызывает языковую модель

```
     """Отправить запрос большой языковой модели и
вывести ответ.
```

```
    Аргументы:
        prompt: введите запрос к языковой модели.

    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                    {'role':'user',
'content':prompt}
                    ]
                )
                                                return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')


def  post_process(raw_answer):                      ❹
```

Выполняет постобработку ответа модели

```
         """Извлечь  поля  из  неструктурированного
текстового ответа.

    Аргументы:
            raw_answer: неструктурированный текст,
сгенерированный LLM.
```

```
    Выводит:
        Список получившихся строк.
    """
    table_text = re.findall(
        '<BeginTable>(.*)<EndTable>',
        raw_answer, re.DOTALL)[0]

    results = []
    for raw_row in table_text.split('\n'):
        if raw_row:
            row = raw_row.split('|')
            row = [field.strip() for field in row]
            row = [field for field in row if field]
            results.append(row)
    return results


def  extract_rows(text,  attributes):                    ❺
```

Извлекает из текста табличные данные

```
    """Извлечь значения атрибутов из текста.

    Аргументы:
        text: извлечь информацию из данного текста.
                attributes:  список  атрибутов  для
извлечения.

    Выводит:
        Список строк со значениями атрибутов.
    """
```

```
    prompt = create_prompt(text, attributes)
    result_text = call_llm(prompt)
    result_rows = post_process(result_text)
    return result_rows

if __name__ == '__main__':                                    ❻
```

Извлекает информацию и записывает результат

```
    parser = argparse.ArgumentParser()
        parser.add_argument('file_path', type=str,
help='Путь к входному файлу')
       parser.add_argument('attributes', type=str,
help='Список атрибутов')
    args = parser.parse_args()

    input_df = pd.read_csv(args.file_path)
    attributes = args.attributes.split('|')

    extractions = []
        for text in input_df['text'].values:  ❼
```

Выполняет итерацию по текстам

```
                extractions += extract_rows(text,
attributes)

    result_df = pd.DataFrame(extractions)
    result_df.columns = attributes
    result_df.to_csv('result.csv')
```

Функция `main` ❻ считывает два входных параметра из командной строки:

- путь к файлу в формате `.csv`, содержащему текст для анализа;
- список атрибутов для извлечения, разделенный вертикальными чертами.

После открытия входного файла (с помощью библиотеки `pandas`) мы перебираем все входные текстовые документы ❼. Обратите внимание: ожидается, что входной текст находится в столбце `text` входного файла. Для выполнения фактического извлечения применяется функция `extract_rows` ❺. Получив входной текст и список атрибутов для извлечения, эта функция генерирует соответствующий промпт, получает необработанный ответ от языковой модели и проводит его последующую обработку, чтобы сформировать структурированный вывод (который она и возвращает). После обработки всех входных текстов ❼ окончательный результат сохраняется в файл `result.csv` (если такой файл уже существует, то будет перезаписан).

4.3.5. Проверка

Код из листинга 4.2, можно найти в файле `listing2.py`, доступном по ссылке **Listing 2**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе. Там же можно скачать файл `biographies.csv`, в котором хранится небольшой набор данных для экспериментов с вашим экстрактором (он немного отличается от нашего сценария, но общедоступных данных о заявках по электронной почте мало). В файле содержатся биографии пяти знаменитостей, а также их имена — для каждого выделена одна строка. Перейдите в

каталог, содержащий файл `listing2.py` (и данные), и запустите код:

```
python listing2.py biographies.csv
    "имя|место рождения|дата рождения"
```

Первый параметр — набор данных (если он находится в другом каталоге, то скорректируйте путь соответственно). Второй параметр — список атрибутов для извлечения. Для разделения атрибутов снова используется вертикальная черта. Обратите внимание, что атрибуты идентифицируются только по их именам; нет необходимости ссылаться на предопределенные категории. Языковая модель способна понимать семантику атрибутов, основываясь только на имени.

После выполнения кода (которое не должно занимать больше минуты) результаты будут сохранены в файл `result.csv`. Например, выполнив код с данными из примера, вы получите следующую таблицу:

```
    имя     место рождения   дата рождения
0    Сергей Михайлович Брин         Москва       21 августа 1973
1      Мартин Лютер Кинг мл.           Атланта, Джорджия     15 января 1929
2   Энн Э. Воджицки     <N/A>     28 июля 1973
3               Мария     Саломея     Склодовская-Кюри     Варшава     7 ноября 1867
4      Алан Мэтисон Тьюринг           Мейда-Вейл, Лондон     23 июня 1912
```

Даже при запуске одного и того же кода с одними и теми же данными можно заметить небольшие различия (вызванные рандомизацией при генерации вывода модели). Каждая строка

в этом файле (кроме строки заголовка) представляет собой извлеченные данные. Мы извлекаем имя, место и дату рождения. Следовательно, предполагается, что из каждой биографии будет извлечено по одной строке (именно так и происходит в пробном запуске). Обратите внимание, что некоторые значения пропущены: во фрагменте биографии Энн Э. Воджицки не указано место рождения. Языковая модель реагирует соответствующе и вставляет вместо конкретного значения метку-заполнитель (`<N/A>`).

4.4. Кластеризация

Представьте, что по-прежнему работаете в компании Banana. Работа отличная, но есть одна проблема: ваш электронный почтовый ящик все так же завален письмами! Это не только заявки от желающих пройти летнюю стажировку (с ними мы разобрались в предыдущем разделе). Темы писем самые разные, и на прочтение всей значимой корреспонденции вы тратите много времени. Изучив письма подробнее, вы замечаете, что многие из них избыточны. Например, вы обращаете внимание, что в разных письмах обсуждаются одни и те же события в компании. На мгновение вы задумываетесь, а не применить ли код классификации текста (рассмотренный в разделе 4.2), чтобы классифицировать письма по нескольким категориям (например, связанным с конкретными событиями в компании). После этого можно будет прочитать всего несколько писем из каждой категории и получить полное представление о том, что происходит в компании. Увы, есть одна проблема: исчерпывающий список тем сложно составить и поддерживать, поскольку в ходе вашей работы они будут постоянно меняться. Вместо этого хорошо было бы автоматически группировать разные письма, имеющие общие

черты, например посвященные одному и тому же событию. Тогда вам не придется заранее придумывать список тем.

Таким образом, требуется сгруппировать похожие письма в кластеры. Это еще одна классическая задача обработки текста — его кластеризация. Если вы хотите объединить связанные текстовые документы, не указывая предварительно набор категорий, то можете использовать методы кластеризации.

Итак, в этом разделе вы увидите, как с помощью языковых моделей выполнять кластеризацию текста.

4.4.1. Обзор

Кластеризация — классический подход, используемый в информатике. Методы кластеризации появились намного раньше языковых моделей и расширенного анализа текста. Однако традиционно кластеризация фокусируется на элементах, которые выражены в виде векторов. Необходимо объединить (в один кластер) векторы, находящиеся на небольшом расстоянии друг от друга (и конечно, существуют различные показатели расстояния, которые можно применить к векторам). Но в нашем случае все немного иначе: в сценарии мы хотим собрать в один кластер похожие письма (или похожие текстовые документы в целом). Как же перейти от документов к векторам?

Ответ — использовать *эмбеддинги*. Эмбеддинг представляет собой текстовый документ в виде вектора (как правило, многомерного). Именно это нам и нужно! Разумеется, этот подход имеет смысл только в том случае, если мы преобразуем текстовые документы в векторы, которые отражают смысл содержания документов. В идеале нам нужно, чтобы документы с похожими векторами (то есть векторами с небольшим расстоянием согласно выбранному показателю

расстояния) имели и схожее содержание. А значит, нам не удастся использовать наивные методы для отображения текстовых документов в виде векторов. Вместо этого потребуется применить подход, который будет учитывать семантику текста и принимать ее во внимание при генерации векторного представления.

К счастью, языковые модели могут помочь! Поставщики наподобие OpenAI предлагают языковые модели, которые принимают текст в качестве входных данных и выдают эмбеддинги на выходе. Итак, имея коллекцию текстовых документов для кластеризации, можно вычислить для них эмбеддинги и применить к полученным векторам любой классический алгоритм кластеризации. Этот процесс показан на рис. 4.3. Далее мы обсудим, как его реализовать.

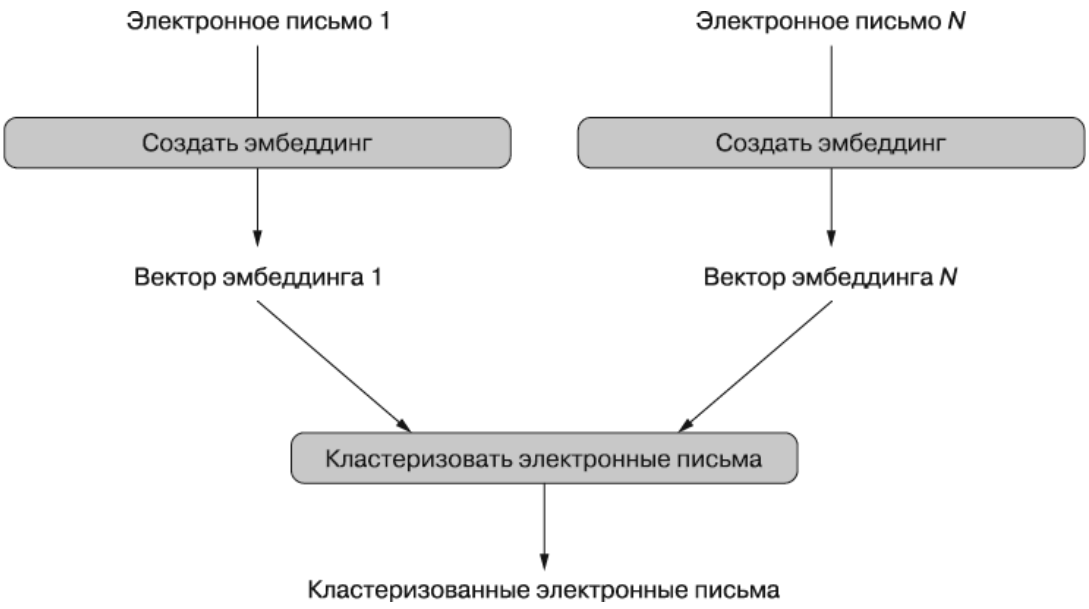


Рис. 4.3. Кластеризация электронных писем. Сначала мы вычисляем эмбеддинги для всех писем. Затем кластеризуем эти векторы, чтобы собрать письма с похожим содержанием в один кластер

4.4.2. Вычисление эмбеддингов

В примерах, которые приводились ранее, мы использовали эндпоинт `chat.completions` с OpenAI. Чтобы выполнить кластеризацию, вместо него возьмем эндпоинт `embeddings` OpenAI. Цель эмбеддинга — создать вектор, который сжимает семантику текста. Вычислить эмбеддинги можно с помощью разных моделей. Размерность вектора зависит от используемой модели.

Для кода, который приводится ниже, используется модель `text-embedding-ada-002`. Можете попробовать заменить ее другими моделями (список моделей OpenAI для вычисления эмбеддингов представлен на странице **https://platform.openai.com/docs/guides/embeddings**) и сравнить качество вывода.

Например, эмбеддинги для текстовых документов можно генерировать так:

```
import openai
client = openai.OpenAI()

response = client.embeddings.create(
    model='text-embedding-ada-002',
    input=text)
```

Вот фрагмент соответствующего ответа:

```
CreateEmbeddingResponse(
    data=[
        Embedding(embedding=[                          ❶ Эмбеддинг
            -0.005983137525618076, -0.000303583248751238, ...],
            index=0, object='embedding')],
    model='text-embedding-ada-002',                ❷ Модель, сгенерировавшая эмбеддинги
    object='list',
    usage=Usage(prompt_tokens=517, total_tokens=517)) ❸ Статистика использования
```

Фрагмент содержит значения только для первых нескольких размерностей вектора ❶ (тогда как полный вектор имеет более 1000 размерностей). Помимо эмбеддинга, в ответе есть

название модели ❷ и статистика использования ❸. В отличие от предыдущих случаев статистика использования относится только к количеству токенов в промпте (которое также является общим количеством обработанных токенов). В отличие от диалогового режима здесь языковая модель только считывает токены, но не генерирует их.

Самая важная часть для нас, конечно, сам эмбеддинг. Получить к нему доступ можно с помощью следующей команды:

```
response.data[0].embedding
```

В большинстве случаев одного вызова языковой модели должно быть достаточно для получения нужного эмбеддинга. Конечно, при вычислении эмбеддингов для большого количества писем время от времени придется сталкиваться с проблемами (то есть неудачными попытками соединения). Вот почему в окончательной версии функции эмбеддинга снова содержится механизм повторных попыток:

```
import openai
client = openai.OpenAI()

def get_embedding(text):
    for nr_retries in range(1, 4):
        try:
            response = client.embeddings.create(
                model='text-embedding-ada-002',
                input=text)
            return response.data[0].embedding
        except:
            time.sleep(nr_retries * 2)
```

```
        raise Exception('Не удается выполнить запрос к
    модели OpenAI!')
```

Имея в качестве входных данных текст, мы выполняем до трех попыток получения соответствующего эмбеддинга (увеличивая задержку между попытками после каждого сбоя). Именно эту функцию мы и будем использовать.

4.4.3. Кластеризация векторов

Для кластеризации векторов (представляющих документы) мы будем использовать алгоритм кластеризации *k*-средних. Это очень популярный алгоритм, который работает путем итеративного уточнения, к какому кластеру относится каждый вектор. В отличие от других алгоритмов кластеризации, для него необходимо заранее указать количество кластеров. В нашем тестовом сценарии это означает, что придется указать, насколько детальным должно быть разделение писем по их содержанию.

Как работает алгоритм k-средних

Алгоритм *k*-средних принимает в качестве входных данных набор элементов для кластеризации и целевое количество кластеров. Затем он итеративно перебирает элементы и уточняет, к какому кластеру они относятся, — и так до тех пор, пока не выполнится условие остановки (например, максимальное количество итераций или минимальные изменения в назначениях кластеров между последовательными итерациями). Алгоритм связывает каждый кластер с вектором-центроидом (представляющим собой центр этого кластера). При каждой итерации он распределяет векторы данных по кластерам, к векторам-центроидам

которых они расположены ближе всего. Затем он пересчитывает векторы, связанные с кластерами (путем усреднения по векторам всех элементов, в данный момент назначенных кластеру).

Мы будем использовать реализацию *k*-средних в библиотеке `scikit-learn`. Следуйте инструкциям в первом разделе этой главы и убедитесь, что библиотека установлена (импортируйте методы кластеризации, используя команду `from sklearn.cluster import KMeans`). После импорта библиотеки реализацию *k*-средних можно вызвать с помощью этого (краткого) фрагмента кода:

```
def get_kmeans(embeddings, k):
    kmeans = KMeans(n_clusters=k, init='k-means++')
    kmeans.fit(embeddings)
    return kmeans.labels_
```

В качестве входных данных функция принимает список эмбеддингов и количество целевых кластеров, а затем кластеризует эти векторы с использованием реализации *k*-средних. Результатом кластеризации являются метки, связанные с каждым эмбеддингом. Эти метки обозначают идентификатор связанного кластера.

4.4.4. Полный рабочий код кластеризации текста

В листинге 4.3 показан полный код для кластеризации текстовых документов с помощью эмбеддингов. Вы познакомитесь с функциями для вычисления эмбеддингов ❶ и их кластеризации ❷.

Листинг 4.3. Кластеризация текстовых документов с использованием языковых моделей

```
import argparse
import openai
import pandas as pd
import time

from sklearn.cluster import KMeans

client = openai.OpenAI()

def  get_embedding(text):                                    ❶
```

Вычисляет эмбеддинги

```
    """Вычислить эмбеддинг для входного текста.

    Аргументы:
        text: текст, для которого будет вычисляться
эмбеддинг.

    Выводит:
        Векторное представление входного текста.
    """
    for nr_retries in range(1, 4):
        try:
            response = client.embeddings.create(
                model='text-embedding-ada-002',
                input=text)
            return response.data[0].embedding
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')
```

```
def get_kmeans(embeddings, k):                    ❷
```

Кластеризует эмбеддинги

```
    """Кластеризовать эмбеддинги с использованием
метода k-средних.

    Аргументы:
        embeddings: эмбеддинги.
        k: количество получившихся кластеров.

    Выводит:
                ID кластеров в порядке следования
эмбеддингов.
    """
    kmeans = KMeans(n_clusters=k, init='k-means++')
    kmeans.fit(embeddings)
    return kmeans.labels_

if __name__ == '__main__':                        ❸
```

Считывает текст и записывает кластеры

```
    parser = argparse.ArgumentParser()
        parser.add_argument('file_path', type=str,
help='Путь к входному файлу')
      parser.add_argument('nr_clusters', type=int,
help='Количество кластеров')
    args = parser.parse_args()

    df = pd.read_csv(args.file_path)
```

```
    embeddings = df['text'].apply(get_embedding)
     df['clusterid'] = get_kmeans(list(embeddings),
args.nr_clusters)

    df.to_csv('result.csv')
```

Главная функция в этом листинге ❸ считывает данные из файла на диске. Мы снова предполагаем, что данные содержатся в файле формата `.csv`, и основное внимание уделяем столбцу `text`. Сначала мы перебираем текстовые документы и генерируем соответствующие эмбеддинги (вызывая уже знакомую нам функцию `get_embedding`). Затем кластеризуем эмбеддинги с помощью функции `get_kmeans`. Идентификаторы кластеров становятся дополнительным столбцом в получившейся таблице, которая записывается на диск.

4.4.5. Проверка в работе

Пришло время опробовать выполнить кластеризацию с помощью эмбеддингов! Код из листинга 4.3 (`listing3.py`), а также подходящий набор данных (`textmix.csv`) доступны по ссылкам (**Listing 3** и **Textmix.csv** соответственно), размещенным на сайте книги в разделе дополнительных материалов к главе. В наборе данных содержится смесь текстовых фрагментов из двух источников: сборника стихов и репозитория писем. Мы попытаемся разделить их с помощью кластеризации: предполагается, что письма и стихи будут распределены по разным кластерам.

Перейдите в каталог, содержащий код и данные, и введите в терминале следующую команду:

```
python listing3.py textmix.csv 2
```

Здесь `textmix.csv` — имя входного файла, а `2` — количество целевых кластеров (в этом конкретном случае число 2 кажется логичным, хотя в других сценариях правильно определить количество кластеров может быть сложнее). Результат будет сохранен в файле `result.csv`. Он содержит все столбцы из входного файла, а также дополнительный столбец с идентификатором кластера (поскольку мы используем только два кластера, значением идентификатора будет `0` или `1`). Если выполнить команду, то письма, скорее всего, попадут в один кластер, а стихи — в другой.

Возможно, вы захотите протестировать разные модели, чтобы увидеть различия во времени выполнения и в качестве результата. Вдобавок можно менять текст ввода и варьировать количество кластеров. Помимо этого, можно реализовать и другие сценарии использования эмбеддингов, упоминавшиеся в начале этого раздела. Например, как насчет реализации интерфейса поиска, который сопоставляет утверждение на естественном языке с наиболее тесно связанным с ним документом (сравнивая эмбеддинги вопросов и документов)?

4.4.6. Другие варианты применения эмбеддингов

До сих пор мы использовали векторы для идентификации похожих документов с помощью кластеризации. Но это не единственный вариант применения эмбеддингов. Например, они часто используются для облегчения поиска текстовых документов, связанных с вопросом на естественном языке. В этом случае эмбеддинг, связанный с вопросом, сравнивается с эмбеддингами, связанными с документами. Документы с похожими векторами с большей вероятностью окажутся полезны при ответе на вопрос.

Например, мы надеемся, что эмбеддинги для вопроса «Что такое модель-трансформер?» и текста «Трансформер — это архитектура нейронной сети, часто используемая для языковых моделей» будут похожи, поскольку их темы связаны. Если это так, то можно идентифицировать документ, наиболее релевантный вопросу, сравнив векторы. Точнее, эмбеддинги вычисляются один раз для каждого документа, который может быть полезен для ответа на вопросы. Затем всякий раз, когда поступает новый вопрос, вычисляется связанный с ним эмбеддинг и извлекаются документы с похожими векторами. Далее мы можем сгенерировать ответ на основе этих документов.

Другой вариант использования эмбеддингов — обнаружение аномалий. Чтобы выявить в наборе текстовые документы, разительно отличающиеся от других из того же набора, можно сравнить их эмбеддинги. Опять же, векторы потребуется вычислить только один раз для каждого документа. Поступая так, мы избегаем необходимости использовать языковые модели для сравнения документов. Вместо этого мы просто сравниваем эмбеддинги (что занимает очень мало времени).

В итоге, хотя мы и уделяем основное внимание кластеризации, существует множество сценариев использования эмбеддингов. Поэтому стоит разобраться, как их генерировать и применять.

Резюме

- Языковые модели можно использовать непосредственно для анализа текстовых данных.

- Промпты обычно содержат текст для анализа и инструкции. В инструкциях описывается решаемая задача, а также

указывается формат вывода.

- Диалоговый режим можно использовать для классификации, извлечения информации и ответов на вопросы.
- Изменение формата необработанного вывода модели может потребовать последующей обработки.
- Языковые модели способны преобразовывать текст в эмбеддинги. Вы можете создавать эмбеддинги с помощью эндпоинта `embeddings`. Сравнение эмбеддингов — относительно эффективный метод.
- Эмбеддинги можно использовать для кластеризации, поиска и обнаружения аномалий.

Литература

1. *Katz D.M., Bommarito M.J., Gao S. et al.* GPT-4 Passes the Bar Exam // Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 2024. — № 382 (2270). — Pp. 1–17.

---

7 Здесь и далее docstrings, строки вывода и комментарии переведены для удобства восприятия кода. — *Примеч. ред.*

5. Анализ структурированных данных

В этой главе

- Перевод вопросов в запросы.
- Создание интерфейсов на естественном языке.
- Анализ табличных данных.
- Анализ графовых данных.

Значительная часть информации в мире хранится в виде структурированных данных. Структурированными, по сути, являются данные, хранящиеся в стандартизированном формате. Например, популярными типами структурированных данных являются табличные данные (скажем, данные, которые можно найти в электронной таблице Excel), а также данные, описывающие сущности и их связи в виде графов (скажем, набор данных, описывающий социальную сеть).

Инструменты для обработки структурированных данных доступны уже много десятилетий. Такие данные имеют стандартизированный формат, оптимизированный для облегчения обработки компьютерами. Так зачем же использовать для обработки этих данных большие языковые модели? Проблема имеющихся инструментов, предназначенных для этой цели, — их интерфейс. Как правило, каждый инструмент (или по крайней мере каждая категория инструментов для определенных типов структурированных данных) поддерживает собственный формальный язык запросов.

Используя этот язык, пользователи, как правило, могут выполнять широкий спектр аналитических операций со структурированными данными. Но на изучение таких языков запросов требуется время! Не правда ли, было бы замечательно иметь возможность отправлять запросы ко всем системам на одном языке, в идеале на естественном (например, русском)?

Именно здесь могут помочь большие языковые модели. Они способны переводить вопросы, заданные на естественном языке, на множество формальных языков. Поэтому их можно использовать в качестве универсального интерфейса к различным инструментам анализа данных, поддерживающим широкий спектр типов структурированных данных. В этой главе мы создадим интерфейсы, позволяющие писать запросы на естественном языке к структурированным данным различных типов. Такие интерфейсы позволят нам (или другим людям) анализировать данные, вводя вопросы на естественном языке. Затем система переведет наши вопросы в формальные запросы, выполнит их и представит результаты.

5.1. Краткое содержание главы

В этой главе мы создадим несколько интерфейсов запросов на естественном языке. В общем случае интерфейс запросов на естественном языке обрабатывает запросы к данным, сформулированные на естественном языке. В главе рассматриваются различные типы структурированных данных. Сначала мы создадим интерфейсы на естественном языке, которые будут обрабатывать запросы о табличных данных. Затем создадим интерфейс, обрабатывающий запросы о графах.

Принцип одинаков в обоих случаях. Предполагается, что данные обрабатываются с использованием характерного для

конкретного типа данных инструмента. Например, для табличных данных применяется реляционная система управления базами данных (РСУБД). Для анализа графов используется система управления графовыми данными. Затем с помощью большой языковой модели вопросы на естественном языке переводятся на язык запросов, поддерживаемый конкретным инструментом. Для РСУБД это, как правило, язык структурированных запросов (Structured Query Language, SQL). Системы управления графовыми базами данных поддерживают различные специфичные для графовых данных языки запросов. Мы будем использовать язык запросов Cypher (описывающий аналитические операции с графовыми данными). Более подробно оба языка обсудим в следующих разделах.

Чтобы переводить вопросы в формальные запросы, языковой модели нужен доступ к вопросу (разумеется), а также информация о целевом языке (например, хотим ли мы писать запросы на SQL или Cypher?) и структуре запрашиваемых данных. К примеру, для структуры табличных данных характерны, помимо прочего, имена таблиц, заголовки столбцов, которые появляются в этих таблицах (предположительно, они помогают понять, какова семантика табличных данных), и типы данных каждого столбца (хранятся ли в нем целочисленные значения или строки?). После того как мы предоставим языковой модели все фрагменты информации, добавив их в запрос, модель должна быть способна создать формальный запрос, отражающий семантику нашего вопроса.

Этот запрос можно обработать с помощью специализированного инструмента и получить результат. При условии, что перевод запроса был корректным, результат будет

представлять собой ответ на исходный вопрос. Процесс показан на рис. 5.1.

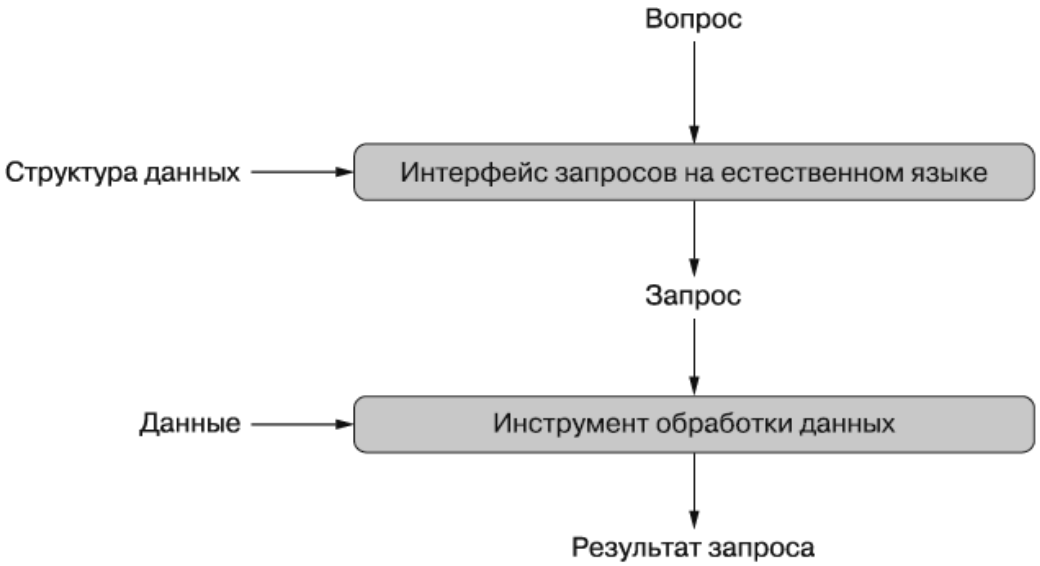


Рис. 5.1. Интерфейс запросов на естественном языке переводит вопросы, заданные на естественном языке, в формальные запросы, принимая во внимание структуру данных. Затем с помощью специализированного инструмента проводится обработка формальных запросов к данным и генерируется результат запроса

Зачем использовать внешние инструменты

Мы выяснили, что языковые модели способны решать разнообразные задачи. Так почему бы просто не использовать их для анализа структурированных данных напрямую? Почему мы применяем для этого внешние инструменты, а языковую модель используем лишь в качестве переводчика?

Основная причина — эффективность. Использовать большие языковые модели дорого, и, по крайней мере в больших наборах данных, размер данных может легко превысить максимальный размер входных данных модели. Следовательно, более разумное решение — использовать

существующие инструменты, способные работать с большими структурированными наборами данных.

ПРЕДУПРЕЖДЕНИЕ В следующих разделах языковые модели используются для написания команд обработки данных. Несмотря на то что во многих вариантах использования модели показывают удивительно высокую эффективность, не стоит рассчитывать на абсолютную корректность их выводов в каждом отдельно взятом сценарии. В одних случаях модели могут писать некорректные запросы. В других — могут написать команды для изменения или удаления ваших данных или изменения конфигурации вашей системы. Всегда сохраняйте резервную копию важных данных, прежде чем разрешать доступ к ним через языковые модели.

5.2. Интерфейс запросов на естественном языке для анализа продаж игр

Снова вернемся в компанию Banana. Выяснилось, что ваш начальник — заядлый любитель компьютерных игр. Он не только любит играть по вечерам, но и обожает анализировать данные о компьютерных играх. Получив на днях набор данных о продажах компьютерных игр, ваш начальник хочет извлечь из него статистическую информацию. Для анализа табличных данных, как правило, используют язык SQL, но начальник толком не умеет писать SQL-запросы. Зная, что вы эксперт в том, что касается языковых моделей и анализа данных, он спрашивает, возможно ли создать интерфейс, который будет переводить вопросы о компьютерных играх в соответствующие SQL-запросы.

Подумав, вы понимаете, что это интересный проект, который можно легко реализовать с помощью языковых моделей. Начнем!

5.2.1. Настройка базы данных SQLite

Прежде чем мы сможем анализировать табличные данные с помощью SQL-запросов, нужно загрузить эти данные в РСУБД — инструмент для эффективной обработки табличных данных. В этом подразделе вы узнаете, как загрузить данные о компьютерных играх в популярную РСУБД SQLite.

В первую очередь убедитесь, что у вас установлена SQLite. Здесь мы будем пользоваться SQLite версии 3. Проверить, установлена ли у вас данная версия SQLite, можно, набрав в терминале следующую команду:

```
sqlite3 --version
```

Если появится сообщение, подобное приведенному ниже, то дополнительная установка не требуется:

```
3.33.0 2020-08-14 13:23:32 fca8...
```

Если отобразится сообщение об ошибке, то перейдите на сайт **www.sqlite.org/download.html**. Выберите версию, совместимую с вашей операционной системой, скачайте все соответствующие файлы и следуйте инструкциям по установке SQLite. В случае возникновения проблем зайдите на сайт книги, перейдите в раздел дополнительных материалов к главе 5 и щелкните на ссылке **SQLite Installation**. На открывшейся странице вы найдете подробные инструкции по установке SQLite на различных платформах. После этого запустите предыдущую команду, чтобы убедиться, что РСУБД SQLite установлена корректно.

Далее с помощью SQLite необходимо создать реляционную базу данных. Такая база, по сути, представляет собой коллекцию таблиц данных. Можете считать каждую такую таблицу обычной электронной таблицей. В ней есть именованные столбцы, связанные с типом данных, и строки (возможно, их много), которые содержат значения для каждого из столбцов. В табл. 5.1 показан пример таблицы с информацией о видеоиграх, где каждая строка представляет одну игру. В таблице четыре столбца. Три из них («Название», «Платформа» и «Жанр») содержат строки (то есть текст). В столбце «Год» приводятся цифры — год выпуска игры.

Таблица 5.1. Пример таблицы данных. Каждая строка таблицы описывает одну видеоигру

Название	**Платформа**	**Год**	**Жанр**
Wii Sports	Wii	2006	Спортивная игра
«Супербратья Марио»	NES	1985	Платформенная игра
Mario Kart Wii	Wii	2008	Гонки
Wii Sports Resort	Wii	2009	Спортивная игра
Pokemon Red/Pokemon Blue	GB	1996	Ролевая игра

В этой таблице показан небольшой фрагмент набора данных, который доступен по ссылке **Games**, размещенной на сайте книги, в разделе дополнительных материалов к главе 5. Мы используем этот набор в следующих разделах, чтобы создать интерфейс запросов на естественном языке, позволяющий пользователям анализировать данные с помощью команд на естественном языке. Но сначала необходимо загрузить эти данные в систему базы данных SQLite.

СОВЕТ Мы выполним все действия, необходимые для загрузки табличных данных в SQLite. Это полезно, если требуется загружать другие данные, помимо обсуждаемых здесь. Если вы не планируете создавать собственную базу, то можете пропустить этот подраздел и скачать файл games.db по ссылке Games SQLite, доступной на сайте книги в разделе дополнительных материалов к главе.

Запустим интерфейс командной строки SQLite. Введите в терминале следующую команду и нажмите клавишу **Enter**:

```
sqlite3 games.db
```

Команда создаст новую базу данных, хранящуюся в файле `games.db`. Одновременно она откроет интерфейс командной строки SQLite. Теперь можно дать инструменту SQLite указание загрузить набор данных, который мы будем использовать в следующих разделах. Сначала необходимо предоставить SQLite некоторые сведения о структуре данных. Нам нужно загрузить расширенную версию табл. 5.1, содержащую дополнительные столбцы. Чтобы описать структуру данных, выполните следующую команду:

```
CREATE TABLE games(                                    ❶ Имя таблицы
rank int, name text, platform text,                    ❷ Список столбцов таблицы
year int, genre text, publisher text, americasales numeric,
eusales numeric, japansales numeric, othersales numeric,
```

Эта команда описывает структуру единственной таблицы `games` ❶ (ведь в ней будет храниться информация о видеоиграх); в скобках мы указываем полный список столбцов таблицы (разделенных запятыми) ❷ . Каждый столбец определяется по имени столбца (например, `rank`, `name` или `genre`), за которым следует обозначение типа столбца. Например, `int` означает, что столбец содержит целые числа, а

`text` — что в столбце хранятся текстовые данные. Все строки таблицы должны предоставлять значения для каждого столбца в соответствии с его типом данных.

После определения структуры таблицы можно загрузить в нее данные из файла `.csv`, доступного по ссылке **Games**, размещенной на сайте книги в разделе дополнительных материалов. Скачайте файл, если еще не сделали этого. Для следующих команд предположим, что файл сохранен в папке `/Downloads/videogames.csv`. Просто замените этот путь на путь к файлу в вашей системе. Загрузите данные с помощью таких команд (все еще в интерфейсе SQLite):

```
.mode csv                                  ❶ Устанавливает режим CSV
.import /Downloads/videogames.csv games    ❷ Импортирует данные
```

Первая команда подготавливает SQLite к загрузке данных из файла `.csv` (применима к загружаемому файлу) ❶ . Вторая команда ❷ импортирует данные: первый параметр — это путь к файлу, из которого требуется загрузить данные, а второй — имя таблицы данных, в которую нужно загрузить файл. В этом случае мы ссылаемся на таблицу, структуру которой определили ранее (`games`). Чтобы проверить, были ли данные загружены успешно (должны, если не появлялось сообщение об ошибке), выполните в SQLite следующую команду:

```
SELECT count(*) FROM games;
```

Если были загружены все данные, то должен отобразиться результат `16599`. Если отображается меньшее значение, проверьте сообщения об ошибках в выводе SQLite. Выйти из консоли SQLite можно с помощью команды `.quit` (не забудьте добавить точку в начале команды, чтобы она сработала). Далее предполагается, что данные были загружены и хранятся в файле SQLite `games.db`.

5.2.2. Основы SQL

Безусловно, с помощью языковой модели можно перевести вопросы в SQL-запросы (которые поймет база данных SQLite). Но стоит ли доверять ее переводам? Обдумав этот вопрос, вы понимаете, что вам не помешало бы освоить хотя бы основы SQL, чтобы проверить вывод языковой модели, прежде чем представлять интерфейс своему начальнику. Этим мы и займемся в текущем подразделе. Разумеется, подробное описание SQL не входит в задачи этой книги. Если оно вам нужно, то вы можете найти его на сайте **www.databaselecture.com**. А здесь мы обсудим основы SQL, которые помогут реализовать интерфейс запросов на естественном языке.

SQL-запросы используются для анализа таблиц данных. Результаты могут быть получены из одной таблицы или путем объединения данных из нескольких таблиц. В нашем примере базы данных, созданном в прошлом подразделе, — всего одна таблица. Используя SQL-запросы, можно, например, подсчитывать строки с определенными свойствами (скажем, все игры одного издателя), фильтровать данные (допустим, показывать только игры, выпущенные в 2017 году) или выполнять различные операции агрегирования (к примеру, для каждого издателя рассчитывать среднюю выручку за одну игру).

В общем случае SQL-запрос описывает таблицу, которую нужно сгенерировать (используя данные, уже имеющиеся в базе). Большинство примеров запросов в этом разделе имеют следующую структуру:

```
SELECT [Столбцы или агрегаты]  ❶ Предложение SELECT
FROM [Список таблиц]           ❷ Предложение FROM
WHERE [Список условий]         ❸ Предложение WHERE
```

Предложение FROM ❷ (код, идущий после ключевого слова FROM и до ключевого слова WHERE) описывает исходные данные, используемые для анализа. Например, предложение FROM может содержать список имен таблиц, разделенных запятыми. В нашем примере базы данных есть только одна таблица для обработки (в сложных запросах одно и то же имя таблицы может появляться в предложении FROM несколько раз, создавая разные копии одной и той же таблицы). Запросы, которые вы увидите в следующих разделах, будут содержать единственную запись в предложении FROM: таблицу games.

Предложение WHERE ❸ определяет предикаты для таблиц, которые появляются в предложении FROM. Например, оно может содержать условие, ограничивающее выборку играми от конкретного издателя. Предложение WHERE может содержать как простые условия (которые могут быть выражены равенством или неравенством для столбца таблицы), так и сложные (соединяющие несколько простых условий с помощью операторов AND или OR). Эти условия используются для фильтрации строк из таблиц в предложении FROM. Строки, которые не удовлетворяют условию в предложении WHERE, отбрасываются и не появляются в результатах запроса.

Наконец, в предложении SELECT ❶ задаются столбцы желаемой итоговой таблицы. Точнее, мы указываем список определений столбцов, разделенных запятыми. Столбцы могут определяться по имени столбца (столбец, имеющийся в одной из таблиц в предложении FROM) или более сложным выражением: например, арифметическим выражением, связывающим несколько столбцов. Как вариант, в предложении SELECT можно указать агрегаты, например count(*), который подсчитывает количество строк. Результат запроса содержит

таблицу с указанными столбцами, значения в которых заполняются в соответствии с определением столбца.

Допустим, мы хотим подсчитать количество видеоигр, изданных в 2017 году. В этом случае результат запроса должен содержать один столбец с агрегатом `count`. Кроме того, в предложении `WHERE` должно быть прописано условие фильтрации, ограничивающее выборку видеоиграми за 2017 год. Наше предложение `FROM`, естественно, будет содержать имя единственной таблицы в базе данных (`games`). Приведенный ниже запрос генерирует желаемый результат:

```
SELECT Count(*)
FROM games
WHERE year = 2017
```

Чтобы немного усложнить задачу, предположим, что нам нужно рассчитать суммарное количество продаж в Европе и Японии всех видеоигр, выпущенных в 2017 году конкретным разработчиком (Activision). Желаемый результат запроса содержит два столбца: название игры и количество продаж. В таблице данных есть два столбца с продажами: в Европе и в Японии (`eusales` и `japansales`), поэтому согласно нашему определению таблицы в предыдущем подразделе желаемый итоговый столбец можно описать, сложив их данные (`eusales + japansales`). Поскольку выборку нужно ограничить видеоиграми за 2017 год, выпущенными компанией Activision, то в предложении `WHERE` можно использовать сложный предикат: `year = 2017 AND publisher = 'Activision'`. Обратите внимание на одинарные кавычки по обеим сторонам названия Activision — они необходимы потому, что в условиях запроса строковые значения заключаются в одинарные кавычки, а

числа — нет. Приведенный ниже запрос генерирует желаемый результат:

```
SELECT name, eusales + japansales
FROM games
WHERE year = 2017 AND publisher = 'Activision'
```

При необходимости, указывая агрегатные функции, можно вычислять эти агрегаты для разных групп строк, определяемых общими значениями в конкретных столбцах. Для этого добавим к предыдущему шаблону запроса заключительное предложение `GROUP BY`, за которым будет следовать разделенный запятыми список столбцов, используемых для формирования групп. Например, допустим, нам нужно посчитать количество продаж видеоигр каждого жанра (скажем, стратегии или экшена) отдельно. С помощью следующего запроса можно вернуть одну строку с агрегатами для каждой категории игр (обратите внимание, что мы также добавляем в предложение `SELECT` столбец `genre`, чтобы можно было связать числа с нужным жанром):

```
SELECT genre, name, eusales + japansales
FROM games
WHERE year = 2017 AND publisher = 'Activision'
GROUP BY genre
```

Краткого пояснения SQL, предоставленного в этом подразделе, явно недостаточно для написания собственных SQL-запросов, за исключением нескольких простых случаев. Однако нам на самом деле и не нужно писать эти запросы — ведь мы планируем, что за нас это будет делать языковая модель! Вводный материал должен дать вам общее представление о запросах, сгенерированных языковой

моделью. А наблюдая за тем, как она преобразует вопросы в запросы, вы сможете легко сделать первые шаги в изучении SQL. В следующем подразделе мы начнем с создания простого переводчика, преобразующего вопросы в SQL-запросы к базе данных `games`.

5.2.3. Обзор

Итак, вы получили базовое представление о SQL, достаточное для проверки результатов работы языковой модели. И теперь мы приступим к разработке переводчика текста в SQL-запросы. Работать с переводчиком будем через командную строку. Введем в качестве входных данных вопрос, и в идеале в результатах вывода хотелось бы получить перевод вопроса в SQL-запрос. Запустив полученный запрос в SQLite, мы должны получить ответ на свой исходный вопрос. Конечно, интерфейс подразумевает работу вручную и не очень удобен в использовании. В идеале хотелось бы, чтобы он автоматически выполнял запросы и показывал соответствующие результаты прямо в интерфейсе запросов. Мы создадим такой интерфейс в следующем разделе. Сейчас просто сосредоточимся на основной проблеме перевода вопросов в запросы. Кроме того, на данный момент наша единственная цель — переводить вопросы о компьютерных играх. Поэтому нужно строго прописать в коде структуру целевой базы данных. Опять же, мы обобщим этот подход в следующем проекте.

Внутренняя логика перевода входного вопроса такова: сначала сформируем промпт. В нем описывается задача перевода и содержатся все актуальные для перевода детали (например, структура целевой базы данных). После отправки этого промпта языковой модели в большинстве случаев должен получиться корректно переведенный SQL-запрос. Возможно, придется еще немного потрудиться, чтобы вычленить из

длинного ответа языковой модели сам запрос. Давайте обсудим действия более подробно, начиная с промптов.

5.2.4. Генерация промптов для перевода из текста в код SQL

Какую информацию необходимо передать языковой модели, чтобы она выполнила перевод успешно? Очевидно, требуется добавить вопрос, который следует перевести. Кроме того, нужно указать целевую систему (SQLite) и описать структуру целевой базы данных. Пока мы строго прописываем в коде структуру базы. Можно просто предоставить языковой модели определение таблицы (`create table...`), использовавшееся в подразделе 5.2.1. Языковая модель поймет, как эта команда соотносится со структурой таблицы. Мы отправляем модели промпт с инструкциями по переводу, содержащий все упомянутые ранее типы информации. Получив его, языковая модель должна быть способна создать соответствующий SQL-запрос.

Используем следующий шаблон промпта:

```
База данных:                                  ❶ Описание базы данных
CREATE TABLE games(rank int, name text, platform text,
year int, genre text, publisher text, americasales numeric,
eusales numeric, japansales numeric, othersales numeric,
globalsales numeric);
Перевести вопрос в SQL-запрос:               ❷ Описание задачи
[Вопрос]                                      ❸ Вопрос для перевода
```

В этом шаблоне содержатся все фрагменты данных, упомянутые ранее. Он описывает целевую базу данных ❶, предоставляя SQL-команды, используемые для создания связанных таблиц (в данном случае — одной таблицы). Обратите внимание, что это не метка-заполнитель, так как здесь интерфейс запросов должен работать только с одной базой данных (структуру которой мы строго прописываем в коде в шаблоне). Далее, шаблон промпта содержит описание задачи ❷: цель — переводить вопросы в SQL-запросы. Наконец,

в шаблоне есть вопрос для перевода ❸ . Здесь применяется метка-заполнитель (в квадратных скобках). Она позволяет использовать один и тот же шаблон промпта для различных вопросов, которые начальник может задать о данных.

Этот код генерирует промпты в соответствии с шаблоном, приведенным выше:

```
def create_prompt(question):
    parts = []

    ❶ Добавляет описание базы данных
    parts += ['База данных:']
    parts += ['create table games(rank int, name text, platform text,']
    parts += ['year int, genre text, publisher text, americasales numeric,']
    parts += ['eusales numeric, japansales numeric, othersales numeric,']
    parts += ['globalsales numeric);']

    ❷ Добавляет описание задачи
    parts += ['Перевести вопрос в SQL-запрос:']
    parts += [question]      ❸ Добавляет вопрос для перевода
    return '\n'.join(parts)  ❹ Возвращает объединенную строку
```

Получив в качестве входных данных вопрос для перевода, код добавляет описание базы данных ❶, затем инструкции для перевода ❷ и, наконец, вопрос для перевода ❸ . Результатом является объединение всех частей промпта ❹.

5.2.5. Полный код

В листинге 5.1 содержится полный код для нашего интерфейса запросов на естественном языке. В нем используется функция генерации промпта ❶ , о которой говорилось выше, а также функция вызова языковой модели ❷ , уже знакомая вам по предыдущим главам.

Листинг 5.1. Перевод вопросов о видеоиграх в SQL-запросы

```
import argparse
import openai
import re
import time
```

```
client = openai.OpenAI()

def  create_prompt(question):                                    ❶
```

Генерирует промпт для перевода

```
    """Сгенерировать промпт для перевода вопроса в
SQL-запрос:

    Аргументы:
        question: вопрос о данных на естественном
языке.
    Выводит:
        Промпт для перевода вопроса.
    """
    parts = []
    parts += ['База данных:']
     parts += ['create table games(rank int, name
text, platform text,']
      parts += ['year int, genre text, publisher
text, americasales numeric,']
    parts += ['eusales numeric, japansales numeric,
othersales numeric,']
    parts += ['globalsales numeric);']
    parts += ['Перевести вопрос в SQL-запрос:']
    parts += [question]
    return '\n'.join(parts)

def call_llm(prompt):                                            ❷
```

Вызывает языковую модель

```
    """Отправить запрос языковой модели и вывести
ответ.
```

```
    Аргументы:
        prompt: ввести запрос к языковой модели.
    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                    {'role':'user',
'content':prompt}
                    ]
                )
                                                return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')

if __name__ == '__main__':                          ❸
```

Считывает запрос и переводит его в формат SQL

```
    parser = argparse.ArgumentParser()
         parser.add_argument('question',  type=str,
help='Вопрос об играх')
    args = parser.parse_args()
```

```
    prompt = create_prompt(args.question)  ❹
```

Генерирует промпт

```
    answer = call_llm(prompt)                ❺
```

Генерирует ответ

❻ Извлекает SQL-запрос из ответа

```
    query = re.findall('```sql(.*)```', answer,
re.DOTALL)[0]

    print(f'SQL: {query}')
```

В этом фрагменте кода мы получаем вопрос о видеоиграх в качестве аргумента командной строки ❸ . Используя введенный вопрос, он генерирует промпт ❹ , который инструктирует языковую модель перевести вопрос в SQL-запрос. Он отправляет промпт в языковую модель и получает ее ответ ❺.

Необработанный ответ от языковой модели GPT-4o, помимо SQL-запроса, который нас интересует, обычно содержит еще и пояснения. Чтобы получить только запрос, нужно извлечь его из необработанного ответа ❻ . Здесь воспользуемся тем, что GPT-4o помещает SQL-запросы между маркерами ```sql и ``` (если вы взаимодействуете с моделями GPT через веб-интерфейс ChatGPT, то все, что находится между этими маркерами, отображается как блок кода). Регулярное выражение ```sql(.*)``` находит SQL-запрос между маркерами, используя функцию Python `re.findall` для возврата списка совпадений с этим регулярным выражением (параметр `re.DOTALL` необходим, чтобы гарантировать, что точка соответствует всем символам, в том числе новой строки, которые могут появляться в SQL-запросах). Мы используем в качестве запроса первое совпадение (то есть мы допускаем, что

функция вернет хотя бы одно совпадение и что первое из них будет корректным).

5.2.6. Проверка в работе

Итак, пора протестировать наш переводчик из текста в SQL! В терминале перейдите в каталог, содержащий код на Python. Предположим, что код сохранен в файле `listing1.py` (доступен по ссылке **Listing 1**, размещенной на сайте книги в разделе дополнительных материалов к этой главе). Выполните следующую команду, чтобы спросить, сколько игр хранится в базе данных:

```
python listing1.py "Сколько игр хранится?"
```

В результате должен получиться следующий SQL-запрос:

```
SELECT COUNT(*) FROM games;
```

Корректен ли он? Выясним это: в терминале перейдите в репозиторий, содержащий файл базы данных SQLite (`games.db`). Затем откройте базу через интерфейс командной строки SQLite:

```
sqlite3 games.db
```

Теперь наконец можно протестировать запрос, сгенерированный нашим переводчиком из текста в SQL. Введите запрос и нажмите клавишу **Enter**. Вы должны получить количество видеоигр, сохраненных в базе данных: 16 599.

Полагаю, вы захотите протестировать и другие запросы. Например, выяснить, удастся ли подсчитать количество видеоигр, выпущенных конкретными издателями или относящихся к определенному жанру! Скорее всего, для

большинства вопросов, касающихся этой простой базы данных, модель GPT-4o окажется вполне способна предоставить точный перевод. Ваш начальник будет доволен.

Интерфейс, который мы создали в этом разделе, все еще имеет несколько ограничений.

- Во-первых, программу приходится перезапускать для каждого нового вопроса.
- Во-вторых, необходимо вручную копировать и выполнять каждый переведенный запрос в интерфейсе системы баз данных.
- В-третьих, и это самое главное, если вы когда-нибудь захотите поработать с другим набором данных, то шаблон промпта придется менять вручную.

В следующем разделе вы узнаете, как преодолеть эти ограничения.

5.3. Универсальный интерфейс запросов на естественном языке

Ваш начальник доволен новым интерфейсом на естественном языке и регулярно делится с вами любопытными сведениями о продажах компьютерных игр. Однако вы поневоле задумываетесь: полностью ли раскрывается потенциал подхода в рамках лишь этого сценария использования? Например, отделу кадров в компании Banana регулярно приходится иметь дело со сложными задачами, касающимися таблиц, в которых хранится информация о сотрудниках. Нельзя ли обобщить наш интерфейс естественного языка, чтобы помочь и кадровикам?

В этом разделе мы расширим интерфейс запросов на естественном языке, чтобы он мог работать с любой базой данных и при этом не приходилось вносить изменения в код. Более того, мы сделаем интерфейс более удобным, выполняя переведенные запросы напрямую и избегая перезапусков между разными вопросами по одним и тем же данным.

5.3.1. Выполнение запросов

Для начала посмотрим, как можно выполнять переведенные запросы напрямую из Python. Это позволит избежать утомительного копирования запросов из одного интерфейса в другой. В Python запросы к базе данных SQLite можно выполнять с помощью библиотеки `sqlite3`.

Предположим, что путь к файлу базы данных хранится в переменной `data_path`. Чтобы выполнять запросы к этой базе данных, необходимо сначала создать соединение:

```
import sqlite3


with sqlite3.connect(data_path) as connection:
    ...
```

Теперь можно выполнять SQL-запросы к базе данных, используя объект `connection`. Предположим, что SQL-запрос, который необходимо выполнить, хранится в переменной `query`. После подключения к базе данных мы сначала создаем объект `cursor` (позволяющий выполнять запросы и получать результаты), а затем используем его для выполнения запроса:

```
import sqlite3
```

```
with sqlite3.connect(data_path) as connection:
    cursor = connection.cursor()
    cursor.execute(query)
```

После выполнения, чтобы получить все строки результата, мы вызываем метод `cursor.fetchall()`. Мы соберем все вместе в функцию, которая принимает в качестве входных данных два параметра — путь к базе данных и запрос для выполнения в ней, — возвращая результат запроса в виде строки:

```
def process_query(data_path, query):
    with sqlite3.connect(data_path) as connection:    ❶ Подключается к базе данных
        cursor = connection.cursor()                  ❷ Создает объект cursor
        cursor.execute(query)                         ❸ Выполняет запрос
        table_rows = cursor.fetchall()                ❹ Извлекает результат запроса
        table_strings = [str(r) for r in table_rows]  ❺ Преобразует в список строк
        return '\n'.join(table_strings)               ❻ Объединяет строки результата
```

После подключения к целевой базе данных ❶ функция создает объект `cursor` ❷, выполняет входящий запрос ❸ и извлекает результат запроса ❹. Завершив преобразование кортежей результата в строковое представление ❺, мы объединяем полученные строки, разделяя их символами новой строки ❻.

5.3.2. Извлечение структуры базы данных

Нам требуется интерфейс, который будет работать с любыми базами SQLite, не вынуждая нас вносить изменения в код. Значит, нужно научиться автоматически извлекать структуру текущей базы данных (информацию о ее таблицах и столбцах).

Для SQLite структуру базы данных можно извлечь, выполнив SQL-запросы. Они обращаются к специальной *таблице схемы* (schema table). Она создается автоматически (то есть ее не нужно создавать вручную). Среди прочего она содержит SQL-команды, которые использовались для создания

других таблиц в базе. Их можно использовать как краткое описание структуры базы, подходящее в качестве входных данных для языковой модели.

Доступ к таблице схемы можно получить по ее имени — `sqlite_master`. Она содержит столбец `sql`, в котором хранится информация о запросах, использованных для создания объектов внутри базы данных. В частности, нас интересуют SQL-команды, которые использовались для создания таблиц. Эти запросы содержат ключевую для перевода информацию, в том числе имена и типы столбцов таблицы, которую они создают. Этот запрос извлекает все SQL-инструкции, использованные для создания таблиц в текущей базе данных:

```
select sql from sqlite_master where type = 'table';
```

Следовательно, достаточно просто выполнить этот запрос из кода Python. К счастью, мы уже выяснили, как это сделать. Получив путь к базе данных, функция (см. ниже) возвращает текст с описанием запросов, использовавшихся для создания всех таблиц в базе:

```
import sqlite3

def get_structure(data_path):
    with sqlite3.connect(data_path) as connection:  ❶ Подключается к базе данных
        cursor = connection.cursor()                ❷ Создает объект cursor

        ❸ Обращается к таблице схемы
        cursor.execute("select sql from sqlite_master where type = 'table';")
        table_rows = cursor.fetchall()              ❹ Извлекает результаты
        table_ddls = [r[0] for r in table_rows]     ❺ Извлекает тексты SQL-запросов
        return '\n'.join(table_ddls)                ❻ Объединяет тексты SQL-запросов
```

И снова мы создаем соединение ❶ и соответствующий объект `cursor` ❷. Далее выполняем запрос к таблице схемы, чтобы получить все SQL-запросы, использованные для создания таблиц в текущей базе данных ❸. Мы извлекаем результаты ❹ и получаем тексты SQL-запросов из результата запроса ❺. Обратите внимание, что эта часть функции немного

отличается от общей функции для выполнения запросов, о которой шла речь в предыдущем разделе. Извлекая значения первого (и для этого конкретного запроса — единственного) поля каждой строки, мы избавляемся от ненужных разделителей между строками, которые в противном случае появились бы в выводе (а потом — в запросах). Результатом является объединение всех строк вывода ❻.

5.3.3. Полный код

В листинге 5.2 представлен полный код универсального интерфейса запросов на естественном языке (файл доступен по ссылке **Listing 2**, которую можно найти на сайте книги в разделе дополнительных материалов к этой главе).

Листинг 5.2. Универсальный интерфейс для преобразования текста в SQL-запрос

```
import argparse
import openai
import re
import sqlite3
import time

client = openai.OpenAI()

def get_structure(data_path):                                    ❶
```

Извлекает структуру базы данных

```
    """Извлечь структуру из базы данных SQLite.

    Аргументы:
        data_path: путь к файлу данных SQLite.
```

```
    Выводит:
        текстовое описание структуры базы данных.
    """
    with sqlite3.connect(data_path) as connection:
        cursor = connection.cursor()
                  cursor.execute("select  sql  from
sqlite_master where type = 'table';")
        table_rows = cursor.fetchall()
        table_ddls = [r[0] for r in table_rows]
        return '\n'.join(table_ddls)


def create_prompt(description, question):          ❷
```

Создает промпт для перевода

```
     """Сгенерировать промпт для перевода вопроса в
SQL-запрос.

    Аргументы:
          description: текстовое описание структуры
базы данных.
         question: вопрос о данных на естественном
языке.

    Выводит:
        промпт для перевода вопроса.
    """
    parts = []
    parts += ['База данных:']
    parts += [description]
    parts += ['Переведи вопрос в SQL-запрос']
    parts += [question]
    return '\n'.join(parts)
```

```
def call_llm(prompt):                                    ❸
```

Вызывает языковую модель

```
     """Отправить запрос языковой модели и вывести
ответ.

    Аргументы:
        prompt: введите запрос к языковой модели.

    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                    {'role':'user',
'content':prompt}
                    ]
                )
                                                 return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')
```

```
def process_query(data_path, query):                    ❹
```

Обрабатывает запрос к базе данных

```
    """Обрабатывает SQL-запрос и выводит результат.

    Аргументы:
        data_path: путь к файлу данных SQLite.
            query: обрабатывает этот запрос к базе
данных.

    Выводит:
        query result.
    """
    with sqlite3.connect(data_path) as connection:
        cursor = connection.cursor()
        cursor.execute(query)
        table_rows = cursor.fetchall()
                table_strings = [str(r) for r in
table_rows]
        return '\n'.join(table_strings)

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
           parser.add_argument('dbpath',  type=str,
help='Путь к данным SQLite')
    args = parser.parse_args()

     data_structure = get_structure(args.dbpath) ❺
```

Считывает структуру данных

```
        while  True:                                                    ❻
```

Отвечает на вопросы, пока пользователь не закроет программу

```
            user_input = input('Введите вопрос:')
            if user_input == 'quit':
                break

            prompt  =  create_prompt(data_structure,
user_input)
            answer = call_llm(prompt)
            query = re.findall('```sql(.*)```', answer,
re.DOTALL)[0]
            print(f'SQL: {query}')

            try:                                                        ❼
```

Обрабатывает запрос к базе данных

```
                result  =  process_query(args.dbpath,
query)
                print(f'Результат: {result}')
            except:
                    print('Ошибка обработки запроса!
Попробуйте переформулировать')
```

В этом коде используется функция для извлечения структуры базы данных ❶ , которую мы обсуждали ранее. Функция для создания запросов ❷ представляет собой немного модифицированную версию той, что использовалась в предыдущем интерфейсе для конкретной базы данных. В отличие от четко прописанной структуры базы данных теперь описание базы передается в виде входных данных и вставляется в промпт. Функция для вызова языковой модели ❸ не изменилась по сравнению с предыдущей версией

интерфейса. С функцией `process_query` ❹ вы познакомились в подразделе 5.3.1.

После чтения аргументов командной строки интерфейс запросов на естественном языке извлекает структуру базы данных ❺. Затем запускается цикл ❻, который продолжается до тех пор, пока пользователь не завершит работу интерфейса. На каждой итерации сначала считывается ввод с клавиатуры (с выходом из цикла, если пользователь вводит команду `quit`), а затем создается промпт и выполняется вызов языковой модели.

Когда исходный вопрос будет переведен в запрос, этот запрос выполняется напрямую ❼. Конечно, перевод может быть неверным и привести к созданию запроса, который невозможно будет выполнить в целевой базе данных. В этом случае SQLite может выдать сообщение об ошибке, а нам необходимо убедиться, что выполнение программы не завершится аварийно. Именно поэтому мы заключаем вызов функции обработки запроса в блок `try-except`.

5.3.4. Проверка в работе

Самое время протестировать наш интерфейс запросов на естественном языке! Теперь он работает с произвольными базами данных, однако у нас всего одна база, поэтому снова используем базу данных видеоигр. В терминале перейдите в каталог, содержащий файл `games.db`, и выполните следующую команду (предположим, что код сохранен в файле `listing3.py`):

```
python listing3.py games.db
```

Откроется поле для ввода, в которое можно вводить вопросы о данных. Вот пример взаимодействия с интерфейсом

запросов на естественном языке:

```
Введите вопрос: Сколько всего видеоигр хранится в базе?
SQL: SELECT COUNT(*) FROM games;
Ответ: (16599,)
Введите вопрос: Сколько видеоигр создала компания Activision?
SQL: SELECT COUNT(*) FROM games WHERE publisher = 'Activision'
Ответ: (975,)
Введите вопрос: Назови видеоигру, которая была выпущена в 2017 году.
SQL: SELECT name
FROM games
WHERE year = 2017
LIMIT 1
Ответ: (Phantasy Star Online 2 Эпизод 4: Подарочное издание)
Введите вопрос: Сколько видеоигр каждого жанра было выпущено?
SQL: SELECT genre, COUNT(*) as num_games
FROM games
GROUP BY genre
Ответ: ('Экшен', 3316)
('Приключения', 1286)
('Файтинг', 848)
('Жанр', 1)
('Прочее', 1739)
('Платформенная игра', 886)
('Головоломка', 582)
('Гонки', 1249)
```

('Ролевая игра', 1488)
('Шутер', 1310)
('Симулятор', 867)
('Спортивные игры', 2346)
('Стратегия', 681)
Введите вопрос: Какие три видеоигры продавались больше в Японии, чем в Европе?
SQL: SELECT name
FROM games
WHERE japansales > eusales
ORDER BY japansales DESC
LIMIT 3;
Ответ: ('Название',)
('Pokemon Red/Pokemon Blue',)
('Pokemon Gold/Pokemon Silver',)
Введите вопрос: Разбей продажи игр в Америке по платформам.
SQL: SELECT platform, sum(americasales) AS total_sales
FROM games
GROUP BY platform
Ответ: ('2600', 90.59999999999992)
('3DO', 0)
('3DS', 78.86999999999996)
('DC', 5.43)
('DS', 390.7099999999977)
('GB', 114.32000000000001)
('GBA', 187.54000000000033)
('GC', 133.46000000000004)
('GEN', 19.27)
('GG', 0)

```
('N64', 139.02000000000015)
('NES', 125.94000000000005)
('NG', 0)
('PC', 93.2800000000005)
('PCFX', 0)
('PS', 336.509999999998)
('PS2', 583.8399999999925)
('PS3', 392.2599999999998)
('PS4', 96.79999999999998)
('PSP', 108.98999999999975)
('PSV', 16.200000000000006)
('Platform', 0.0)
('SAT', 0.7200000000000001)
('SCD', 1)
('SNES', 61.22999999999998)
('TG16', 0)
('WS', 0)
('Wii', 507.7099999999991)
('WiiU', 38.31999999999999)
('X360', 601.0499999999992)
('XB', 186.6900000000008)
('XOne', 83.19000000000003)
Введите вопрос: quit
```

Как видите, можно задавать самые разные вопросы и получать адекватные ответы. Помимо ответа, система также выводит запрос. Обладая определенными знаниями в области SQL, мы можем проверить, точно ли запрос переводит вопрос.

5.4. Интерфейс запросов на естественном языке для графовых данных

Новость о вашем интерфейсе, преобразующем текст в SQL, разлетелась по компании Banana, и некоторые коллеги пользуются им для анализа табличных наборов данных. Ваш новый коллега работает с большими графами, моделируя связи между сотрудниками во внутренней социальной сети компании. Эти данные представлены не в виде таблицы, а в виде графа — формата данных, который отлично подходит для моделирования связей между сущностями (в данном случае людьми). Коллега спрашивает вас, можно ли расширить интерфейс так, чтобы он работал и с такими данными. Зная, что языковые модели в принципе должны уметь работать с различными формальными языками запросов, вы настроены оптимистично и соглашаетесь изучить этот вопрос.

5.4.1. Что такое графовые данные

Подобно реляционным данным, графы — это крайне популярный тип структурированных данных. Граф обычно состоит из набора узлов, соединенных ребрами. С узлами могут быть связаны свойства, а у ребер есть метки. Например, в виде графов часто представляют социальные сети. Здесь узлы обозначают людей, а ребра — дружеские связи и отношения. Кроме того, графы служат естественным представлением дорожных или метрополитеновских сетей. В этом случае узлы обозначают города или станции метро, а ребра — дороги или железнодорожные пути, соединяющие их.

На рис. 5.2 показан пример графа, представляющего сеть метро. Он изображает станции нью-йоркского метро как узлы (они помечены как `Station`). Ребра представляют прямые соединения и помечены названиями соответствующих линий

метро. С узлами связано свойство name, присваивающее узлу имя станции.

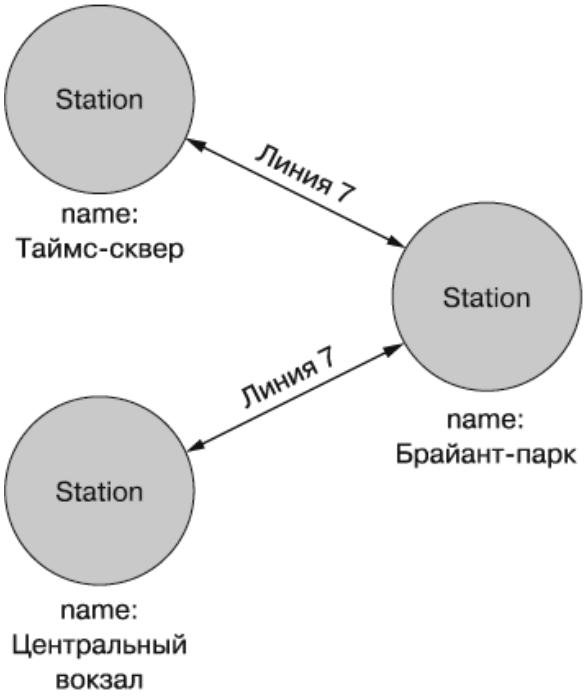


Рис. 5.2. Пример графа, представляющего небольшой фрагмент нью-йоркского метро. Узлы, изображенные в виде кружков, обозначают станции метро и связаны со свойством, которое присваивает им имя представляемой станции метро. Ребра, изображенные стрелками, представляют прямые соединения по определенным линиям метро. Ребра помечены названием линии метро, соединяющей станции

Возможность использования графов для моделирования различных типов данных привела к созданию множества специализированных систем, называемых *графовыми базами данных*, предназначенных для обработки таких данных. Эти системы поддерживают специализированные языки запросов для графов (не SQL), позволяя пользователям формулировать сложные вопросы о хранимых данных. Графовые базы оптимизированы для эффективной обработки больших графов.

Далее с помощью языковых моделей мы будем переводить вопросы на естественном языке в запросы, сформулированные в системе управления графовой базой данных. Мы выясним,

что, по сравнению методом перевода вопросов в SQL-запросы, этот сценарий требует лишь незначительных модификаций.

5.4.2. Настройка базы данных Neo4j

Мы будем работать с системой Neo4j — базой данных, специализированной для работы с графовыми данными. Вам даже не придется ничего устанавливать на локальный компьютер. У Neo4j есть демоверсия онлайн, которой мы и воспользуемся.

Сначала в браузере перейдите по адресу **https://neo4j.com/sandbox/**. Нажмите кнопку **Get Started Free** (Запустить бесплатную песочницу). Откроется форма входа, в которой можно создать новую учетную запись или использовать существующую (например, аккаунт Google). На рис. 5.3 показан экран, который вы должны увидеть далее.

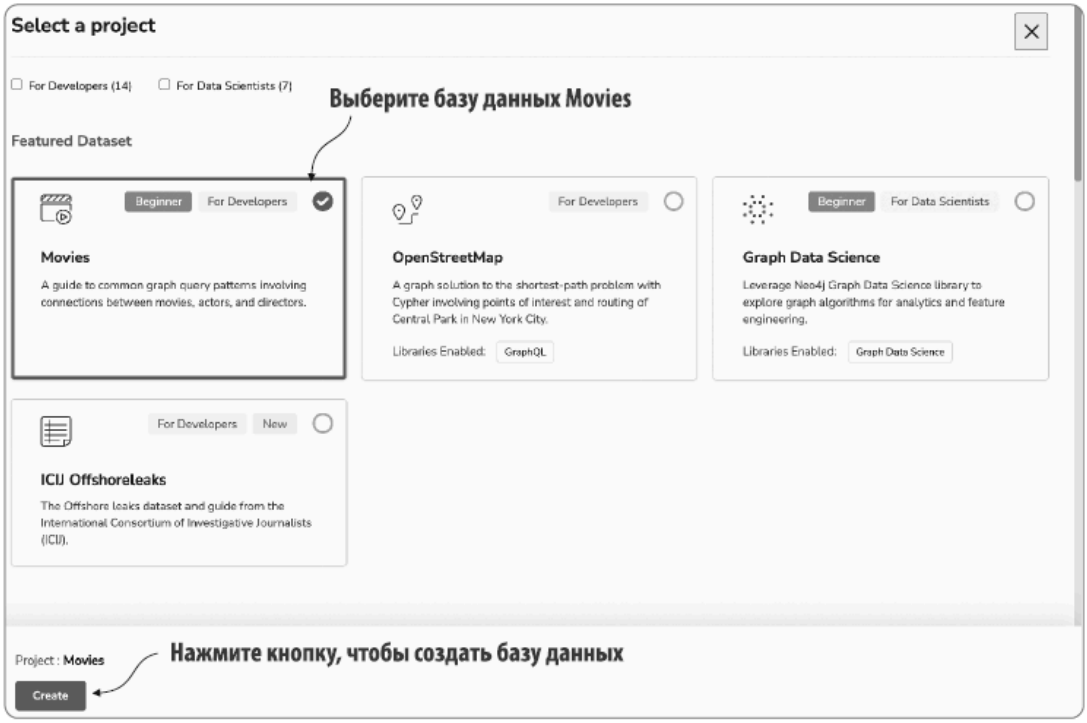


Рис. 5.3. Выберите базу данных Movies и нажмите кнопку Create (Создать), чтобы создать соответствующий экземпляр

Здесь можно выбрать одну из нескольких демонстрационных баз данных, чтобы протестировать Neo4j в работе. Мы выберем базу данных `Movies`. Она содержит информацию о фильмах и актерах, которые в них играли (фильмы и актеры представлены в виде узлов). Щелкните на базе данных `Movies`, а затем нажмите кнопку **Create** (Создать), чтобы создать экземпляр базы и подготовить ее к выполнению запросов. Создание базы может занять несколько минут. После этого откройте базу данных `Movies` — и вы получите доступ к интерфейсу запросов.

На рис. 5.4 показан интерфейс запросов Neo4j. Щелкните на значке базы данных в левом верхнем углу интерфейса. Появится обзор базы. Она содержит несколько типов меток (то есть типов узлов), включая `Person` и `Movie`. Кроме того, в ней содержатся типы отношений, такие как `ACTED_IN` и `DIRECTED`, которые помечают ребра в графе. Например, эти два типа отношений позволяют отслеживать, кто в каком фильме сыграл роль (`ACTED_IN`) и кто какой фильм режиссировал (`DIRECTED`). Свойства связаны с узлами и сопоставляют ключи со значениями. Обзор базы данных показывает ключи свойств, такие как `name` (присваивает актеру имя) и `title` (присваивает фильмам названия).

Запросы можно вводить в текстовое поле в верхней части экрана и отправлять, нажимая кнопку справа от текстового поля.

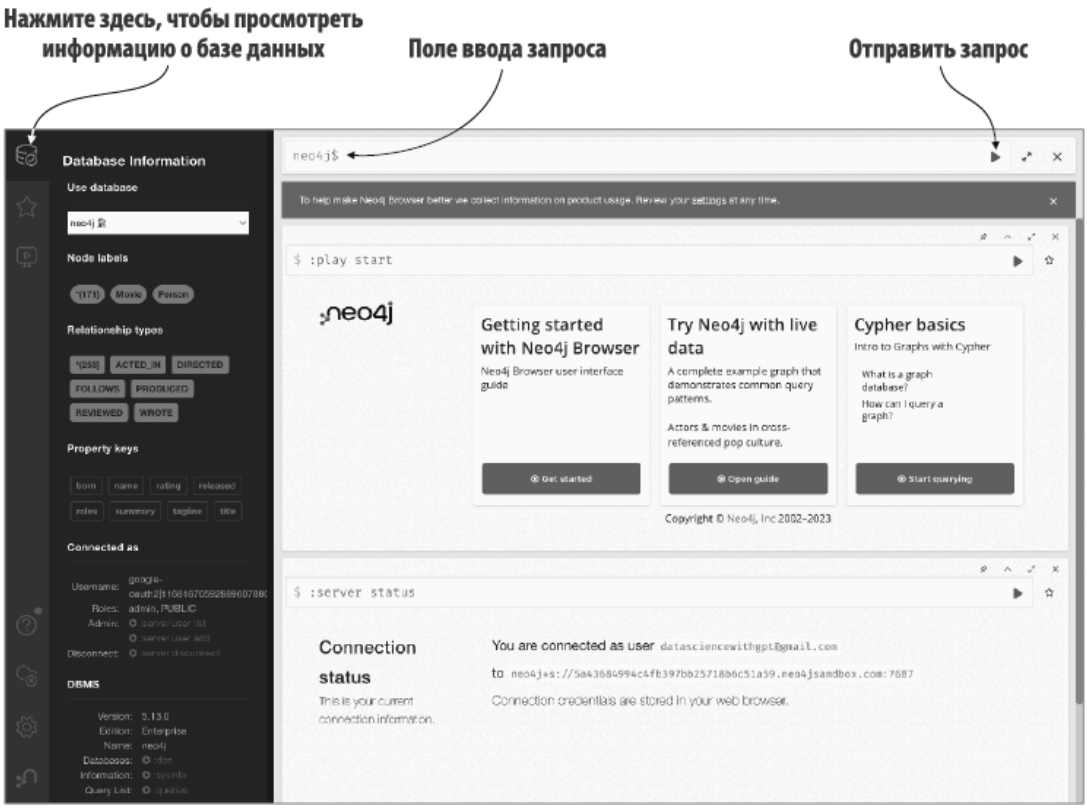


Рис. 5.4. Щелкните на значке базы данных (в левом верхнем углу интерфейса), чтобы получить информацию о текущей графовой базе, в том числе о типах узлов и ребер

5.4.3. Язык запросов Cypher

СУБД Neo4j поддерживает язык запросов Cypher. Подробное изучение этого языка выходит за рамки книги (вы можете изучить документацию Neo4j, доступную по адресу **https://neo4j.com/docs/cypher-manual/current/introduction/**), поэтому здесь мы рассмотрим лишь его основы. Цель такого краткого введения — научить вас понимать семантику запросов, генерируемых языковой моделью, по крайней мере в простых случаях.

Простой тип запроса Cypher использует оператор `MATCH`, описывающий подграф для поиска. Например, мы хотим найти все узлы определенного типа. Запрос для поиска имен всех людей в базе данных `Movies` выглядит так:

```
MATCH (p:Person) ❶ Шаблон для сопоставления
RETURN p.name    ❷ Результат, который нужно вернуть
```

Оператор `MATCH` описывает шаблон для сопоставления ❶. В данном случае шаблон состоит из отдельных узлов `Person`. Оператор `MATCH` присваивает переменные узлам или ребрам, которые появляются в шаблоне. Здесь мы вводим переменную `p` и присваиваем ее узлам, соответствующим шаблону. Оператор `RETURN` ❷ описывает результат запроса на основе сопоставленных шаблонов. В данном случае мы указываем системе вернуть свойство `name` для каждого узла, соответствующего шаблону.

Шаблоны могут выходить за пределы одного узла. Например, нам может потребоваться найти названия всех фильмов с участием Тома Круза. В этом случае нужный нам шаблон будет состоять не из одного, а из двух соединенных узлов. Мы ищем узел типа `Movie`, соединенный через ребро типа `ACTED_IN` с узлом типа `Person`, для свойства `name` которого задано значение `Tom Cruise`. Это можно сделать с помощью такого запроса Cypher:

```
❶ Находит фильмы с участием Тома Круза
MATCH (p:Person name: 'Tom Cruise')-[:ACTED_IN]->(m:Movie)
RETURN m.title                                    ❷ Возвращает название фильма
```

Выражение `(p:Person {name: 'Tom Cruise'})` ❶ находит все узлы типа `Person`, свойство `name` которых имеет значение `Tom Cruise`. Выражение `(m:Movie)` находит все узлы типа `Movie`. Наконец, мы соединяем их с помощью выражения `-[:ACTED_IN]->`. Оно представляет направленное соединение (отсюда форма стрелки) между первым узлом (представляющим Тома Круза) и вторым узлом (представляющим какой-либо фильм). Тип соединения ограничен значением `ACTED_IN` (которое исключает, к примеру, фильмы, в которых Том Круз был режиссером, но не играл).

Наконец, обратите внимание, что выражение MATCH снова присваивает частям этого шаблона переменные. Том Круз будет представлен переменной p, а фильмы, в которых он играл, — переменной m. Выражение RETURN ❷ извлекает свойство title узла фильма.

И напоследок посмотрим, как Neo4j вычисляет агрегаты (аналогично SQL):

```
❶ Находит фильмы с участием Тома Круза
MATCH (p:Person name: 'Tom Cruise')-[:ACTED_IN]->(m:Movie)
RETURN count(*)                                          ❷
                                                  Возвращает количество фильмов
```

Запрос похож на предыдущий и просто подсчитывает количество фильмов с участием Тома Круза в качестве актера ❶ . Оператор RETURN ❷ содержит соответствующий агрегат. Введя данный запрос, вы должны получить результат 3 (очевидно, демонстрационная база данных неполная).

5.4.4. Перевод вопросов на Cypher

Мы будем использовать подход, аналогичный тому, с помощью которого мы переводили вопросы в SQL-запросы. В первую очередь необходимо изменить промпт для нашей языковой модели. Дадим ей указание переводить не на SQL, а на Cypher. К счастью, языковые модели, такие как GPT-4o, были предварительно обучены на большом и разнообразном наборе данных. Среди обучающих данных, вероятно, содержались и запросы на Cypher, так что языковые модели можно использовать для перевода. Как всегда, абсолютных гарантий нет, и запросы на Cypher, сгенерированные языковой моделью, могут не совсем точно отражать наши вопросы. Однако, по крайней мере для простых запросов, перевод, как правило, корректен.

5.4.5. Генерация промптов

Для перевода вопросов в запросы Cypher необходимо добавить в промпт некоторую информацию.

- Во-первых, это сам вопрос, который требуется перевести.
- Во-вторых, описание структуры базы данных. В случае с SQL структура базы определяется, например, именами таблиц и столбцов. В случае базы данных Neo4j необходимо добавить информацию о типах узлов и ребер, а также имена наиболее релевантных свойств. По сути, вся эта информация показана в левой части рис. 5.4.

Для простоты мы сосредоточимся на представленной ранее демонстрационной базе данных, в которой содержится информация о фильмах. Значит, нам нужно будет строго прописать структуру базы. Конечно, как и в случае с интерфейсом преобразования текста в SQL, можно расширить интерфейс так, чтобы он работал с произвольными графовыми базами данных.

Мы будем использовать следующий шаблон промпта:

```
База данных Neo4j:              ❶ Описание базы данных
Метки узла: Movie, Person
Типы отношений: ACTED_IN, DIRECTED,
FOLLOWS, PRODUCED, REVIEWED, WROTE
Ключи свойств: born, name, rating, released
roles, summary, tagline, title
[Вопрос]                        ❷ Вопрос для перевода
Запрос Cypher:                  ❸ Указание целевого языка
```

Шаблон содержит описание базы данных ❶. Оно включает в себя спецификацию типа базы (Neo4j), а также список меток узлов, типов отношений и свойств. Обратите внимание, что структура базы строго прописана в шаблоне промпта. При использовании интерфейса с другой базой эту часть промпта придется заменить. Далее в шаблоне указан вопрос для

перевода ❷ . Это метка-заполнитель, ведь пользователи должны иметь возможность задавать разные вопросы о данных. Промпт завершается указанием целевого языка для его перевода ❸, что служит для языковой модели инструкцией перевести вопрос в запрос на языке Cypher.

В этом фрагменте кода создается экземпляр шаблона для заданного вопроса:

```
def create_prompt(question):
    parts = []
    parts += ['База данных Neo4j:']
    parts += ['Метки узла: Movie, Person']
          parts  +=  ['Типы  отношений:  ACTED_IN,
DIRECTED,']
    parts += ['FOLLOWS, PRODUCED, REVIEWED, WROTE']
     parts += ['Ключи свойств: born, name, rating,
released']
    parts += ['roles, summary, tagline, title']
    parts += [question]
    parts += ['Запрос Cypher:']
    return '\n'.join(parts)
```

5.4.6. Полный код

В листинге 5.3 используется ранее рассмотренная функция для генерации промптов ❶ и повторно применяется функция для вызова языковой модели GPT-4o ❷ (допускающая при необходимости выполнение нескольких попыток).

Листинг 5.3. Перевод текстовых вопросов в запросы Cypher для Neo4j

```
import argparse
import openai
```

```
import re
import time

client = openai.OpenAI()

def  create_prompt(question):                                    ❶
```

Генерирует промпты для перевода

```
    """Сгенерировать промпт для перевода вопроса в
запрос Cypher.

    Аргументы:
         question: вопрос о данных на естественном
языке.

    Выводит:
        промпт для перевода вопроса.
    """
    parts = []
    parts += ['База данных Neo4j:']
    parts += ['Метки узла: Movie, Person']
         parts  +=  ['Типы  отношений:  ACTED_IN,
DIRECTED,']
    parts += ['FOLLOWS, PRODUCED, REVIEWED, WROTE']
     parts += ['Ключи свойств: born, name, rating,
released']
    parts += ['roles, summary, tagline, title']
    parts += [question]
    parts += ['Запрос Cypher:']
    return '\n'.join(parts)
```

```
def call_llm(prompt):                                    ❷
```

Вызывает языковую модель

```
     """Отправить запрос большой языковой модели и
вывести ответ.

    Аргументы:
        prompt: введите запрос к языковой модели.

    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                    {'role':'user',
'content':prompt}
                    ]
                )
                                                 return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
```

```
        parser.add_argument('question',  type=str,
help='Вопрос о фильмах')
    args = parser.parse_args()


        prompt  =  create_prompt(args.question)  ❸
```

Создает промпт

```
      answer  =  call_llm(prompt)                ❹
```

Генерирует ответ

❺ Извлекает запросы на языке Cypher

```
     query = re.findall('```cypher(.*)```', answer,
re.DOTALL)[0]


    print(f'Запрос Cyper: {query}')
```

Получив во входных данных вопрос о базе данных, код генерирует соответствующий промпт ❸ , получает ответ от языковой модели ❹ и, наконец, извлекает из этого ответа запрос на языке Cypher ❺ . Регулярное выражение, используемое для извлечения, немного различается, поскольку GPT заключает запросы Cypher в шаблон '''cypher ... '''. Наконец, мы выводим извлеченный запрос.

5.4.7. Проверка в работе

Файл с кодом из листинга 5.3 доступен по ссылке **Listing 3**, размещенной на сайте книги в разделе дополнительных материалов к главе. Скачайте код и запустите из командной строки, введя команду, содержащую вопрос о количестве фильмов в базе данных:

```
python listing3.py "Сколько фильмов здесь хранится?"
```

Должен быть выведен примерно такой запрос:

```
MATCH (m:Movie)
RETURN COUNT(m) AS numberOfMovies
```

Теперь этот запрос можно ввести в интерфейс Neo4j и получить соответствующий результат (38). Попробуйте ввести еще несколько запросов, чтобы лучше понять возможности языковой модели. Как видите, внеся всего несколько изменений в шаблон промпта, мы превратили интерфейс преобразования текста в SQL в интерфейс преобразования текста в Cypher, который в большинстве случаев работает корректно.

Резюме

- Структурированные данные имеют стандартный формат, что упрощает их синтаксический анализ. Примеры — табличные данные и графы.
- Структурированные данные часто обрабатываются с помощью специализированных инструментов.
- Реляционные системы управления базами данных обрабатывают табличные данные и обычно поддерживают SQL-запросы.
- Системы управления графовыми данными обрабатывают данные, представляющие графы.

- Языковые модели выполняют перевод с естественного языка на многие формальные языки запросов.
- Помимо вопроса, в промптах, составляемых для перевода, указывается структура базы данных.
- Делайте резервную копию базы данных, прежде чем выполнять запросы, сгенерированные языковыми моделями.
- Не доверяйте слепо языковой модели при генерации точных запросов.

6. Анализ изображений и видеороликов

В этой главе

- Анализ изображений.
- Сравнение изображений.
- Анализ видеороликов.

В предыдущих главах вы узнали, как анализировать текст и структурированные данные. Все ли нюансы были учтены? Отнюдь. Подавляющая часть существующих данных представлена в виде изображений и видеороликов. Например, видео занимает две трети от общего объема данных, передаваемых через Интернет. В этой главе мы выясним, каким образом языковые модели могут помочь извлекать полезную информацию и из таких типов данных.

В следующих разделах представлено несколько небольших проектов по обработке изображений и видеоданных. Модель GPT-4o — изначально мультимодальная. Ее можно использовать для всех перечисленных задач.

- Сначала мы выясним, как с помощью GPT-4o получать ответы на произвольные вопросы (на естественном языке) об изображениях.
- Затем используем эту модель для создания приложения, которое будет автоматически отмечать изображенных на фотографиях людей, добавляя теги.

- Наконец, мы используем GPT-4o для автоматической генерации заголовков видеофайлов.

Цель этих мини-проектов — проиллюстрировать возможности обработки визуальных данных, предлагаемые последним поколением больших языковых моделей. Выполнив проекты, вы сможете создавать собственные приложения для обработки изображений и видеороликов в рамках различных сценариев.

6.1. Настройка

Для запуска примеров кода потребуется установить еще один пакет Python, а именно библиотеку OpenCV для обработки изображений. Чтобы ее установить, выполните в терминале следующую команду:

```
pip install opencv-python==4.8.1.78
```

Мы будем использовать эту библиотеку, например, для чтения изображений с диска и разделения видео на кадры.

Затем нужно будет установить библиотеку, которая позволит отправлять запросы напрямую в веб-сервисы OpenAI (с ее помощью мы будем отправлять изображения, сохраненные на локальном компьютере, в OpenAI):

```
pip install requests==2.31.0
```

Отлично! Если при выполнении этих команд не появились сообщения об ошибках, значит, ваша система теперь настроена для анализа данных изображений и видеороликов с помощью языковой модели GPT-4o. Переходим к первому проекту.

6.2. Получение ответов на вопросы об изображениях

Нейронные сети, способные обнаруживать на изображениях объекты (скажем, автомобили), существуют уже много лет. Так в чем же особенность обработки изображений с помощью языковой модели GPT-4o?

Основное ограничение классических моделей для обработки изображений заключается в том, что их нужно обучать для выполнения конкретных задач анализа. Например, у вас есть нейронная сеть, которая очень хорошо выявляет изображения кошек. Ее можно использовать для фильтрации фотографий кошек из вашей личной коллекции. Однако, возможно, вам интересны не кошки в целом, а конкретно персидские кошки золотистого окраса. Если ваша модель не обучена обнаруживать именно эту породу кошек, то вам не повезло и придется самим тегировать достаточное количество примеров изображений. Это утомительно, и в итоге вы, скорее всего, перестанете пользоваться моделью и будете просматривать фотографии вручную. Особенность обработки изображений с помощью GPT-4o (и подобных моделей) заключается в том, что модель решает широкий спектр задач, касающихся изображений, используя лишь описание задачи (на естественном языке).

Мы применим эту особенность модели, чтобы создать универсальную систему вопросов и ответов для изображений. Вы, пользователь, будете формулировать произвольные вопросы на естественном языке и указывать на изображение, а система будет генерировать текстовый ответ. Например, запрос к системе на обнаружение «персидских кошек золотистого окраса» на изображении должен работать по умолчанию и не требовать специальных обучающих данных для такой задачи.

6.2.1. Определение мультимодального ввода

В этом разделе мы создадим систему, которая принимает два вида входных данных:

- URL изображения в Интернете;
- вопрос об изображении на естественном языке.

Выходные данные — это ответ на вопрос (в текстовом виде). Внутри система использует языковую модель GPT-4o для обработки вопроса применительно к поданному во входных данных изображению. Она генерирует мультимодальные промпты, комбинируя несколько типов данных (в данном случае — текст и изображения). На рис. 6.1 показан пример промпта: он содержит одно изображение (яблока) и вопрос об изображении (изображен ли здесь банан). В данном случае правильный ответ — «Нет».



Рис. 6.1. Мультимодальный промпт, содержащий изображение и текст. В промпт входит инструкция, указывающая языковой модели, что она должна определить, изображен ли на картинке банан. В данном случае ожидаемый результат — «Нет» (а если бы на картинке был банан, то «Да»)

Как составлять подобные промпты для модели GPT-4o? Можно повторно использовать эндпоинт `chat.competions`. Напомню, что он принимает в качестве входных данных список предыдущих сообщений, которыми обменялись пользователь и (потенциально) система. Для работы системы, отвечающей на вопросы об изображениях, требуется лишь одно сообщение — и оно исходит от пользователя.

В отличие от кода, с которым мы работали ранее, сообщения теперь могут содержать мультимодальный контент. В этом конкретном случае контент состоит из одного текстового фрагмента (вопроса, заданного пользователем) и одного изображения (на данный момент указанного как URL). Мы будем использовать это сообщение далее в коде (где `question` — переменная, содержащая текст вопроса, а `image_url` — URL изображения):

```
{'role':'user', 'content':[
    {'type':'text', 'text':question},  ❶ Текст вопроса
    {'type':'image_url', 'image_url':{
        'url':image_url}}              ❷ URL изображения
    ]
}
```

Прежде всего обратите внимание на атрибут `role`, идентифицирующий сообщение как сгенерированное пользователем. Далее содержимое сообщения указано как список словарей Python. Каждый из этих словарей описывает один элемент сообщения. Мы теперь рассматриваем мультимодальные данные (то есть изображения и текст), поэтому необходимо уточнить тип (или *модальность*) каждого входного элемента. Это достигается путем присвоения атрибуту `type` значения `text` или `image_url`. Фактическое содержимое указывается с помощью атрибута `text` ❶ или (в случае изображения) атрибута `image_url` ❷. Языковая модель GPT-4o способна понимать, что вопрос относится к изображению, и может корректно обработать оба элемента.

СОВЕТ Здесь входные данные содержат одно изображение, однако оно может состоять из множественных элементов одного типа: например, нескольких изображений. Мы воспользуемся этой возможностью в проекте, с которым будем работать в следующем разделе.

6.2.2. Обсуждение кода

В листинге 6.1 показан полный код системы для получения ответов на вопросы об изображениях. После того как на вход поданы URL изображения и вопрос ❸, все самое интересное (то есть получение ответов на вопросы об изображении) происходит в функции `analyze_image` ❶.

Листинг 6.1. Получение ответов на вопросы об изображениях с помощью языковых моделей

```
import argparse
import openai
import time

client = openai.OpenAI()

def analyze_image(image_url, question):                    ❶
```

Отвечает на вопрос об изображении

```
    """Использовать языковую модель для ответа на
вопросы об изображении.

    Аргументы:
        image_url: URL, ведущий к изображению.
        question: вопрос от изображении.
```

```
    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                     messages=[                    ❷
```



```
                    {'role':'user', 'content':[
                                    {'type':'text',
'text':question},
                                {'type':'image_url',
'image_url':{
                                'url':image_url
                                }
                            }]
                    }]
                )
                                                return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')

if __name__ == '__main__':
```

```
        parser = argparse.ArgumentParser()          ❸
```

Параметры ввода

```
        parser.add_argument('imageurl', type=str,
help='URL изображения')
        parser.add_argument('question', type=str,
help='Вопрос об изображении')
    args = parser.parse_args()


          answer = analyze_image(args.imageurl,
args.question)
    print(answer)
```

Как видите, для получения ответов на вопросы об изображениях требуется лишь несколько строк кода на Python! Функция `analyze_image` ❶ содержит всего один вызов языковой модели GPT-4o с использованием сообщения, описанного в предыдущем подразделе ❷. Тот факт, что теперь мы предоставляем мультимодальные входные данные, не меняет формат ответа. Снова мы получаем объект, который содержит сообщение, сгенерированное языковой моделью. Хотя входные данные теперь могут быть мультимодальными, вывод представляет собой текст. Модель получила инструкцию сгенерировать ответ на заданный вопрос ❸ (то есть именно то, что нужно пользователю), поэтому выходные данные выводятся напрямую пользователю.

6.2.3. Проверка в работе

Пора протестировать нашу систему! Представьте, что вы снова в компании Banana, где вам предстоит разработать ее новый логотип. Вы хотите создать логотип на основе изображения банана. Поискав в Интернете, вы находите большие

репозитории с изображениями фруктов. Но какие из них — бананы? Просматривать изображения вручную не хочется, лучше делегировать эту задачу языковой модели. К счастью, можно просто взять код из предыдущего подраздела, указав URL всех картинок с фруктами с вопросом: «Это банан (“Да”, “Нет”)?» По сути, вы будете использовать систему для получения ответов на вопросы об изображениях как метод классификации (это лишь один из множества возможных сценариев ее применения). Затем можно написать простой скрипт, который будет перебирать все соответствующие URL и сохранять те, на которые система ответит «Да».

На сайте книги доступны ссылки на файл с кодом из листинга 6.1 (**Listing 1**), а также на изображения фруктов (**Fruit 1**). Скачайте код, перейдите в терминале в каталог с ним и выполните следующую команду, требующую от системы ответить на вопрос, изображен ли на картинке банан:

```
ру python [URL] 'Это банан ("Да", "Нет")?'
```

В этой команде замените `[URL]` URL изображения (подходящий URL можно получить, например скопировав адрес ссылки **Fruit 1**, доступной на сайте книги).

Классифицировать объекты относительно просто, особенно такие часто встречающиеся, как бананы. Поэтому для большинства примеров должны отобразиться точные результаты. Протестируйте систему, используя несколько разных фруктов, а может, и другие изображения на свой выбор. Диапазон вопросов, которые можно задавать, практически неисчерпаем (даже если не учитывать довольно щедрый лимит на длину ввода 128 000 токенов, что эквивалентно примерно 300 страницам текста).

Обращаю ваше внимание на один важный нюанс. Обработка изображений с помощью модели GPT-4o может быть

дорогостоящей! Точная стоимость зависит от размера изображения и используемой степени детализации. Управлять детализацией можно с помощью параметра `detail`. Например, выберите низкую степень, используя следующую спецификацию URL изображения (во входных данных модели):

```
{'type':'image_url',  'image_url':{'url':image_url,
'detail':'low'}}
```

Установите значение атрибута `detail` — `low`, тогда стоимость будет эквивалентна 85 токенам за изображение (то есть стоимости обработки текста в 85 токенов с помощью языковой модели GPT-4o). Если задать значение степени детализации `high` (установлено по умолчанию), то стоимость будет состоять из фиксированной суммы за 85 токенов и переменной суммы, которая зависит от размера изображения. Чтобы рассчитать переменную составляющую стоимости, сначала масштабируем изображение до размера (в пикселах) 2048 × 2048 (сохраняя соотношение сторон). Этот этап масштабирования применяется только к изображениям размером более 2048 × 2048 пикселов. Второй этап масштабирования выполняется в любом случае. Он масштабирует меньшую сторону изображения до размера 768 пикселов. Теперь определим, какое минимальное количество квадратов 512 × 512 пикселов потребуется, чтобы покрыть изображение после второго этапа масштабирования. Переменные затраты рассчитываются пропорционально количеству квадратов: каждый квадрат умножается на 170 токенов (такова стоимость обработки одного квадрата, установленная OpenAI).

Например, мы хотим обработать изображение размером 1024 × 1024 пикселов, установив высокую степень детализации. В этом случае первый этап масштабирования можно

пропустить, так как изображение помещается в квадрат 2048 × 2048 пикселов. Однако второй этап масштабирования выполняется в любом случае. Он масштабирует изображение до размера 768 × 768 пикселов. Чтобы покрыть квадрат со стороной 768 пикселов, нам нужны четыре квадрата размером 512 × 512 пикселов. Это означает, что за обработку изображения мы заплатим:

- 85 токенов (фиксированная составляющая);
- 4 × 170 токенов = 680 токенов (переменная составляющая).

Итого мы заплатим 85 + 4 × 170 = 765 токенов. По текущим ценам это соответствует 0,003825 доллара (то есть менее одного цента). Хотя сумма и кажется приемлемой, всегда учитывайте стоимость обработки больших репозиториев изображений, выполняемой с помощью языковых моделей.

> СОВЕТ Чтобы узнать стоимость обработки изображений с определенным разрешением, можно воспользоваться калькулятором цен OpenAI: https://openai.com/pricing.

6.3. Тегирование людей на изображениях

Представьте ситуацию: вы только что вернулись из (заслуженного) отпуска с друзьями и, конечно, сделали множество фотографий. Вы хотите отправить друзьям фото с ними. Но как сделать это эффективно? Можно просмотреть фотографии вручную и отметить каждого друга отдельно. Но вы только вернулись из отпуска, ваш электронный почтовый ящик переполнен, и на просмотр фото совсем нет времени. Как же быть?

6.3.1. Обзор

В этом подразделе мы создадим небольшое приложение, которое автоматически тегирует людей на изображениях. Пользователи предоставляют три элемента входных данных:

- путь к каталогу, содержащему фотографии, к которым нужно добавить теги;
- путь к каталогу, содержащему фото людей, которых нужно найти;
- путь к каталогу, куда будут сохраняться размеченные фотографии.

Для простоты мы будем использовать в качестве тегов имена файлов. Мы предполагаем, что фотографии с человеком, которого нужно найти, названы по имени этого человека. Например, в нашем каталоге с фотографиями людей, которых нужно найти, есть изображения с именами `Joe.png` и `Jane.png`. Чтобы тегировать нужное изображение, мы просто изменим имя файла, добавив в его начало имена людей, которые запечатлены на нем.

Предположим, у нас есть изображение `beach.png`, на котором есть и Джо, и Джейн. Тогда в выходном каталоге мы создадим два файла с именами `Joebeach.png` и `Janebeach.png`, показывая, что обе они есть на фотографии пляжа. Если нам нужно отправить все фотографии, на которых изображен один и тот же человек, например Джо, то можно найти все файлы, имя которых удовлетворяет регулярному выражению `Joe*.png` (где * обозначает произвольные строковые значения).

Сначала на этапе внутренней обработки требуется загрузить, во-первых, изображения людей, которых нужно

найти, и, во-вторых, фото, которые нужно тегировать. Для каждого человека, которого мы ищем, и для каждой фотографии, которую нужно тегировать, будем перебирать все возможные пары. Например, если мы ищем пять человек и у нас есть десять фото для тегирования, то придется рассмотреть 50 пар. Для каждой из этих пар мы используем языковую модель GPT-4o, чтобы решить, запечатлен ли соответствующий человек на тегируемой фотографии.

Для этого понадобятся мультимодальные промпты, содержащие текст и два изображения. Первое — фото человека, которого нужно найти, а второе — фото, на котором его нужно отметить. С помощью текстового промпта можно дать языковой модели указание сравнить изображения и определить, есть ли этот человек на фотографии. При каждом совпадении — то есть сочетании человека и фото, на котором он есть, — мы будем копировать соответствующую фотографию в выходную папку, добавляя перед ее названием имя человека.

На рис. 6.2 показан пример промпта. Слева — фотография Джейн, одной из тех, кого мы ищем. Справа — фото для тегирования. В текстовой инструкции содержится указание для языковой модели сравнить две фотографии и дать ответ «Да», если на них изображен один и тот же человек (или «Нет», если люди разные). В данном случае на фотографиях изображены разные люди и правильный ответ должен быть «Нет».



Рис. 6.2. Мультимодальный промпт, содержащий два изображения и текст: языковая модель получает указание проверить, изображен ли на обеих фотографиях один и тот же человек (ожидаемый ответ: «Да») или нет (ожидаемый ответ: «Нет»)

6.3.2. Кодирование локально хранящихся изображений

В разделе 6.2 мы с помощью языковой модели GPT-4o анализировали интернет-изображения. Теперь речь идет о наших личных отпускных фотографиях. Допустим, нам не хочется выкладывать их в сеть и делать общедоступными. Каким образом передать их только GPT-4o?

Скорее всего, придется преобразовать изображения в формат, подходящий для GPT-4o. Языковая модель GPT-4o поддерживает широкий спектр форматов изображений, таких как PNG, JPEG, WEBP и GIF. Для любого формата размер файла изображения в данный момент ограничен 20 Мбайт. Чтобы загрузить изображения поддерживаемых типов в GPT-4o, сначала нужно закодировать их с помощью кодировки base64.

Что такое кодирование base64

Кодирование base64 — это способ представления двоичных данных в виде текстовой строки, состоящей из печатаемых символов. Как следует из названия base64, используемый для строки алфавит основан на 64 символах. Это означает, что

каждый символ можно представить с помощью шести бит (поскольку они позволяют представить 2^6 = 64 возможных символа). Учитывая, что хранение данных в компьютерах осуществляется на уровне байтов (по восемь бит), кодирование целесообразно производить группами из трех байт (24 бита). Используя кодировку base64, три байта можно использовать для представления четырех символов (поскольку 24 / 6 = 4).

В Python кодирование двоичных данных в формат base64 выполняется с помощью модуля `base64`. Приведенный ниже код открывает файл изображения, путь к которому указан в переменной `image_path`, и кодирует его с использованием формата base64:

```
with open(image_path, 'rb') as image_file:
    encoded = base64.b64encode(image_file.read())
```

Мы преобразовали двоичные данные изображения в строку в формате base64. Прежде чем отправлять такие изображения языковой модели GPT-4o, необходимо выполнить еще одно (последнее) преобразование: представить строку в кодировке UTF-8.

Что такое кодировка UTF-8

UTF-8 — это способ представления строковых данных. Он чрезвычайно популярен и используется примерно на 98 % сайтов. UTF-8 поддерживает более миллиона символов из самых разных языков. Все эти символы можно представить с помощью фиксированного количества байтов: четыре байта для представления каждого символа. Однако это

неэффективно, ведь не учитывается тот факт, что некоторые символы встречаются гораздо чаще других. Если закодировать распространенные символы меньшим количеством байтов, оставив представления с большим количеством байтов для более редких, то можно представить тот же текст, используя меньше байтов. Именно это и делает UTF-8, а поскольку для представления разных символов может потребоваться разное количество байтов, то UTF-8 также называют *стандартом переменной длины*. При этом он был разработан так, чтобы сохранять обратную совместимость со старым стандартом ASCII: первые 128 символов кодируются в нем точно так же.

Преобразовать нашу строку base64, кодирующую изображение, в формат UTF-8 в Python можно с помощью функции `decode`. При условии, что изображение все еще закодировано в переменной `encoded`, можно сделать это с помощью следующего кода:

```
image = encoded.decode('utf-8')
```

Полученное изображение, закодированное как текстовая строка UTF-8, подходит в качестве входных данных для языковой модели GPT-4o. Далее мы рассмотрим, как загрузить изображения в этом формате на платформу OpenAI. После их загрузки ссылки на эти изображения можно будет добавить в наши запросы. Изображения обычно указываются как компоненты запроса:

```
{'type':'image_url', 'image_url':{'url':image_url}}
```

Здесь `image_url` обозначает URL, ведущий к изображению, которое необходимо проанализировать. Ранее мы использовали для этого общедоступные URL. Теперь мы анализируем личные изображения, которые отправим компании OpenAI, чтобы использовать их только для обработки конкретных запросов. С учетом того, что элемент `image` по-прежнему представляет изображение, которое закодировано как строка, URL изображения можно задать так:

```
image_url = {'url':f'
{image}'}
```

В этом коде предполагается, что изображение хранится в формате PNG (если нет, то замените значение `png` на соответствующий идентификатор формата, например `jpeg`). URL объединяет метаданные об изображении (такие как тип изображения и кодировка) с суффиксом строки, представляющим саму фотографию.

6.3.3. Передача изображений из локального хранилища в OpenAI

Воспользуемся этим проектом как возможностью увидеть альтернативный способ взаимодействия с моделями GPT. Такой подход даст представление о внутренних механизмах работы библиотеки Python от OpenAI. До сих пор мы использовали обертки Python, которые в фоновом режиме отправляли запросы на платформу OpenAI. Для отправки своих локальных изображений языковой модели GPT-4o мы сами создадим такие запросы.

Мы используем библиотеку `requests` Python для создания HTTP-запросов, отправляем промпты (с текстом и изображениями) в GPT-4o и получаем ответ. Точнее говоря, мы будем отправлять HTTP-запросы методом `POST`. Это тип

запроса, принимаемый платформой OpenAI. Такие запросы можно отправить с помощью метода `requests.post`.

Запросы будут содержать всю информацию, которая нужна GPT-4o для решения интересующей нас задачи (в данном случае проверки, изображен ли на двух фотографиях один и тот же человек). Прежде всего в запрос необходимо добавить заголовки (`headers`). Мы будем использовать следующие заголовки:

```
headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer ...'
}
```

Как видите, они заданы как словарь Python. Для нашего сценария использования требуется сохранить только два свойства: тип данных (мы планируем отправлять контент в формате JSON) и учетные данные доступа (три точки обозначают наш ключ доступа OpenAI).

Далее необходимо указать полезную нагрузку, то есть содержимое, которое следует отправить с помощью запроса в первую очередь:

```
payload = {
    'model': 'gpt-4o',                          ❶ Указание модели
    'messages': [
        {'role': 'user', 'content': ...}        ❷ Первое сообщение
        ],
    'max_tokens':1                              ❸ Длина вывода
}
```

Обратите внимание: полезная нагрузка содержит именно те поля, которые мы обычно указывали в вызовах метода `completions.create`. И это не случайно, поскольку второй метод внутри себя создает запросы с аналогичной полезной нагрузкой. Сначала полезная нагрузка указывает модель ❶ : `gpt-4o` (чтобы иметь возможность обрабатывать

мультимодальные входящие промпты). Мы указываем список сообщений, в котором всего одна запись ❷ . Сообщение помечено как исходящее от пользователя (`'role':'user'`), а его содержимое, обозначенное тремя точками, будет содержать текстовые инструкции и изображения. Наконец, мы ограничиваем длину ответа одним токеном (`'max_tokens':1`) ❸ . Это логично, ведь нам нужен бинарный результат: один и тот же человек либо есть на нескольких входных изображениях (ожидаемый ответ: «Да»), либо нет (ожидаемый ответ: «Нет»).

Сформировав заголовки и полезную нагрузку, можно вызвать модель GPT-4o с помощью такого кода:

```
response = requests.post(
    'https://api.openai.com/v1/chat/completions',
    headers=headers, json=payload)
```

Первым параметром в вызове метода `requests.post` указан URL, на который нужно отправить запрос. В данном случае адрес `https://api.openai.com/v1/chat/completions` показывает, что требуется выполнить задачу типа `Completion`, используя одну из чат-моделей OpenAI (к ним относится и GPT-4o). Мы используем созданные ранее заголовки и полезную нагрузку.

В ответе содержится объект с результатом работы GPT-4o. Получить доступ к ответу (в котором говорится, изображен ли на двух фотографиях один и тот же человек) можно с помощью такого фрагмента кода:

```
response.json()['choices'][0]['message']['content']
```

6.3.4. Полный рабочий код для реализации

Теперь мы готовы обсудить полную реализацию от начала до конца! В листинге 6.2 представлен код, позволяющий отмечать людей на фотографиях.

Листинг 6.2. Тегирование людей на фотографиях из локального хранилища

```
import argparse
import base64
import os
import requests
import shutil


def load_images(in_dir):                                    ❶
```

Загружает изображения с диска

```
    """Загружает изображения из каталога.

    Аргументы:
        in_dir: путь к входному каталогу.

    Выводит:
          словарь сопоставлений имен файлов с PNG-
изображениями.
    """
    name_to_image = {}
    file_names = os.listdir(in_dir)
    for file_name in file_names:
        if file_name.endswith('.png'):
                image_path = os.path.join(in_dir,
file_name)
                  with open(image_path, 'rb') as
image_file:
```

```
                                                encoded   =
base64.b64encode(image_file.read())
                image = encoded.decode('utf-8')
                name_to_image[file_name] = image

    return name_to_image

def create_prompt( ❷ Создает мультимодальный промпт
    person_image, image_to_label):
    """ Создать промпт для сравнения изображений.

    Аргументы:
        person_image: изображение человека.
         image_to_label: изображение для назначения
метки.

    Выводит:
          Промпт для проверки, изображен ли один и
тот же человек
        на обеих иллюстрациях.
    """
    task = {'type':'text',
            'text':'На этих фотографиях один и тот
же человек ("Да"/"Нет")?'}
    prompt = [task]
    for image in [person_image, image_to_label]:
        image_url = {'url':f'
{image}'}
                image_msg  =  {'type':'image_url',
'image_url':image_url}
        prompt += [image_msg]
```

```
    return prompt

def call_llm(ai_key, prompt):                    ❸
```

Генерирует ответ на промпт

```
        """Вызвать языковую модель для обработки
промпта с локальными изображениями.

    Аргументы:
        ai_key: ключ для доступа к OpenAI.
            prompt: промпт, объединяющий текст и
локальные изображения.

    Выводит:
        Ответ языковой модели.
    """
    headers = {
        'Content-Type': 'application/json',
        'Authorization': f'Bearer {ai_key}'
    }
    payload = {
        'model': 'gpt-4o',
        'messages': [
            {'role': 'user', 'content': prompt}
            ],
        'max_tokens':1
        }
    response = requests.post(
         'https://api.openai.com/v1/chat/completions
',
```

```
        headers=headers, json=payload)
    return response.json()['choices'][0]['message']
['content']

if __name__ == '__main__':                                    ❹
```

Отмечает изображения с людьми

```
        parser = argparse.ArgumentParser()                    ❺
```

Параметры командной строки

```
        parser.add_argument('peopledir', type=str,
help='Изображения людей')
        parser.add_argument('picsdir', type=str,
help='Изображения для тегирования')
        parser.add_argument('outdir', type=str,
help='Выходной каталог')
    args = parser.parse_args()

    people_images = load_images(args.peopledir)
    unlabeled_images = load_images(args.picsdir)

            for person_name, person_image in
people_images.items(): ❻ Перебирает людей
                    for un_name, un_image in
unlabeled_images.items(): ❼ Перебирает изображения
                prompt = create_prompt(person_image,
un_image)
            ai_key = os.getenv('OPENAI_API_KEY')
            response = call_llm(ai_key, prompt)
                    description = f'un_name versus
person_name?'
```

```
            print(f'description -> response')

                if response == 'Да':          ❽
```

Копирует изображение в случае совпадения

```
                                    labeled_name  =
f'person_name[:-4]un_name'
                                     source_path  =
os.path.join(args.picsdir, un_name)
                                    target_path  =
os.path.join(args.outdir, labeled_name)
                            shutil.copy(source_path,
target_path)
```

Сначала рассмотрим основную функцию ❹ . Как обсуждалось ранее, в качестве параметров командной строки пользователи указывают три каталога ❺ : с фотографиями, которые требуется тегировать; с изображениями людей, которых нужно отметить на фотографиях; и каталог для вывода.

На первом этапе мы загружаем все изображения для тегирования, а также все изображения людей, которых нужно найти. Для этого используется функция `load_images` ❶ . Она получает список всех файлов во входном каталоге, а затем рассматривает те, которые заканчиваются суффиксом `.png` (то есть все изображения в формате PNG). Как обсуждалось ранее, изображения необходимо закодировать в виде строк (применив кодировку base64), которые в конечном счете представлены с помощью кодировки UTF-8. Результатом работы функции `load_images` является словарь Python, сопоставляющий имена файлов с соответствующими закодированными изображениями. Этот словарь возвращается как результат функции.

Применив функцию `load_images` к каждому из двух входных каталогов, мы получим два словаря Python. Один сопоставляет имена файлов, содержащих изображения людей (которые, как мы договорились ранее, являются именами этих людей), с соответствующим закодированным изображением. Другой сопоставляет имена файлов изображений, которые необходимо тегировать, с закодированными изображениями.

Наша цель — сопоставить каждую фотографию, предназначенную для тегирования, со всеми людьми, которые на ней запечатлены. Поскольку мы используем промпты, одномоментно сравнивающие только две фотографии, то придется просмотреть все сочетания людей и изображений, которые требуется тегировать. Именно поэтому мы используем двойной вложенный цикл `for`: один перебирает людей ❻ , а другой — размечаемые изображения ❼.

Для каждого сочетания изображения и человека с помощью функции `create_prompt` создается мультимодальный промпт. Эта функция ❷ объединяет оба закодированных изображения вместе с текстовыми инструкциями в промпт. Текстовые инструкции («На фотографиях один и тот же человек (“Да”/“Нет”)?») определяют задачу, а также ожидаемый формат вывода («Да» или «Нет»). Каждый промпт отправляется в языковую модель GPT-4o через функцию `call_llm`. Как обсуждалось ранее, эта функция ❸ использует API запросов для отправки фотографий, находящихся в локальном хранилище, в GPT-4o. Если GPT-4o отвечает «Да», значит, рассматриваемый в данный момент человек есть на изображении, которое сейчас размечается.

Если человек есть на фотографии ❽ , то мы помечаем ее следующим образом. Берем имя человека (имя соответствующего файла изображения без расширения `.png`) и добавляем в начало имени файла изображения, которое

требуется тегировать. Затем файл этого изображения копируем в выходной каталог, уже с новым именем (которое содержит тег).

6.3.5. Проверка в работе

Пробуем. Если у вас есть настоящие фотографии из отпуска, на которых можно кого-то отметить, то можете использовать их. Если нет, то подходящие тестовые данные доступны по ссылке **Tagging**, размещенной на сайте книги в разделе дополнительных материалов к главе. Скачайте заархивированный файл и распакуйте. После распаковки в полученной папке должно быть три подкаталога:

- `people` — фотографии людей (в данном случае фото актеров из серии фильмов «Мстители»). Имена файлов содержат имена соответствующих актеров;
- `pics` — еще один набор фотографий (в данном случае дополнительные фото тех же актеров, что и в папке `people`), который нужно тегировать именами актеров;
- `processed` — пустая папка, которую можно использовать в качестве выходного каталога.

Предположим, что путь к месту хранения распакованной папки — `/tagging` (так, путь `/tagging/people` будет вести к подкаталогу с фотографиями людей, которых нужно найти). Выполните код, запустив из терминала такую команду:

```
python listing2.py /tagging/people /tagging/pics
/tagging/processed
```

СОВЕТ При запуске кода на платформе под управлением операционной системы Windows пути придется адаптировать. В частности, необходимо будет заменить символ / на \.

Во время обработки программа выводит информацию о том, есть ли конкретные люди на конкретных фотографиях. В примере данных есть фото двух человек, которых нужно найти, и четыре изображения, которые требуется тегировать. Это означает, что обработка должна занимать не более нескольких минут (обычно менее двух).

После завершения обработки зайдите в выходную папку. Там должны находиться размеченные изображения, к именам которых добавлены префиксы — имена людей, запечатленных на этих фото. Неплохо для нескольких строк кода на Python!

6.4. Генерация заголовков для видеороликов

Помимо множества фотографий (которые теперь можно автоматически тегировать благодаря коду, описанному в предыдущем разделе!), в отпуске вы сняли и немало видеороликов. Автоматически присвоенные имена файлов малоинформативны. В каком видеоролике показывается, как вы плаваете в океане? Было бы здорово присвоить видеороликам осмысленные названия, и это поможет быстрее находить нужные. Но у кого есть время вручную подписывать видеоролики? И снова для автоматического выполнения этой задачи можно использовать языковые модели.

6.4.1. Обзор

Мы разработаем систему, которая будет автоматически присваивать видеороликам подходящие заголовки. Эта система

использует языковую модель GPT-4o в фоновом режиме. Чтобы присвоить заголовки видеороликам, необходимо отправить мультимодальный промпт, содержащий кадры видео (то есть изображения) и текст, в котором указание для языковой модели — придумать заголовок. На рис. 6.3 показан пример промпта.

Промпт состоит из нескольких видеокадров (на рис. 6.3 мы видим только первый и последний кадры; три точки обозначают, что между ними есть еще кадры) и текстовой инструкции «Придумай краткий заголовок для видеоролика». Обратите внимание, что мы платим за каждый видеокадр, который отправляем в языковую модель GPT-4o. Следовательно, обработка видеоданных через GPT-4o быстро станет дорогостоящей!



Рис. 6.3. Мультимодальный промпт для обработки видеоролика: языковая модель получает инструкцию сгенерировать подходящий заголовок на основе подборки видеокадров

В качестве ответа языковая модель GPT-4o должна отправить логичное название. В примере, показанном на рис. 6.3, это может быть отсылка к автомобилям и, возможно, даже к местоположению (на кадрах указано белым текстом).

6.4.2. Кодирование видеокадров

Сначала необходимо обсудить форматы видео. В предыдущем разделе вы узнали, как прописывать в коде изображения,

находящиеся в локальном хранилище. Теперь мы расширим этот подход на работу с видеороликами. Наша цель — извлечь последовательность кадров. Однако видеоролики обычно хранятся не как последовательность кадров, а с использованием более эффективных кодировок. Для нас это означает, что сначала придется извлечь из видеоролика изображения.

Сделаем это с помощью библиотеки OpenCV. Это библиотека компьютерного зрения с открытым исходным кодом. Она предоставляет различные функции для компьютерного зрения, а также для обработки изображений и видеороликов в целом. Разумеется, функцию компьютерного зрения для нас выполнит модель GPT-4o. И все равно OpenCV будет полезна для извлечения кадров из видеороликов. Если вы еще не сделали этого, то самое время установить библиотеку OpenCV, следуя инструкциям в разделе 6.1.

Допустим, установка прошла успешно, и можно получить доступ к OpenCV из кода Python. Соответствующая библиотека Python называется `cv2` (это название будет часто встречаться в виде префикса в приведенных ниже фрагментах кода).

Для работы с видеороликами, находящимися в локальном хранилище, сначала нужно открыть соответствующий файл. Запустите этот код, чтобы открыть видео, сохраненное по пути `video_path`:

```
video = cv2.VideoCapture(video_path)
```

Используя переменную `video`, теперь можно прочитать содержимое видеоролика, применив метод `read`:

```
success, frame = video.read()
```

Результат состоит из кортежей с двумя компонентами: индикатором успешности выполнения (`success`) и видеокадром (`frame`). Индикатор указывает, удалось ли нам прочитать еще один кадр. Он перестает работать при достижении конца видеоролика. В этом случае мы не получаем действительный кадр, и значение индикатора становится `False`.

Предположим, что нам удалось прочитать еще один кадр. В этом случае мы превратим кадр в изображение, которое можно отправить языковой модели GPT-4o. Библиотека OpenCV предоставляет соответствующую функциональность:

```
_, buffer = cv2.imencode('.jpg', frame)
```

Функция `imencode` превращает видеокадр в изображение соответствующего типа. Здесь мы преобразуем кадр в изображение в формате JPEG. В полученном кортеже нам интересен второй компонент (`buffer`). Он содержит двоичное представление соответствующего изображения.

Эта ситуация знакома нам по предыдущему разделу: есть двоичное представление изображения и его нужно превратить в формат, подходящий для языковой модели GPT-4o. И снова мы сначала кодируем изображение как строку с помощью кодировки base64, а затем представляем эту строку через UTF-8:

```
encoded = base64.b64encode(buffer)
frame = encoded.decode('utf-8')
```

Полученный кадр корректно закодирован для добавления в промпт для языковой модели GPT-4o. Завершив обработку видеоролика, закройте соответствующий объект захвата видео, используя такой код:

```
video.release()
```

Далее мы соберем фрагменты кода и генерируем заголовки для произвольных видеороликов.

6.4.3. Полный код реализации

В коде листинга 6.3 создаются названия для видеороликов, находящихся в локальном хранилище. Единственный входной параметр — путь к видеоролику. Получив его, программа извлекает несколько видеокадров ❹ , а затем генерирует промпт, дающий модели GPT-4o указание придумать заголовок для видеоролика на основе выборки кадров. После отправки этого промпта языковой модели ее ответ будет содержать предлагаемый заголовок видеоролика.

Листинг 6.3. Генерация заголовка видеоролика с помощью языковых моделей

```
import argparse
import cv2
import base64
import openai
import time

client = openai.OpenAI()

def extract_frames(video_path):                    ❶
```

Извлекает видеокадры

```
    """ Извлекает кадры из видеоролика.

    Аргументы:
        video_path: путь к видеофайлу.
```

```
    Выводит:
        Список первых десяти видеокадров.
    """
    video = cv2.VideoCapture(video_path)
    frames = []
    while video.isOpened() and len(frames) <= 10:
        success, frame = video.read()
        if not success:
            break

        _, buffer = cv2.imencode('.jpg', frame)
        encoded = base64.b64encode(buffer)
        frame = encoded.decode('utf-8')
        frames += [frame]

    video.release()
    return frames

def create_prompt(frames):                                    ❷
```

Создает мультимодальный промпт

```
    """ Составить промпт для генерации заголовка к
видеоролику.

    Аргументы:
        frames: кадры видеоролика.

    Выводит:
          промпт, содержащий мультимодальные данные
(список).
```

```
    """
        prompt = ['Придумай точное название для
видеоролика.']
    for frame in frames[:10]:
        element = {'image':frame, 'resize':768}
        prompt += [element]
    return prompt


def call_llm(prompt):                                    ❸
```

Отправляет запрос в языковую модель

```
     """ Отправить запрос большой языковой модели и
вывести ответ.

    Аргументы:
        prompt: ввести запрос к языковой модели.

    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                    {'role':'user',
'content':prompt}
                    ]
                )
                                                 return
response.choices[0].message.content
```

```
        except:
            time.sleep(nr_retries * 2)
    raise Exception('Не удается выполнить запрос к
модели OpenAI!')

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
        parser.add_argument('videopath',  type=str,
help='Путь к видеофайлу')
    args = parser.parse_args()

      frames = extract_frames(args.videopath)     ❹
```

Создает название для видеоролика

```
    prompt = create_prompt(frames)
    title = call_llm(prompt)
    print(title)
```

Код извлекает видеокадры с помощью функции `extract_frames` ❶ . Как обсуждалось ранее, она использует библиотеку OpenCV, чтобы открыть видеоролик для извлечения кадров, а затем последовательно считывает каждый кадр. Для генерации названия видео мы будем использовать не более десяти кадров. Вот почему извлечение заканчивается после максимум десяти кадров (или меньшего количества, если видео очень короткое). Каждый извлеченный кадр кодируется в соответствии с требованиями GPT-4o (то есть изображения в формате JPEG, закодированные как строки). Результатом функции является список закодированных кадров.

В процессе генерации промпта ❷ мы объединяем соответствующие текстовые инструкции («Придумай краткий

заголовок для видео») с первыми десятью кадрами из видеоролика. Для отправки этих изображений вместе с инструкциями в модель GPT-4o снова воспользуемся оберткой Python ❸ . Как вариант, можно самостоятельно составлять запросы (как мы делали в предыдущем проекте). Ответ языковой модели должен содержать подходящий заголовок для нашего видеоролика.

Разумеется, мы отправляем только первые несколько кадров видеоролика. Если его содержание после этих нескольких кадров кардинально изменится, то заголовок может оказаться неоптимальным. Причина, по которой мы отправляем только десять кадров, — стоимость вычислений. Помните, что вы платите за каждое отправленное в промпт изображение! Отправка всех кадров длинных видеороликов, как правило, обходится непомерно дорого. Вот почему мы довольствуемся отправкой только небольшого подмножества видеокадров.

6.4.4. Проверка в работе

Протестируем наш генератор заголовков видеороликов! На сайте книги в разделе дополнительных материалов к этой главе есть ссылка **Cars**, по которой вы можете найти короткий видеоролик с камеры наблюдения, показывающий движение на оживленной дороге. Скачайте его на свой компьютер.

Откройте терминал и перейдите в каталог, содержащий код для этой главы. Предполагается, что видеоролик был загружен в тот же каталог (если нет, замените имя видеоролика — `cars.mp4` — на полный путь к нему).

Запустите следующую команду:

```
python listing3.py cars.mp4
```

Спустя несколько секунд вычислений должен появиться предлагаемый вариант заголовка видеоролика: например, «Дорожная ситуация на шоссе I-5 у SR 516 и 188-й улицы» (точное название может меняться при разных запусках из-за рандомизации).

Обратите внимание, что название содержит информацию (указание местоположения), которая доступна в видеоролике только в виде текста. Использование GPT-4o для извлечения текста из изображений может быть полезно в различных сценариях: например, для извлечения данных из бланков.

Резюме

- Языковая модель GPT-4o обрабатывает как изображения, так и текст.
- В промптах можно объединять фрагменты текста и изображения.
- Модель GPT-4o поддерживает несколько форматов изображений.
- Изображения можно указывать через общедоступный URL.
- Изображения, находящиеся в локальном хранилище, можно загружать в OpenAI.
- Модель GPT-4o обрабатывает изображения, закодированные в виде строк.
- Стоимость обработки изображений выше, чем обработки текста, и может зависеть от размера изображения.

Обработка изображений с низкой степенью детализации снижает затраты.

- Модуль `base64` может кодировать изображения в строки.
- Для отправки в языковую модель GPT-4o видео нужно разделять на кадры.
- Для извлечения кадров из видео можно использовать библиотеку OpenCV.

7. Анализ аудиоданных

В этой главе

- Транскрибация аудиозаписей.
- Перевод аудиоданных.
- Генерация речи.

Посмотрите любой качественный научно-фантастический телесериал или фильм, и вы нигде не увидите, чтобы люди при взаимодействии с компьютерами набирали команды на клавиатуре. Будь то «Звездный путь» или «Космическая одиссея 2001 года» (оба выпущены в 1960-х), люди разговаривают со своими компьютерами (а не набирают текст). И на то есть веские причины! Для большинства пользователей голос — самая естественная форма общения (потому что первая). Неудивительно, что люди представляли себе разговор с компьютерами задолго до того, как он стал технически возможен.

Теперь реальность догнала научную фантастику, и голосовые помощники, такие как Alexa от Amazon, Assistant от Google и Cortana от Microsoft (а также многие другие), распространены повсеместно. Новейшее поколение моделей распознавания (и генерации) речи достигло уровня, близкого к человеческому. И конечно, голосовое взаимодействие с компьютерами — лишь один из вариантов использования этой удивительной технологии.

В этой главе мы будем использовать новейшие модели OpenAI для транскрибации речи, ее перевода и генерации — и

все это в нескольких мини-проектах. Сначала мы выясним, что для транскрибации голосовых записей в текст требуется всего несколько строк кода на Python. После этого мы рассмотрим более сложные приложения, начиная с голосовой версии нашего интерфейса запросов на естественном языке к базе данных, с которым мы работали в главе 5. И если раньше текст вопросов приходилось набирать, то теперь их можно просто произнести, и система выдаст ответ. Наконец, вы увидите, как создать систему синхронного перевода, которая превратит ваш голосовой ввод в голосовой вывод на другом языке.

7.1. Подготовка

Прежде чем приступить к этим увлекательным проектам, необходимо выполнить несколько подготовительных действий. Вы будете записывать голосовой ввод со своего компьютера. Для этого нужен микрофон. В большинстве современных ноутбуков есть встроенный микрофон. Он не обязательно должен быть профессиональным; подойдет любой способ записи звука на вашем компьютере. Помимо микрофона, вам потребуется программное обеспечение для преобразования аудиозаписей в файлы, которое можно активизировать из кода Python. Для этого можно использовать библиотеку Python `sounddevice`. Чтобы установить нужную версию, выполните в терминале следующую команду:

```
pip install sounddevice==0.4
```

Эта библиотека взаимодействует с библиотекой Python `scipy`, которую тоже нужно установить. Выполните в терминале команду:

```
pip install scipy==1.11
```

Вместе библиотеки позволят вам записывать голосовой ввод (который затем можно будет транскрибировать, перевести или кратко пересказать с помощью моделей OpenAI).

Мы рассмотрели вопросы ввода данных, а как насчет вывода? В некоторых проектах нам потребуется не только слушать аудиозаписи, но и генерировать их! Для генерации речи мы снова воспользуемся моделями генеративного ИИ от компании OpenAI. Однако после того, как речь будет сгенерирована и сохранена в аудиофайле, потребуются библиотеки для воспроизведения ее на компьютере из кода Python. Мы будем использовать библиотеку `playsound`. Чтобы установить нужную версию, запустите в терминале эту команду:

```
pip install playsound==1.3
```

В некоторых операционных системах (в частности, macOS) дополнительно потребуется установить библиотеку `PyObjC`:

```
pip install PyObjC==10.0
```

Если вы еще не сделали этого, работая над прошлой главой, то установите библиотеку `requests` (позволяющую отправлять запросы напрямую к API OpenAI):

```
pip install requests==2.31.0
```

Отлично! Если при выполнении этих команд не появились сообщения об ошибках, значит, система подготовлена для обработки аудиоданных с помощью моделей-трансформеров OpenAI. В следующем разделе мы приступим к первому проекту.

7.2. Транскрибация аудиофайлов

Начав работать в компании Banana, вы поразились количеству совещаний. Их слишком много, быть на всех не получается, но пропустить что-то важное не хочется! К счастью, в Banana разумно подошли к вопросу и ввели общее правило: делать аудиозаписи всех рабочих встреч (с согласия участников). Однако прослушивание всех этих записей по-прежнему отнимает слишком много времени.

Было бы хорошо получать расшифровки записей, позволяющие быстро находить всю информацию, касающуюся вашего отдела, с помощью обычного текстового поиска. К сожалению, в Banana такие расшифровки не предусмотрены, а никто из коллег не хочет вести протоколы совещаний. Можно ли создавать такие расшифровки автоматически? В этом разделе вы увидите, что это не только возможно, но и несложно.

7.2.1. Транскрибация речи

Для преобразования речи в текст мы воспользуемся моделью Whisper от OpenAI. В отличие от моделей, с которыми мы работали до этого (в частности, моделей GPT), Whisper предназначена именно для транскрибации аудиозаписей.

Что такое модель Whisper

Whisper — это модель-трансформер, обученная на большом количестве аудиозаписей (более 680 000 часов записей, если быть точным!). Whisper обучалась на многоязычном корпусе аудиофайлов и поэтому поддерживает множество языков ввода, которые она преобразует в русский (то есть вы получаете одновременно расшифровку и перевод речи).

Как и в случае с моделями GPT, мы получим доступ к Whisper, используя библиотеку OpenAI для Python. А значит, не нужно ничего дополнительно настраивать на компьютере (при условии, что вы установили библиотеку OpenAI для Python, как описано в главе 3).

В этом разделе мы будем использовать модель Whisper для транскрибации аудиофайла с компьютера. Предположим, что аудиофайл изначально хранится на диске. Whisper поддерживает широкий спектр форматов файлов: MP3, MP4, MPEG, MPGA, M4A, WAV и WEBM. На момент написания книги размер файла был ограничен 25 Мбайт. Представим, что нужный файл есть и что путь к нему хранится в переменной `audio_path`. Теперь для преобразования его содержимого в текст нужно лишь ввести следующие строки кода на Python:

```
import openai
client = openai.OpenAI()

with open(audio_path, 'rb') as audio_file: ❶ Открывает аудиофайл

❷ Транскрибирует содержимое
transcription = client.audio.transcriptions.create(
    file=audio_file, model='whisper-1')
```

Сначала ❶ необходимо открыть аудиофайл. Для этого можно выполнить команду `open` в Python. Обратите внимание, что в качестве параметра команды `open` используется параметр `rb`. Он указывает Python, что мы хотим прочитать файл (`r`) и что открываем двоичный файл (`b`). Двоичным является файл, который не содержит читаемых символов. Звуковые файлы, наподобие того, который мы пытаемся открыть, обычно квалифицируются как двоичные. После обработки первой строки содержимое файла доступно через переменную `audio_file`.

На втором этапе ❷ выполняется фактическая транскрибация. Теперь мы используем другой эндпоинт, предназначенный для обработки аудиоданных. Из него мы

вызываем сервис транскрибации (`transcriptions.create`), используя два параметра:

- `file` — ссылка на файл для транскрибации;
- `model` — имя модели для транскрибации.

Мы ссылаемся на ранее открытый файл (`audio_file`) и выбираем `whisper-1` в качестве модели для транскрибации. Результатом является объект, содержащий расшифрованный текст и метаданные о процессе транскрибации. Доступ к тексту можно получить через поле `text` (то есть через `transcription.text`). После распаковки в полученной папке должны находиться три каталога.

Как видите, для транскрибации текста требуется всего несколько строк кода на Python! В следующем подразделе мы используем этот код для создания простого сервиса транскрибации.

7.2.2. Полный рабочий код

В листинге 7.1 содержится код простой программы транскрибации. Фактически транскрибация происходит в функции `transcribe` ❶ . По сути, мы обсуждали этот код в предыдущем подразделе. Получив в качестве входных данных аудиофайл, функция возвращает расшифрованный текст.

Листинг 7.1. Транскрибация аудиофайлов в текст

```
import argparse
import openai

client = openai.OpenAI()
```

```
def transcribe(audio_path):      ❶ Транскрибирует аудио в текст
    """Транскрибировать аудиофайл в текст.

    Аргументы:
        audio_path: путь к аудиофайлу.

    Выводит:
        транскрибированный текст.
    """
    with open(audio_path, 'rb') as audio_file:
                                transcription       =
client.audio.transcriptions.create(
        file=audio_file, model='whisper-1')
    return transcription.text


if __name__ == '__main__':       ❷ Функция main

    parser = argparse.ArgumentParser()
        parser.add_argument('audiopath',  type=str,
help='Путь к аудиофайлу')
    args = parser.parse_args()

    transcript = transcribe(args.audiopath)
    print(transcript)
```

Функция `main` ❷ в качестве входных данных считывает путь к аудиофайлу (который должен содержать речь). После вызова функции `transcriptions.create` она выводит расшифрованный текст на экран.

7.2.3. Проверка в работе

Для тестирования необходимо сначала получить аудиофайл с записанной речью. Можно использовать любой такой файл (в том числе запись ваших рабочих совещаний, если она есть) при условии, что он соответствует ограничениям по формату и размеру, описанным в подразделе 7.2.1. Однако помните, что вы платите за каждую минуту обработанных аудиоданных! На момент написания книги использование модели Whisper через библиотеку OpenAI стоило 0,006 доллара за минуту (более актуальную информацию о ценах можно найти на сайте **https://openai.com/pricing**). Следовательно, обработка длинных записей может оказаться дорогостоящей.

Если вам не хочется использовать собственную запись, то можете взять другую — из дополнительных материалов к этой главе. Запись доступна по ссылке **Audio**. По умолчанию файл должен называться `QuoteFromTheAlchemist.mp3`.

Там же, в дополнительных материалах к главе, вы можете найти ссылку на файл с кодом из листинга 7.1 (**Listing 1**). Скачав файл, перейдите в соответствующий репозиторий в терминале. Предположим, вы загрузили аудиофайл в текущий каталог, теперь выполните в терминале следующую команду для создания расшифровки файла примера:

```
python listing1.py QuoteFromTheAlchemist.mp3
```

Если все получится, то в терминале должен отобразиться такой вывод (для файла примера с сайта):

```
На том самом месте, где ты сейчас стоишь, я сам два
года назад несколько раз видел один и тот же сон.
```

Нажмите на файл примера, прослушайте его и убедитесь, что расшифровка точна!

Далее мы интегрируем расшифровку речи в более сложные приложения.

7.3. Голосовые запросы к реляционным базам данных

Анализировать табличные данные — интересное занятие! Значительная часть вашей работы в компании Banana состоит в том, чтобы изучать таблицы данных, извлекать ценную информацию, а затем готовить соответствующие отчеты и визуализации. Вы используете интерфейс преобразования текста в SQL (см. главу 5) для автоматического перевода текстовых вопросов в формальные запросы (на языке SQL), их выполнения и представления результатов запроса. Это упрощает анализ данных и выполняется быстрее, чем написание сложных SQL-запросов с нуля.

Однако есть проблема: во время анализа данных вам лучше думается, когда вы расхаживаете по кабинету. Вот только, чтобы ввести запрос, каждый раз приходится возвращаться к столу. Нельзя ли изменить интерфейс запросов так, чтобы он поддерживал голосовой ввод вместо текстового?

Оказывается, действительно можно! В этом разделе вы узнаете, как использовать модели OpenAI для создания простого голосового интерфейса запросов к табличным данным.

7.3.1. Подготовка

Мы создадим интерфейс голосовых запросов, который будет обрабатывать устные вопросы о табличных данных. Он представляет собой расширение интерфейса запросов, который мы обсуждали в главе 5. Предположим, что устные вопросы касаются данных, хранящихся в SQLite — популярной системе

для обработки запросов к реляционным базам данным. Краткое введение в SQLite и инструкции по установке изложены в главе 5. Чтобы протестировать приведенный ниже код, сначала необходимо установить систему баз данных SQLite.

Система SQLite обрабатывает запросы, сформулированные на SQL (языке структурированных запросов). К счастью, вам не потребуется самостоятельно писать SQL-запросы (их напишет языковая модель). Однако модели неидеальны и иногда могут выдавать некорректные запросы. Чтобы выявлять такие случаи, полезно иметь опыт работы с SQL. Краткое введение в SQL представлено в главе 5. Более подробную информацию можно найти на сайте **www.databaselecture.com**.

Наш интерфейс голосовых запросов обрабатывает устные вопросы, поэтому необходимо убедиться, что микрофон работает. Кроме того, перед выполнением приведенного ниже кода удостоверьтесь, что у интерфейса голосовых запросов есть все необходимые разрешения для доступа к микрофону.

7.3.2. Обзор

Интерфейс голосовых запросов обрабатывает устные вопросы о табличных данных, хранящихся в базе SQLite. Например, загрузив базу с информацией о продажах компьютерных игр, можно задать такие вопросы:

- «Сколько игр компания Activision продала в 2023 году?»;
- «Сколько экшен-игр было выпущено в период с 2019 по 2021 год?».

Получив устный вопрос, интерфейс голосовых запросов выполняет следующие действия.

1. Транскрибирует устный вопрос в текст.

2. Переводит текстовый вопрос в SQL-запрос.

3. Обрабатывает SQL-запрос к данным с помощью SQLite.

4. Отображает результат запроса пользователю.

На рис. 7.1 более подробно показаны разные этапы обработки. Процесс выполняется для каждого устного вопроса.

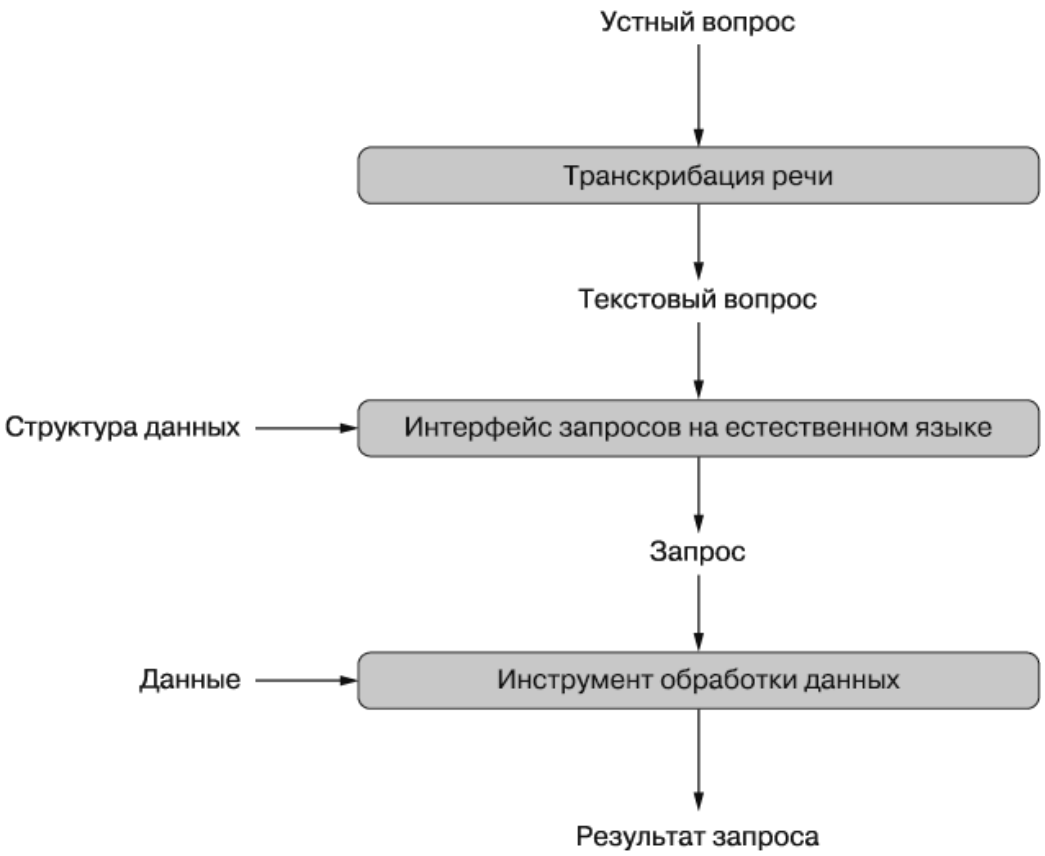


Рис. 7.1. Интерфейс голосовых запросов транскрибирует устные вопросы в текст, переводит текстовые вопросы в SQL-запросы, наконец, обрабатывает эти запросы и выводит результат

7.3.3. Запись аудио

Работая с приложением расшифровки аудиозаписей, мы предполагали, что аудиозапись уже доступна. В этом проекте мы будем отправлять голосовые запросы многократно. Следовательно, нам придется записывать их. Как это сделать в Python? Начнем с того, что для этой цели потребуется импортировать две библиотеки:

```
import sounddevice      ❶ Записывает аудио
import scipy.io.wavfile ❷ Сохраняет файлы в формате .wav
```

Библиотека `sounddevice` ❶ содержит множество полезных функций для записи аудиоввода с микрофона. Что мы будем делать с записями? Хранить их на диске в формате `.wav`. В предыдущем разделе описывалось, как выполнить расшифровку аудиоданных, сохраненных в этом формате. Здесь используется вторая библиотека, `scipy` ❷: она позволяет сохранять записи в формате `.wav` на диск.

При записи необходимо определить два важных момента:

- с какой частотой дискретизации следует считывать ввод с микрофона;
- сколько секунд речи записывать.

Мы будем записывать речь в течение 5 секунд. Этого времени, как правило, достаточно для большинства голосовых запросов. Если запись завершается слишком рано или если вы, произнеся голосовой запрос, часто ждете ее окончания, то попробуйте изменить настройки. При более сложной реализации запись шла бы непрерывно или останавливалась на месте паузы в речи. Чтобы не усложнять механизм записи, будем просто записывать звук в течение заранее заданного промежутка времени для каждого голосового запроса.

Что касается частоты дискретизации — то есть количества точек аудиоданных, сохраняемых в секунду, — мы выберем 44 100 Герц. Это стандарт для записей качества компакт-диска. Общее количество *фреймов* — количество точек аудиоданных, полученных в сумме, — будет равно 44 100, умноженным на количество секунд записи (в нашем случае это 5 секунд). Сохраним количество фреймов и частоту дискретизации во вспомогательных переменных:

```
sample_rate = 44100
nr_frames = 5 * sample_rate
```

Теперь мы готовы записывать с помощью функции `rec` библиотеки `sounddevice`:

```
recording = sounddevice.rec(                    ❶ Настраивает запись
    nr_frames, samplerate=sample_rate, channels=1)
sounddevice.wait()                              ❷ Ждет завершения записи
```

Первая команда ❶ начинает запись через микрофон, передавая в качестве входных данных общее количество кадров для записи, а также частоту дискретизации. Количество каналов (третий параметр в вызове функции) зависит от микрофона, используемого для записи. Если у микрофона более одного канала, то попробуйте указать в нем большее значение. После начала записи необходимо подождать, пока пройдет заданное время. Мы добьемся этого с помощью команды `wait` ❷.

В результате выполнения приведенного выше кода в переменной `recording` оказываются записанные аудиоданные. Как обсуждалось ранее, запись необходимо сохранить на диск как файл с расширением `.wav`. Для этого достаточно одной команды из библиотеки `scipy`:

```
scipy.io.wavfile.write(output_path,     sample_rate,
recording)
```

Вот и все! Мы записали несколько секунд звука и сохранили их в файле на диске.

7.3.4. Полный рабочий код

В листинге 7.2 приведен код интерфейса голосовых запросов. Помимо стандартных библиотек `openai` и `argparse`, мы импортируем ❶ библиотеки для обработки аудио (`sounddevice` и `scipy`), а также библиотеку `sqlite3` (для обработки SQL-запросов) и библиотеку `time` (для ожидания заданного количества времени (для голосового ввода)).

Листинг 7.2. Отправка запросов к базе данных SQLite с помощью голосовых команд

```
import argparse                                          ❶
```

Импортирует библиотеки

```
import openai
import re
import scipy.io.wavfile
import sounddevice
import sqlite3
import time

client = openai.OpenAI()


def get_structure(data_path):                            ❷
```

Извлекает схему базы данных

```
    """Извлечь структуру из базы данных SQLite.
```

```
    Аргументы:
        data_path: путь к файлу данных SQLite.

    Выводит:
        текстовое описание структуры базы данных.
    """
    with sqlite3.connect(data_path) as connection:
        cursor = connection.cursor()
                  cursor.execute("select  sql  from
sqlite_master where type = 'table';")
        table_rows = cursor.fetchall()
        table_ddls = [r[0] for r in table_rows]
        return '\n'.join(table_ddls)


def record(output_path):                                ❸
```

Записывает аудиофайл

```
    """Записать аудио и сохранить в файле .wav.

    Аргументы:
        output_path: сохранить аудиозапись здесь.
    """
    sample_rate = 44100
    nr_frames = 5 * sample_rate
    recording = sounddevice.rec(
                nr_frames,  samplerate=sample_rate,
channels=1)
    sounddevice.wait()
                scipy.io.wavfile.write(output_path,
sample_rate, recording)
```

```
def transcribe(audio_path):                                    ❹
```

Транскрибирует аудиозапись

```
    """ Транскрибировать аудиофайл в текст.

    Аргументы:
        audio_path: путь к аудиофайлу.

    Выводит:
        транскрибированный текст.
    """
    with open(audio_path, 'rb') as audio_file:
                                    transcription    =
client.audio.transcriptions.create(
            file=audio_file, model='whisper-1')
        return transcription.text


def create_prompt(description, question): ❺
```

Создает запрос для преобразования текста в формат SQL

```
    """ Сгенерировать промпт для перевода вопроса в
SQL-запрос.

    Аргументы:
          description: текстовое описание структуры
базы данных.
         question: вопрос о данных на естественном
языке.
```

```
    Выводит:
        Промпт для перевода вопроса.
    """
    parts = []
    parts += ['База данных:']
    parts += [description]
    parts += ['Переведи этот вопрос в SQL-запрос:']
    parts += [question]
    parts += ['SQL-запрос:']
    return '\n'.join(parts)


def call_llm(prompt):                                    ❻
```

Переводит в SQL

```
     """ Отправить запрос большой языковой модели и
вывести ответ.

    Аргументы:
        prompt: ввести запрос к языковой модели.

    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                      {'role':'user',
'content':prompt}
```

```
                    ]
                )
                                                    return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')


def process_query(data_path, query):                ❼
```

Обрабатывает SQL-запрос

```
        """ Обрабатывает SQL-запрос и возвращает
результат.

    Аргументы:
        data_path: путь к файлу данных SQLite.
             query: обработать этот запрос к базе
данных.

    Выводит:
        результат запроса.
    """
    with sqlite3.connect(data_path) as connection:
        cursor = connection.cursor()
        cursor.execute(query)
        table_rows = cursor.fetchall()
               table_strings = [str(r) for r in
table_rows]
        return '\n'.join(table_strings)
```

```
if __name__ == '__main__':                                    ❽
```

Обрабатывает голосовые запросы

```
    parser = argparse.ArgumentParser()
           parser.add_argument('dbpath',   type=str,
help='Путь к данным SQLite')
    args = parser.parse_args()

    data_structure = get_structure(args.dbpath)

    while True:                                               ❾
```

Основной цикл

```
          user_input = input('Нажмите Enter, чтобы
начать запись (введите quit,
                                    чтобы выйти из
приложения).')
        if user_input == 'quit':
            break

          audio_path = 'question.wav'                         ❿
```

Расшифровывает голосовой ввод

```
        record(audio_path)
        question = transcribe(audio_path)
        print(f'Вопрос: {question}')

            prompt  =  create_prompt(data_structure,
question)  ⓫ Преобразование в SQL
        answer = call_llm(prompt)
```

```
        query = re.findall('```sql(.*)```', answer,
re.DOTALL)[0]
        print(f'SQL: {query}')


        try:                                                    ⓬
```

Выполняет SQL-запрос

```
            answer = process_query(args.dbpath,
query)
            print(f'Ответ: {answer}')
        except:
            print('Ошибка обработки запроса!
Попробуйте переформулировать.')
```

Теперь обсудим функции, представленные в этом листинге 7.2.

Мы обрабатываем голосовые запросы, которые адресованы данным, хранящимся в реляционной базе данных. Чтобы переводить голосовые команды в формальные запросы, сформулированные на языке SQL, необходимо иметь определенные сведения о структуре базы данных. В частности, нужно знать имена таблиц данных и их столбцов (то есть схему базы). Функция `get_structure` ❷ извлекает команды, примененные для создания схемы. Эти команды содержат имена таблиц и столбцов, а также типы данных, связанные со столбцами таблиц. Мы используем эти команды как часть промпта, дав языковой модели указание переводить вопросы в SQL-запросы.

Прежде чем вопросы можно будет переводить, их сначала необходимо записать через микрофон. Для этого предназначена функция `record` ❸. Она использует библиотеку `sounddevice` для записи 5 последовательных секунд ввода аудиоданных через микрофон. Полученная аудиозапись

сохраняется на диске как файл в формате `.wav` по пути, который указан как входной параметр функции (`output_path`). Строго говоря, сохранять вводимые аудиоданные в виде файла не обязательно (можно обрабатывать их непосредственно в памяти). Но сохраненная на диск аудиозапись может пригодиться при отладке. Если системе не удастся преобразовать голосовой ввод в соответствующие запросы, то вы сможете сами прослушать аудиофайл, чтобы оценить уровень фонового шума и общее качество звука. Если микрофон настроен неправильно (распространенная проблема), то аудиозаписи будут содержать только тишину.

После записи входных данных через микрофон их нужно сначала преобразовать в текст. Для этого используется функция `transcribe` ❹. Получив путь к аудиофайлу (в данном случае — к записи с микрофона), она возвращает расшифровку, сгенерированную с помощью модели Whisper от OpenAI (той же, что мы использовали ранее).

Далее необходимо перевести вопросы в формальные SQL-запросы. Разумеется, для этой задачи используются языковые модели. Функция `create_prompt` ❺ генерирует подходящий промпт. Он содержит ранее извлеченное описание базы данных, транскрибированный вопрос и описание задачи. Функция `call_llm` ❻ вызывает для перевода вопросов языковую модель GPT-4o, используя в качестве входных данных упомянутый ранее промпт. Наконец, функция `process_query` ❼ обрабатывает получившиеся запросы в базе данных и возвращает результат запроса.

Пора собрать все вместе! Наш интерфейс голосовых запросов принимает в качестве входных данных путь к файлу базы данных SQLite ❽. После извлечения схемы базы мы входим в главный цикл ❾. Каждая итерация обрабатывает один голосовой запрос (если только пользователь не введет

команду `quit`; в этом случае программа завершится). Для простоты мы ждем, пока пользователь нажмет клавишу **Enter**, прежде чем записывать голосовой ввод (более сложная версия записывала бы непрерывно). После этого записываем голосовой ввод с микрофона. Мы выводим транскрибированный вопрос и сохраняем саму запись на диске в файле `question.wav` ❿ . Затем переводим транскрибированный текст в запрос ⓫ , выполняем его ⓬ (здесь потребуется обработка исключений на случай некорректных запросов!) и показываем результат пользователям.

7.3.5. Проверка в работе

Файл с кодом из листинга 7.2 доступен по ссылке **Listing 2**, которую можно найти на сайте книги в разделе дополнительных материалов к этой главе. Скачайте код и перейдите в содержащую его папку в терминале.

Помимо кода, для тестирования интерфейса голосовых запросов нам потребуется база данных SQLite. В главе 5 мы обсуждали, как настроить демонстрационную базу, содержащую информацию о продажах компьютерных игр. Предположим, что она хранится в одной папке с кодом и называется `games.db` (конечно, для проверки голосового интерфейса вы можете использовать любую другую базу данных SQLite). Теперь введите в терминале следующую команду:

```
python listing2.py games.db
```

Обновите путь к файлу базы, указав тот, к которому хотите получить доступ. В зависимости от операционной системы и настроек безопасности, возможно, вам придется разрешить

приложению доступ к микрофону. После этого нажмите **Enter** и задайте вопрос. Например, если вы работаете с базой данных игр, то можете спросить: «Сколько игр было продано в 2007 году?» или «Сколько игр каждого жанра было выпущено?». Должен отобразиться примерно такой вывод:

```
Нажмите Enter, чтобы начать запись (введите quit, чтобы выйти из приложения).
Вопрос: Сколько игр каждого жанра было выпущено?
SQL: SELECT genre, COUNT(*) as num_games
FROM games
GROUP BY genre
Ответ: ('Экшен', 3316)
('Приключения', 1286)
('Файтинг', 848)
('Жанр', 1)
('Прочее', 1739)
('Платформенная игра', 886)
('Головоломка', 582)
('Гонки', 1249)
('Ролевая игра', 1488)
('Шутер', 1310)
('Симулятор', 867)
('Спортивные игры', 2346)
('Стратегия', 681)
```

В этом выводе содержится транскрибированный вопрос, преобразованный в SQL-запрос, и результат запроса (или сообщение об ошибке, если запрос не может быть выполнен). Очевидно, что путь от голосового вопроса до результата запроса долог! Ошибка в записи, транскрибации или переводе приведет к неверным результатам. Прежде чем доверять

результату запроса, обязательно проверьте дополнительный вывод, чтобы убедиться, что система не ошиблась.

> СОВЕТ Если голосовой интерфейс выдает только бессмыслицу, то проверьте записи в файле question.wav. Если вы ничего не слышите, то убедитесь, что у вашего приложения есть доступ к микрофону. По умолчанию приложения обычно не имеют такого доступа (чтобы злоумышленникам было сложнее использовать их для слежки за вами с помощью вредоносного ПО). Для предоставления доступа необходимо обновить настройки безопасности.

7.4. Перевод речи в речь

Филиал компании Banana в Париже начал изучать языковые модели и потенциальные приложения для задач анализа данных. Вы заслужили репутацию эксперта в этой теме, и менеджер просит вас проконсультировать французскую команду о том, с чего начать. Есть лишь одна маленькая проблема: вы не говорите по-французски. Услышав, что в парижском филиале компании большинство собраний персонала проводится на французском языке, вы собираетесь отказаться от задания. Но, поразмыслив, понимаете, что все может быть не так уж и страшно. Да, вы не говорите по-французски, но модель GPT-4o точно говорит! Можно ли использовать языковые модели для перевода с одного языка на другой?

Оказывается, можно. В этом разделе мы создадим инструмент перевода, который будет получать входные данные в речевой форме на одном языке и предоставлять голосовой вывод на другом языке. Поскольку инструмент предоставляет вывод в голосовой форме, вам даже не придется отрабатывать

французское произношение. Просто говорите на своем языке и ждите, пока инструмент выполнит устный перевод. Таким образом, вы сможете сотрудничать с французскими коллегами и одновременно демонстрировать возможности современных языковых моделей!

7.4.1. Обзор

Наш инструмент перевода обрабатывает голосовой ввод. Как и прежде, мы будем использовать модель Whisper от OpenAI для транскрибации речи в текст. Затем с помощью модели GPT-4o будем переводить текст на другой язык. В наших примерах в качестве целевого языка будет использоваться французский. Однако благодаря удивительным возможностям таких моделей, как GPT-4o, инструмент не ограничится именно этим языком! Он позволит пользователям указывать целевой язык в качестве входных данных, и эта информация добавится в промпт для модели — как инструкция, на какой язык переводить.

После того как текстовый перевод будет сгенерирован, нам потребуется его голосовая версия. Оказывается, можно использовать еще одну модель OpenAI для преобразования текста в голосовой вывод на различных языках. На рис. 7.2 показан полный конвейер обработки, начиная с голосового ввода на первом языке и заканчивая голосовым выводом на втором.

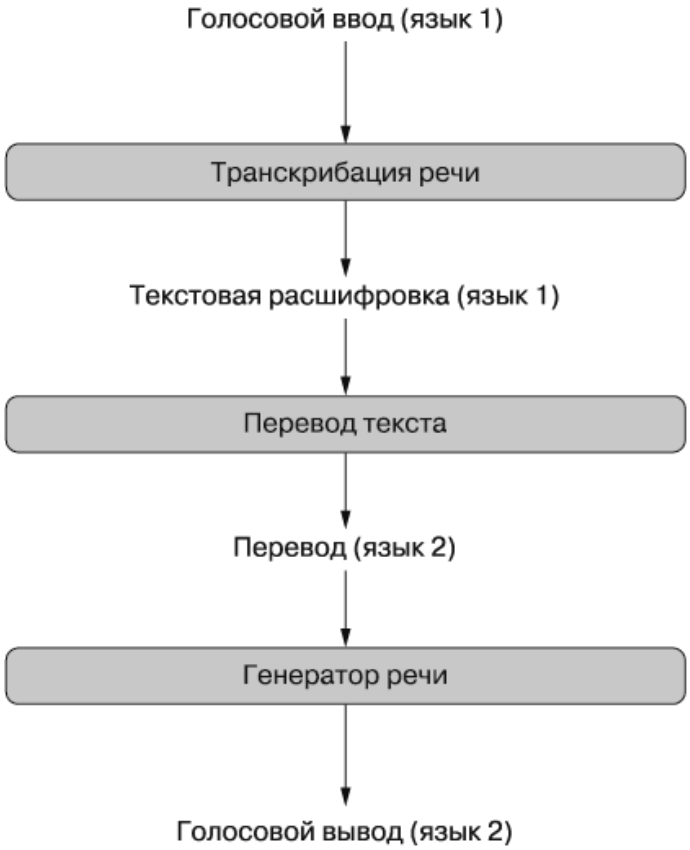


Рис. 7.2. Наш инструмент перевода записывает голосовой ввод на первом языке, транскрибирует ввод в текст, переводит этот текст на второй язык и, наконец, генерирует голосовой вывод

7.4.2. Генерация речи

Конвейер на рис. 7.2 требует выполнения нескольких преобразований. В разделе 7.2 мы выяснили, как данные из голосового ввода можно преобразовать в текст. Переводить текст с помощью языковых моделей относительно легко (попросите модель GPT-4o перевести с одного языка на другой, и она это сделает). Однако мы пока не можем преобразовывать письменный текст (например, на французском языке) в голосовой вывод. Далее мы обсудим, как это сделать.

OpenAI (как и другие провайдеры) предлагает несколько моделей преобразования текста в речь (text-to-speech, TTS).

Такие модели принимают в качестве входных данных письменный текст, а в выводе выдают голосовую версию. Этот фрагмент кода превращает в речь содержимое текстовой строки (хранящейся в переменной `speech_text`):

```
import openai
client = openai.OpenAI()

response = client.audio.speech.create(
    model='tts-1', voice='alloy',
    input=speech_text)
```

В этом примере мы используем новый эндпоинт (`audio.speech`) и настраиваем метод `create` с помощью трех параметров:

- `model` — имя модели, используемой для генерации голосового вывода. Мы используем модель преобразования текста в речь `tts-1` компании OpenAI;
- `input` — мы генерируем устный вывод для этого текста. Отправьте текст на любом из языков, поддерживаемых моделью (**https://github.com/openai/whisper**);
- `voice` — для озвучивания речи можно выбирать разные голоса. Здесь мы используем вариант `alloy`.

Этого достаточно для генерации голосового вывода с помощью OpenAI. Мы уже знаем, как транскрибировать речь и как переводить текст с одного языка на другой, и теперь у нас есть все необходимое, чтобы запрограммировать инструмент перевода.

А какова цена?

На момент написания книги компания OpenAI брала плату 1,5 цента за 1000 токенов при генерации текста с помощью модели TTS и вдвое больше за версию высокого качества (TTS HD). Скорее всего, стоимость со временем изменится, поэтому обязательно посетите сайт с ценами OpenAI (**https://openai.com/pricing**), чтобы получить актуальную информацию.

7.4.3. Полный рабочий код

В листинге 7.3 представлен полный рабочий код нашего инструмента перевода. Начнем с обсуждения импортируемых библиотек ❶. Помимо библиотеки `openai` и модуля `argparse`, использовавшихся во всех предыдущих проектах, мы импортируем библиотеки `sounddevice` и `scipy` для записи и сохранения аудиофайлов, а также модуль `time` для ограничения времени записи.

Листинг 7.3. Перевод голосового ввода на другой язык

```
import argparse                                    ❶
```

Импортирует библиотеки

```
import openai
import playsound
import requests
import scipy.io.wavfile
import sounddevice
import time

client = openai.OpenAI()
```

```
def   record(output_path):                                    ❷
```

Записывает аудиофайл

```
    """Записать аудио и сохранить в файле .wav.

    Аргументы:
        output_path: сохранить аудиозапись здесь.
    """
    sample_rate = 44100
    nr_frames = 5 * sample_rate
    recording = sounddevice.rec(
                 nr_frames,  samplerate=sample_rate,
channels=1)
    sounddevice.wait()
                 scipy.io.wavfile.write(output_path,
sample_rate, recording)

def transcribe(audio_path):                                   ❸
```

Транскрибирует аудиозапись

```
    """Транскрибировать аудиофайл в текст.

    Аргументы:
        audio_path: путь к аудиофайлу.

    Выводит:
        транскрибированный текст.
    """
    with open(audio_path, 'rb') as audio_file:
                                    transcription    =
client.audio.transcriptions.create(
```

```
            file=audio_file, model='whisper-1')
        return transcription.text
```

❹ Создает промпт для перевода

```
def create_prompt(to_translate, to_language):
    """Сгенерировать промпт для перевода текста на
целевой язык.

    Аргументы:
        to_translate: перевести этот текст.
        to_language: перевести текст на этот язык.

    Выводит:
        Переведенный текст.
    """
    parts = []
        parts += [f'Перевести этот текст на
{to_language}:']
    parts += [to_translate]
    parts += ['Переведенный текст:']
    return '\n'.join(parts)


def call_llm(prompt):                                ❺
```

Использует языковую модель

```
    """ Отправить запрос большой языковой модели и
вывести ответ.

    Аргументы:
        prompt: введите запрос к языковой модели.
```

```
    Выводит:
        Ответ языковой модели.
    """
    for nr_retries in range(1, 4):
        try:
                                        response   =
client.chat.completions.create(
                model='gpt-4o',
                messages=[
                                      {'role':'user',
'content':prompt}
                    ]
                )
                                                return
response.choices[0].message.content
        except:
            time.sleep(nr_retries * 2)
     raise Exception('Не удается выполнить запрос к
модели OpenAI!')


def generate_speech(speech_text):                    ❻
```

Генерирует речь

```
    """ Генерирует речь из конкретного текста.

    Аргументы:
          speech_text: сгенерировать речь из этого
текста.

    Выводит:
        результат запроса.
    """
```

```
    response = client.audio.speech.create(
        model='tts-1', voice='alloy',
        input=speech_text)
    return response.content

if __name__ == '__main__':                                    ❼
```

Преобразует речь с одного языка на другой

```
    parser = argparse.ArgumentParser()
        parser.add_argument('tolanguage',  type=str,
help='Целевой язык')
    args = parser.parse_args()

    while True:                                               ❽
```

Основной цикл

```
          user_input = input('Нажмите Enter, чтобы
начать запись (введите quit,
                                    чтобы выйти из
приложения.')
        if user_input == 'quit':
            break

         audio_path = 'to_translate.wav'                      ❾
```

Транскрибирует входные данные

```
        record(audio_path)
        to_translate = transcribe(audio_path)
                    print(f'Оригинальный   текст:
{to_translate}')
```

❿ Переводит на целевой язык

```
            prompt = create_prompt(to_translate,
args.tolanguage)
        translated = call_llm(prompt)
        print(f'Текст перевода: {translated}')


          speech = generate_speech(translated)    ⓫
```

Генерирует голосовой вывод

```
        with open('translation.mp3', 'wb') as file:
            file.write(speech)


         playsound.playsound('translation.mp3') ⓬
```

Воспроизводит сгенерированную речь

Библиотека `playsound` используется для воспроизведения аудиофайлов, сгенерированных моделями OpenAI. Поскольку мы генерируем речь через HTTP-интерфейс OpenAI, то импортируем библиотеку `requests` для создания HTTP-запросов. А теперь обсудим функции, используемые в листинге 7.3.

Как и в предыдущем проекте, аудиоданные записываются с микрофона. Функция `record` ❷ записывает 5 секунд аудиоввода и сохраняет их на диске в файле формата `.wav`. Функция `transcribe` ❸ транскрибирует этот аудиоввод в текст. Обе функции мы подробно рассмотрели при работе с предыдущими проектами этой главы.

Функция `create_prompt` ❹ генерирует промпт для перевода. Как и в предыдущих проектах, промпт содержит описание задачи и все релевантные входные данные. В данном

случае мы хотим перевести с исходного языка (английского) на целевой язык (французский). Обратите внимание, что целевой язык указан как входной параметр (`to_language`). Этот параметр соответствует текстовому фрагменту, описывающему желаемый язык вывода. В простейшем случае это может быть название языка (например, французский). Кроме того, пользователи могут даже попросить озвучить текст на определенном диалекте (скажем, швабский диалект немецкого языка) или в определенном стиле (например, шекспировский английский). Целевой язык интегрируется в описание задачи, которое появляется в промпте вместе с текстом для перевода.

Обратите внимание, что исходный язык указывать не требуется. Предполагается, что языковая модель способна распознать язык входного текста (иначе бессмысленно надеяться, что она его переведет).

После вызова функции `call_llm` ❺ с передачей ей промпта мы должны получить переведенный текст. Функция `generate_speech` ❻ генерирует соответствующую речь, используя подход, который мы рассмотрели в предыдущем подразделе.

Приложение-переводчик ❼ ожидает в качестве входных данных текст, описывающий целевой язык. Этот параметр представляет собой строку, которая может содержать произвольный текст. Она просто ставится вместо метки-заполнителя в промпте, используемом для перевода. В главном цикле ❽ пользователи нажимают клавишу **Enter** и после этого начинают говорить или вводят команду `quit`, чтобы завершить работу приложения.

При записи пользовательского ввода мы сначала сохраняем пятисекундную аудиозапись в файле с именем `to_translate.wav`, а затем транскрибируем ввод с помощью функции `transcribe` ❾. После этого мы используем языковую

модель GPT-4o для перевода входных данных на целевой язык ❿ , а затем генерируем речь из перевода ⓫ . Мы сохраняем сгенерированную речь на диске в файле формата `.mp3` (это означает, что последний вывод можно будет легко прослушать еще раз) и, наконец, с помощью библиотеки `playsound`, как вы уже догадались, воспроизводим сгенерированный звуковой файл.

7.4.4. Проверка в работе

Самое время протестировать наш переводчик! Его код доступен по ссылке **Listing 3**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе. Скачайте код и в терминале перейдите в папку с ним. Можно выбрать целевой язык для перевода. Конечно, качество перевода и звукового вывода может варьироваться в зависимости от вашего выбора. В частности, и модель, используемая для расшифровки речи, и модель, выполняющая генерацию речи, поддерживают набор примерно из 60 распространенных языков. Транскрибация аудиоввода или генерация голосового вывода на менее распространенных языках может не сработать. Найдите в Интернете актуальный список поддерживаемых языков для расшифровки (**https://help.openai.com/en/articles/7031512-whisper-api-faq**) и генерации речи (**https://platform.openai.com/docs/guides/text-to-speech**). Пока что в соответствии со сценарием, описанным в начале раздела, мы выберем в качестве целевого языка французский. Чтобы запустить переводчик, введите в терминале команду:

```
python listing3.py "French"
```

Строго говоря, кавычки вокруг слова `French` не нужны. Но описание желаемого целевого языка может состоять из нескольких слов, поэтому в следующих примерах мы используем кавычки, чтобы избежать ошибок, если консоль неверно интерпретирует наш ввод как значения для нескольких параметров.

Как и в предыдущем проекте, приложению необходимо предоставить доступ к микрофону. Нажмите **Yes** (Да), если появится запрос на доступ к микрофону; в противном случае убедитесь, что настройки безопасности разрешают такой доступ. Вот отрывок диалога с инструментом перевода:

```
Нажмите Enter, чтобы начать запись (введите quit,
чтобы выйти из приложения).
Исходный текст: Добрый день, мои парижские коллеги.
Переведенный текст: Bonjour mes collègues à Paris.
Нажмите Enter, чтобы начать запись (введите quit,
чтобы выйти из приложения).
Исходный текст: Позвольте, я расскажу вам кое-что о
языковых моделях.
Переведенный текст: Laisse-moi t'apprendre quelque
chose à propos des modèles de langage.
```

Вы увидите расшифровку ввода и сгенерированный перевод. Кроме того, вы должны услышать голосовую версию перевода (если нет, то проверьте настройки аудиовыхода). Неплохо для нескольких строк кода на Python!

Перевод на французский кажется вполне подходящей задачей для нашего переводчика. Однако полагаю, что это не самый интересный сценарий его применения. Протестируем и другие варианты, чтобы показать возможности языковых моделей: посмотрим, можно ли перевести наш аудиоввод в

выражения, обладающие высоким стилем. В терминале введите следующую инструкцию:

```
python listing3.py "Английский язык, стиль Шекспира"
```

Вот что получится, если придать обычному приветствию гораздо более утонченную форму (это задание может стать хорошим введением в курс по языковым моделям для коллег из американского филиала Banana):

```
Нажмите Enter, чтобы начать запись (введите quit, чтобы выйти из приложения).
Исходный текст: Добрый день, мои парижские коллеги.
Переведенный текст: Внемлите, мои добрые соратники, приветствую вас!
Нажмите Enter, чтобы начать запись (введите quit, чтобы выйти из приложения).
Исходный текст: Позвольте, я расскажу вам кое-что о языковых моделях.
Переведенный текст: Внемлите, о благородные слушатели, дабы я поведал вам о таинствах языковых моделей.
```

Протестируйте его еще с несколькими целевыми языками! Возможности (почти) безграничны.

Резюме

- Модель Whisper от OpenAI транскрибирует голосовой ввод в текст.

- Доступ к тексту расшифровки можно получить через эндпоинт API-метода для транскрибации аудио.
- Стоимость расшифровки зависит от количества минут.
- Компания OpenAI предлагает несколько моделей, способных преобразовать текст в речь.
- Можно выбрать голос и качество генерируемой речи.
- Стоимость генерации речи зависит от количества токенов.

Часть III. Углубленное изучение

В этой части вы научитесь писать более эффективные и экономически выгодные приложения, использующие языковые модели.

В главе 8 мы обсудим несколько наиболее популярных поставщиков, помимо OpenAI, их модели и библиотеки. Ведь прежде, чем применять модели на больших наборах данных, крайне важно сравнить модели, предлагаемые разными поставщиками. Это позволяет найти оптимальный баланс между стоимостью и качеством в рамках конкретного сценария.

В главе 9 описываются методы оптимизации затрат на примере демонстрационного сценария. В ней мы рассмотрим такие темы, как промпт-инжиниринг, оптимальная настройка параметров конфигурации модели и дообучение — процесс, с помощью которого языковая модель адаптируется под выполнение одной конкретной задачи. Как показывается в главе, использование этих методов способно значительно повысить качество модели и снизить стоимость ее работы.

В главе 10 мы обсудим два популярных фреймворка для разработки сложных приложений на основе языковых моделей: LangChain и LlamaIndex. Оба можно эффективно использовать для анализа данных. В частности, в главе показаны варианты применения этих фреймворков для создания агентов. Этот подход позволяет языковым моделям решать сложные задачи анализа данных с помощью различных вычислительных инструментов.

8. Альтернативы GPT

В этой главе

- Claude от Anthropic.
- Command R+ от Cohere.
- Bard от Google.
- Модели от Hugging Face.

В предыдущих главах основное внимание мы уделяли GPT и другим моделям OpenAI. Однако это не единственная компания в отрасли. Наоборот, в настоящее время мы являемся свидетелями «кембрийского взрыва» языковых моделей, когда новые модели появляются каждую неделю. Прежде чем использовать модель в производственной среде, стоит убедиться, что она действительно подходит для решения задачи. В этой главе мы рассмотрим множество существующих альтернатив OpenAI и обсудим плюсы и минусы разных моделей, а также способы их использования.

В основе почти всех современных языковых моделей лежит схожая архитектура (архитектура трансформера). Однако модели от разных поставщиков могут различаться по способу обучения, способу представления текста или по тому, как они предлагаются и оцениваются. Все эти факторы могут повлиять на стоимость обработки и качество выходных данных для вашей конкретной задачи. Модели типа GPT-4o мощные и решат почти любую задачу. Но эта универсальность имеет свою цену: если есть небольшая специализированная модель,

обученная только для выполнения конкретной задачи, то, возможно, оптимальным выбором станет именно она.

В следующих разделах вы увидите, что многие из моделей доступны через интерфейсы, аналогичные интерфейсу OpenAI. Это хорошая новость: не придется изучать новую и сложную структуру каждый раз при выборе другой модели!

> **Что делать, если в этой главе не указана ваша любимая модель или поставщик**
>
> Если модель не упоминается в главе, это не означает, что она не окажется более подходящей альтернативой для вашей задачи. Ведь количество поставщиков и моделей постоянно растет, поэтому представить полный обзор всех доступных моделей невозможно. Интерфейсы разных поставщиков, как правило, похожи, поэтому вы без особых проблем сможете применять полученные знания при работе с другими моделями. Кроме того, обратите внимание, что в этой главе поставщики моделей перечисляются в алфавитном порядке, а не по степени приоритетности.

8.1. Anthropic

Многие произведения Айзека Азимова, одного из величайших фантастов в истории, строятся вокруг «трех законов робототехники» и их интерпретации.

1. Робот не может причинить вред человеку или своим бездействием допустить, чтобы человеку был причинен вред.

2. Робот должен повиноваться командам, отдаваемым людьми, за исключением случаев, когда такие команды противоречат Первому закону.

3. Робот должен защищать свое существование до тех пор, пока такая защита не противоречит Первому или Второму законам.

Цель здесь — сформулировать краткий набор правил, гарантирующих, что роботы будут полезными и не причинят вреда. Послужили ли эти законы источником вдохновения или нет, но эта идея связана с языковыми моделями, созданными Anthropic — еще одним поставщиком больших языковых моделей.

Компания Anthropic, основанная в 2021 году (несколькими бывшими сотрудниками OpenAI), неоднократно называла идею «конституционного ИИ» [1] одним из своих отличительных факторов по сравнению с другими моделями. Вкратце, она означает, что, обучая модели давать пользователям точные и безобидные ответы, мы опираемся на небольшой набор правил — условную «конституцию», чтобы оценивать качество ответов. Вместо тестировщиков-людей для разметки ответов, сгенерированных моделью во время обучения, мы используем второй ИИ, которому поручено оценивать ответы первого в соответствии с конституцией.

На момент написания книги самой новой моделью, выпущенной компанией Anthropic, была Claude 3.5 (отсылка к легендарному Клоду Шеннону). В этом разделе мы протестируем Claude, используя (как вы уже догадались) веб-интерфейс и библиотеку для Python.

8.1.1. Общение с Claude

Мы немного поболтаем с языковой моделью Claude (версии 3.5), чтобы получить представление о ее возможностях. Сначала перейдите на сайт Anthropic по адресу **www.anthropic.com** и нажмите кнопку **Try Claude** (Попробовать Claude).

Если у вас еще нет учетной записи Anthropic, то потребуется указать адрес электронной почты и номер телефона. После проверки ваших данных должен отобразиться интерфейс чата Claude (рис. 8.1).

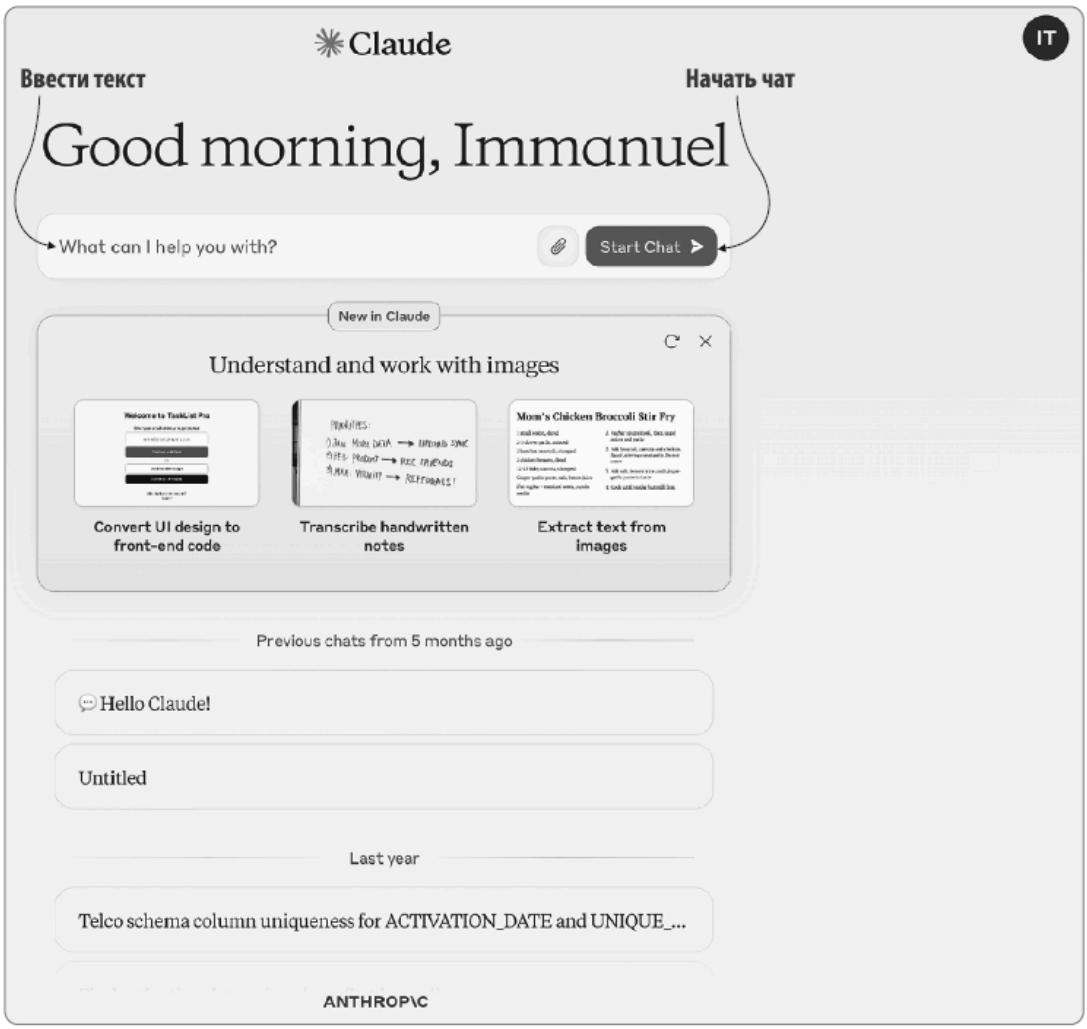


Рис. 8.1. Веб-интерфейс чат-бота Claude компании Anthropic

Интерфейс вполне понятен интуитивно: просто введите текст в соответствующее поле и нажмите кнопку справа, чтобы начать общение. Начните с дружеского приветствия, поговорите о погоде или попробуйте с помощью модели Claude решить несколько задач из главы 2 (например, классифицировать отзывы по тональности или перевести вопросы в SQL-запросы).

8.1.2. Библиотека для Python

После создания учетной записи в Anthropic можно создать ключи, перейдя по ссылке **https://console.anthropic.com/settings/keys**. Обязательно скопируйте свой ключ доступа после создания (так как к нему нельзя будет получить доступ позже)!

Получив ключ, перейдите в терминал и выполните следующую команду:

```
pip install anthropic==0.28
```

Она установит библиотеку Anthropic для Python. Если вы уже знакомы с библиотекой OpenAI для Python (а после прочтения предыдущих глав, полагаю, так и есть), то очень быстро привыкнете к библиотеке Anthropic.

Например, в листинге 8.1 показан код на языке Python для ответа на вопросы с помощью языковой модели Claude. Конечно, этот код не делает ничего такого, чего нельзя сделать через веб-интерфейс. Код приведен просто для того, чтобы показать, насколько легко использовать Claude через Python-интерфейс.

Листинг 8.1. Использование модели Claude от Anthropic для ответов на вопросы

```
import argparse
```

```
from anthropic import Anthropic

if __name__ == '__main__':

    parser = argparse.ArgumentParser()                    ❶
```

Определяет параметры

```
        parser.add_argument('ai_key',   type=str,
help='Ключ доступа к Anthropic')
        parser.add_argument('question',  type=str,
help='Вопрос для Claude')
    args = parser.parse_args()

    anthropic = Anthropic(api_key=args.ai_key)            ❷
```

Настраивает подключение к Anthropic

```
    completion = anthropic.messages.create(               ❸
```

Использует модель Claude для ответа

```
        model='claude-3-5-sonnet-20241022',
        max_tokens=100,
        messages=[
            {
                'role':'user',
                'content':args.question
            }])

    print(completion.content)                             ❹
```

Выводит ответ модели

В качестве параметров ❶ мы используем ключ доступа Anthropic и вопрос, на который хотим получить ответ. Как и с библиотеками OpenAI, доступ настраивается с помощью ключа ❷. После этого можно создавать промпты, на которые модель Claude будет генерировать ответы ❸.

Используя эндпоинт Anthropic — `anthropic.messages`, мы указываем идентификатор модели, которую следует использовать (`claude-3-5-sonnet-20241022` — на момент написания книги эта модель была самой новой), и максимальное количество токенов для генерации ответа (используя параметр `max_tokens`). Как и чат-модели OpenAI, Claude предназначена для общения с пользователем. Поэтому входные данные для модели Claude представляют собой список сообщений (в этом конкретном сценарии в списке только один элемент). Результат выполнения промпта моделью Claude содержится в поле `content` (его мы и выводим ❹).

Код из листинга 8.1 доступен по ссылке **Anthropic**, размещенной на сайте книги в разделе дополнительных материалов к этой главе. Для запуска кода откройте терминал и перейдите в папку, где он хранится. Затем выполните следующую команду:

```
python anthropic_claude.py ... "Что такое конституционный AI?"
```

Вместо многоточия укажите свой ключ доступа Anthropic. При выполнении программы должен отобразиться ответ на ваш вопрос, сгенерированный моделью Anthropic.

8.2. Cohere

В главе 2 мы коротко обсудили галлюцинации. По сути, галлюцинация возникает, когда языковая модель выдумывают что-то из-за отсутствия доступа к данным, которые актуальны для поставленной задачи. Канадский стартап Cohere уделяет особое внимание предотвращению подобных галлюцинаций с помощью метода *контекстуального обоснования* (grounding). Обоснование ответа языковой модели означает его привязку к реальным данным, что снижает шансы на «творческий вывод», не основанный на реальной информации.

Cohere предлагает широкий выбор коннекторов для доступа своих моделей к внешним источникам данных. Например, поддерживается веб-поиск, а также доступ к различным базам данных. На внутреннем уровне Cohere обращается к этим источникам данных и предоставляет языковой модели информацию, адаптированную под конкретный запрос. И что самое удобное, Cohere показывает все источники данных, использованные для генерации ответа. Если вы подозрительны (а к ответам языковых моделей вообще стоит относиться с недоверием), то можете проверить ссылки и убедиться, что они подтверждают сгенерированный ответ.

Посмотрим, как все это работает на практике. Пора пообщаться с моделью Command R+ от Cohere!

8.2.1. Общение с Command R+

На момент написания книги Command R+ была одной из новейших моделей Cohere. Как обычно, ее работу можно протестировать через веб-интерфейс, а для обработки больших объемов данных использовать Python. Сначала протестируем веб-интерфейс. Для этого перейдите на сайт **https://cohere.com/** и нажмите кнопку **Try Now** (Попробовать). После регистрации учетной записи перейдите на вкладку **Playground** (Песочница). Должен отобразиться веб-интерфейс (рис. 8.2).

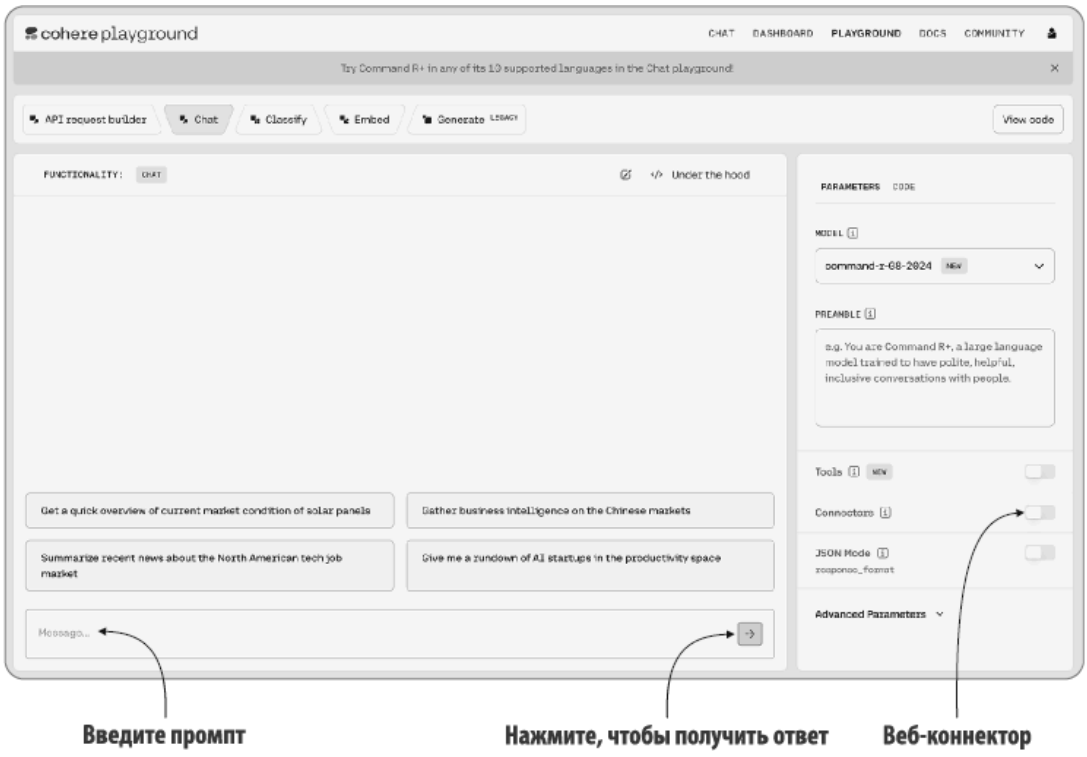


Рис. 8.2. Веб-интерфейс для общения с языковой моделью Cohere Command R+

На рис. 8.2 показаны поле для ввода промпта и кнопка для генерации ответа. А что за окно справа? Здесь указываются коннекторы, которые будут использоваться при генерации ответов. Коннектор позволяет Cohere получать доступ к внешним источникам данных. Веб-коннектор можно активировать (или деактивировать) с помощью соответствующего переключателя. Этот коннектор позволяет модели Cohere выполнять запросы в Интернете, как делаем все мы, когда пытаемся ответить на сложный вопрос, требующий фактических знаний. Попробуйте и посмотрите, как меняются ответы на фактологические вопросы при активации или деактивации веб-коннектора.

Что такое RAG

Полагаю, вы заметили, что на сайте Cohere активно рекламируют RAG, но что это такое? RAG расшифровывается

как Retrieval Augmented Generation — генерация ответа, дополненная результатами поиска. Это означает, что при генерации ответа входные данные для языковой модели дополняются информацией, извлеченной из внешнего источника.

8.2.2. Библиотека для Python

Cohere предлагает библиотеку для Python, аналогичную описанным в предыдущих разделах. Введите в терминале следующую команду для установки необходимой библиотеки:

```
pip install cohere==4.43
```

В листинге 8.2 содержится код простого интерфейса вопросов и ответов (файл с кодом доступен по ссылке **Cohere**, размещенной на сайте книги в разделе дополнительных материалов к этой главе).

Листинг 8.2. Использование библиотеки для ответов на вопросы

```
import argparse
import cohere

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
           parser.add_argument('ai_key',   type=str,
help='Ключ доступа к Cohere')
         parser.add_argument('question',  type=str,
help='Ответьте на этот вопрос')
    args = parser.parse_args()
```

```
        client = cohere.Client(args.ai_key)     ❶
```

Настраивает доступ

❷ Генерирует ответ с помощью Cohere

```
        prompt = f'Ответьте на этот вопрос: {args.question}'
    result = client.chat(prompt, connectors=[{'id': 'web-search'}])
```

❸ Выводит ответ и цитаты

```
    print(f'Ответ: result.text')
                print(f'Поисковые запросы: result.search_results')
              print(f'Результаты веб-поиска: result.documents')
```

Пользователи вводят в командной строке свой ключ доступа и вопрос. Для получения ключа перейдите по ссылке **https://dashboard.cohere.com/api-keys**. После настройки библиотеки Cohere с ключом доступа ❶ генерируется ответ с помощью библиотеки Cohere ❷ . Обратите внимание на обращение к коннекторам (`connectors`) в вызове функции `chat`. Здесь указывается список коннекторов, которые позволяют модели Cohere получать доступ к данным из внешних источников. Коннекторы указываются в виде списка (то есть можно разрешить доступ не к одному, а к нескольким). В данном случае используется коннектор с идентификатором `web-search` (этот же коннектор для веб-поиска применялся через веб-интерфейс в предыдущем подразделе).

Наконец ❸, программа выводит ответ модели и список веб-источников, к которым она обращалась для генерации ответа

(вместе с запросами, выполненными для получения указанных документов). Это позволяет нам проверить, действительно ли сгенерированный ответ подтверждается исходными материалами.

Теперь опробуем этот код. Перейдите в папку с ним и выполните в терминале следующую команду (вместо многоточия укажите свой ключ доступа Cohere):

```
python cohereqa.py ... "Где родился Стив Джобс?"
```

Вы получите примерно такой ответ (он немного сокращен):

```
❶ Сгенерированный ответ

Ответ: Стивен Пол Джобс родился в городе Сан-Франциско, штат Калифорния, в Соединенных Штатах.
Данное ему при рождении имя было изменено на Стив Джобс после того, как его усыновили Пол и Клара Джобс.


Джобс родился у Абдулфаттаха Джандали и Джоан Шибл 24 февраля 1955 года.
После того как его отдали на усыновление, Джобса забрали Пол и Клара Джобс, супружеская пара со скромным достатком.


Поисковые запросы: [      ❷ Запросы для веб-поиска
    {'search_query':
        {'text': 'Где родился Стив Джобс,
        'generation_id': '...'},
    'document_ids': [
          'web-search_1:0', 'web-search_3:1', 'web-search_4:0',
```

```
        'web-search_5:0', 'web-search_9:1'],
    'connector': {'id': 'web-search'}
    }]

Результаты веб-поиска: [          ❸ Использованные веб-источники
     {'id': 'web-search_4:0', 'snippet': 'Краткая
биография Стива Джобса
    История Стива Джобса от рождения до смерти – и
не только. Стивен
    Пол Джобс родился 24 февраля 1955 года в Сан-
Франциско, штат Калифорния.
    ... ', 'title': 'Краткая биография | all about
Steve Jobs.com',
                                                'url':
'https://allaboutstevejobs.com/bio/short_bio'},
    ...]
```

Рассмотрим вывод подробнее. Начальная его часть ❶ представляет собой ответ, сгенерированный моделью. Он выглядит правдоподобно, но можно ли ему доверять? Здесь на помощь приходят оставшиеся части вывода. Средняя ❷ предоставляет информацию о веб-запросах, использованных Cohere для формирования сгенерированного ответа. Они выбираются автоматически на основе входного вопроса. В данном случае веб-запросы точно соответствуют входному вопросу (для более длинного входного текста это не обязательно). В конце вывода ❸ представлены текстовые фрагменты и URL документов, полученных с помощью предыдущих запросов. В данном случае текстовые фрагменты, взятые из веб-документов (например, «Стивен Пол Джобс родился 24 февраля 1955 года в городе Сан-Франциско, штат Калифорния»), подтверждают ответ модели.

8.3. Google

Корпорацию Google представлять не нужно. Она активно занимается разработкой языковых моделей с самого начала их развития. Фактически архитектура «трансформер» [2], используемая практически всеми языковыми моделями, была изобретена (в основном) исследователями Google. Неудивительно, что корпорация разрабатывает собственные модели. На момент написания книги одной из последних моделей Google была Gemini, и мы протестируем ее в этом разделе.

8.3.1. Общение с Gemini

Чтобы попробовать поработать с моделью Gemini, перейдите по адресу **https://gemini.google.com/**. После регистрации учетной записи появится интерфейс (рис. 8.3).

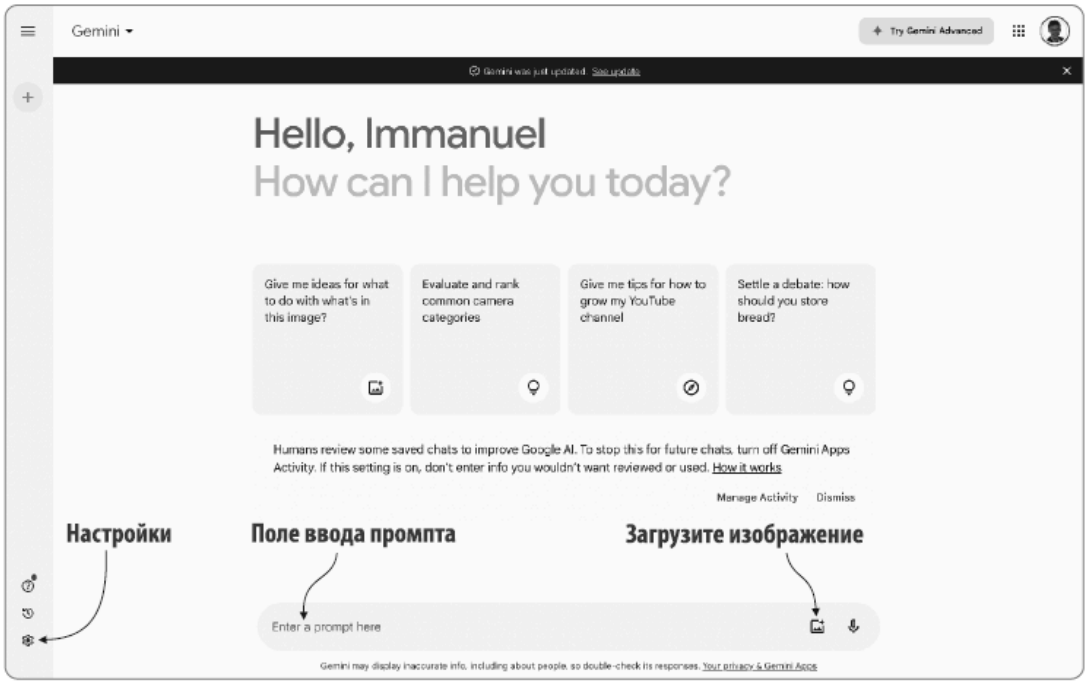


Рис. 8.3. Веб-интерфейс для общения с моделью Gemini от Google. Нажмите Settings (Настройки), чтобы активизировать дополнительные функции

Просто введите текст в соответствующее поле ввода (на рис. 8.3 обозначено как «Поле ввода промпта») и нажмите клавишу **Enter** для генерации ответа. Модель Gemini не ограничивается текстовым вводом. Нажмите кнопку справа, чтобы загрузить изображения. В чатах с Gemini можно ссылаться на эти изображения и задавать вопросы о них.

Особенностью Gemini, которая отличает ее от всех других моделей, которые мы обсуждали до сих пор, является ее интеграция с другими инструментами Google. Нажмите кнопку **Settings** (Настройки) (отмечена на рис. 8.3), а затем выберите вариант **Extensions** (Подключенные приложения). Например, можно предоставить Gemini доступ к вашим электронным письмам, нажав соответствующую кнопку. Когда-нибудь сталкивались с проблемой поиска информации в письмах годичной давности в ящике электронной почты? Языковая модель Google Gemini может помочь с этим.

8.3.2. Библиотека для Python

Как и другие поставщики языковых моделей, корпорация Google предлагает для доступа к модели Python-библиотеку. Ее можно установить, введя в терминале следующую команду:

```
pip install google-generativeai==0.7
```

Перейдите по адресу **https://aistudio.google.com/app/apikey**, чтобы получить ключ доступа к API Google. Следуйте инструкциям и скопируйте ключ после его создания. В листинге 8.3 показано, как использовать модель Gemini в Python для ответа на вопросы.

Листинг 8.3. Ответы на вопросы с использованием модели Gemini от Google

```
import argparse
```

```
import google.generativeai as genai

if __name__ == '__main__':
```

❶ Определяет параметры ввода

```
    parser = argparse.ArgumentParser()
          parser.add_argument('api_key',  type=str,
help='Ключ Google API')
         parser.add_argument('question',  type=str,
help='Вопрос, на который нужно ответить')
    args = parser.parse_args()
```

❷ Настраивает API с помощью ключа доступа

```
    genai.configure(api_key=args.api_key)
```

❸ Генерирует ответ, используя модель Gemini

```
        model  =  genai.GenerativeModel('gemini-1.5-
flash')
    reply = model.generate_content(args.question)
```

❹ Выводит ответ

```
    print(reply.text)
```

Действия аналогичны тем, которые мы выполняли при работе с предыдущими библиотеками. Входные параметры ❶ содержат ключ доступа (как вариант, его можно сохранить в переменной окружения), а также вопрос, на который модель должна дать ответ. Далее мы настраиваем библиотеку Google с помощью ключа доступа ❷ . Теперь можно создать модель и использовать ее для ответа на вопросы, применив метод

`generate_content` ❸ . Наконец, программа выводит сгенерированный ответ ❹.

Код из листинга 8.3 доступен по ссылке **Google**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе. В терминале перейдите в каталог, содержащий код. Например, чтобы протестировать модель Gemini, выполните следующую команду (замените многоточие своим ключом доступа Google):

```
python google.py ... "В чем смысл жизни?"
```

Полный обзор библиотеки Google выходит за рамки этой книги. Однако, имея опыт работы с библиотеками других поставщиков больших языковых моделей, вы сможете быстро освоить и этот API.

8.4. Hugging Face

Поставщики наподобие OpenAI инвестируют миллионы долларов в обучение таких моделей, как GPT-4o. Результатом дорогостоящего обучения являются значения параметров модели, которые обеспечивают ее более высокую производительность. Вложив немалые средства, вы, естественно, не захотите бесплатно делиться результатами обучения, не так ли? Следовательно, модели вроде GPT-4o обычно являются закрытыми, то есть компания OpenAI не сообщает значения параметров, полученные в результате обучения (обратите внимание, что к другим моделям, например Whisper, компания OpenAI дает общий доступ). Вместо этого OpenAI обрабатывает ваши промпты, используя собственную инфраструктуру, и берет плату за обработку (так мы в конечном счете оплачиваем дорогостоящее обучение моделей).

Однако языковые модели все чаще сталкиваются с конкуренцией со стороны чрезвычайно активного сообщества разработчиков открытого исходного кода. Университеты, стартапы и энтузиасты обучают собственные модели и часто публикуют их (и их веса) бесплатно для общего использования. Благодаря этому пользователи могут запускать такие модели самостоятельно локально, на собственной инфраструктуре. Для небольших моделей часто достаточно ноутбука с графическим процессором (graphics processing unit, GPU). Для более крупных моделей может потребоваться кластер GPU (или же вам придется обратиться к поставщикам облачных услуг, которые будут запускать модели с открытым исходным кодом за вас). Помимо потенциальных финансовых преимуществ (запуск моделей на собственной инфраструктуре может быть дешевле), есть и другие причины, почему локальный запуск моделей может оказаться единственным жизнеспособным вариантом. Например, возможно, вы не хотите доверять внешним поставщикам особо конфиденциальные данные. В этом случае локальный запуск — единственный вариант.

Как правило, модели с открытым исходным кодом значительно меньше моделей, предлагаемых поставщиками облачных услуг. Это логично, ведь, в конце концов, у кого найдется несколько миллионов долларов на обучение модели? Однако благодаря огромному количеству доступных моделей часто можно найти модель с открытым исходным кодом, специализирующуюся именно на решении интересующей вас задачи. Например, платформа Hugging Face Transformers на момент написания книги предлагала более 1 миллиона моделей с архитектурой «трансформер»! Какую бы задачу вы ни решали, велика вероятность, что найдется подходящая для нее модель. В этом разделе мы рассмотрим платформу Hugging Face и узнаем, как использовать ее модели локально.

8.4.1. Веб-платформа

Перейдите по адресу **https://huggingface.co/**. Платформа Hugging Face Transformers предлагает различные ресурсы, связанные с моделями-трансформерами. Сюда входят не только модели, но и наборы данных, которые можно использовать для обучения собственных моделей, а также облачные предложения, позволяющие запускать модели с открытым исходным кодом на облачной инфраструктуре Hugging Face.

В данный момент нас интересуют модели. Нажмите кнопку **Models** (Модели), чтобы увидеть список моделей (рис. 8.4).

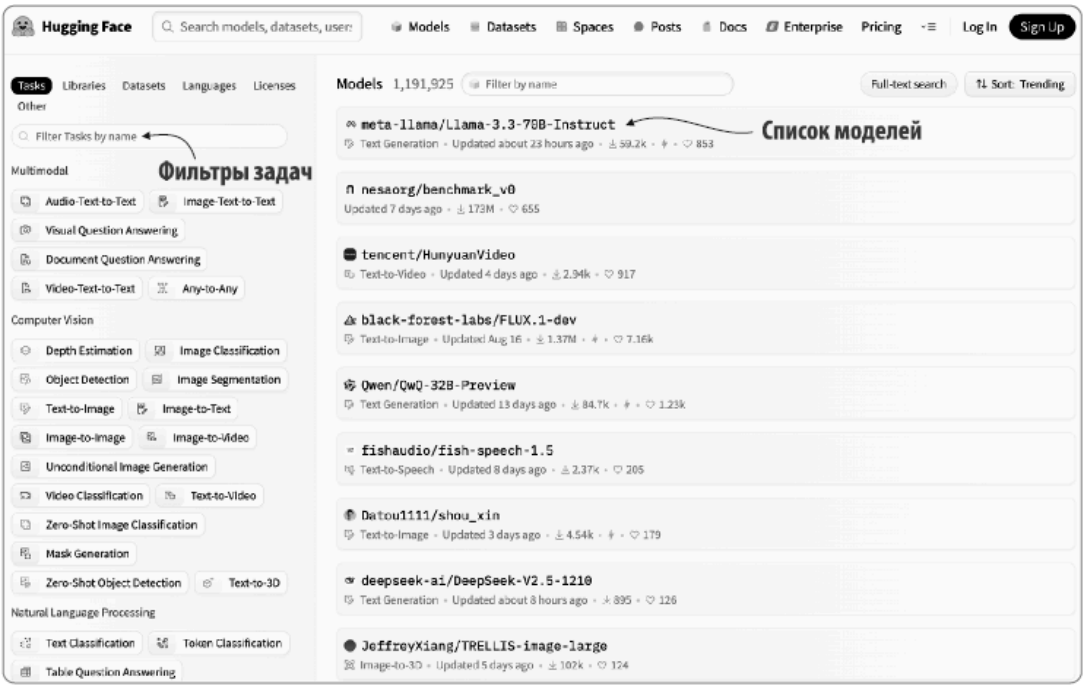


Рис. 8.4. Обзор моделей-трансформеров платформы Hugging Face. Нажмите кнопку фильтров Tasks (Задачи), чтобы сузить выбор. Нажимая на модели в списке Models (Модели), можно увидеть подробные сведения о них

Мы видим список из более чем миллиона моделей-трансформеров (и, поскольку количество моделей растет с каждым днем, полагаю, он еще увеличится). Их слишком много. Сузим выбор. В левой части экрана представлены различные параметры фильтрации, позволяющие сократить

список до моделей, которые вас действительно интересуют. Например, можно отфильтровать по типу задачи, которую должна выполнять модель. Сюда входят классификация текста (например, классификация отзывов по тональности), вопросы и ответы к изображениям (к примеру, «На картинке изображено яблоко?») и преобразование речи в текст (скажем, транскрибация голосовых запросов в текст). Почти для всех задач, которые описаны в этой книге, можно найти специализированную модель. Щелкните на названии любой стандартной задачи, и отобразятся только те модели, которые ее решают.

Щелкнув на названии одной из оставшихся моделей, вы увидите ее подробное описание. Например, на рис. 8.5 оно показано для модели BLIP от компании Salesforce (**https://huggingface.co/Salesforce/blip-image-captioning-large**) — она обрабатывает изображения и генерирует подходящие подписи. Слева находится подробное описание модели, а также ссылки на соответствующие статьи и примеры кода, показывающие, как использовать модель на разных аппаратных платформах (то есть локально). Справа находится интерфейс, позволяющий опробовать модель на нескольких примерах изображений.

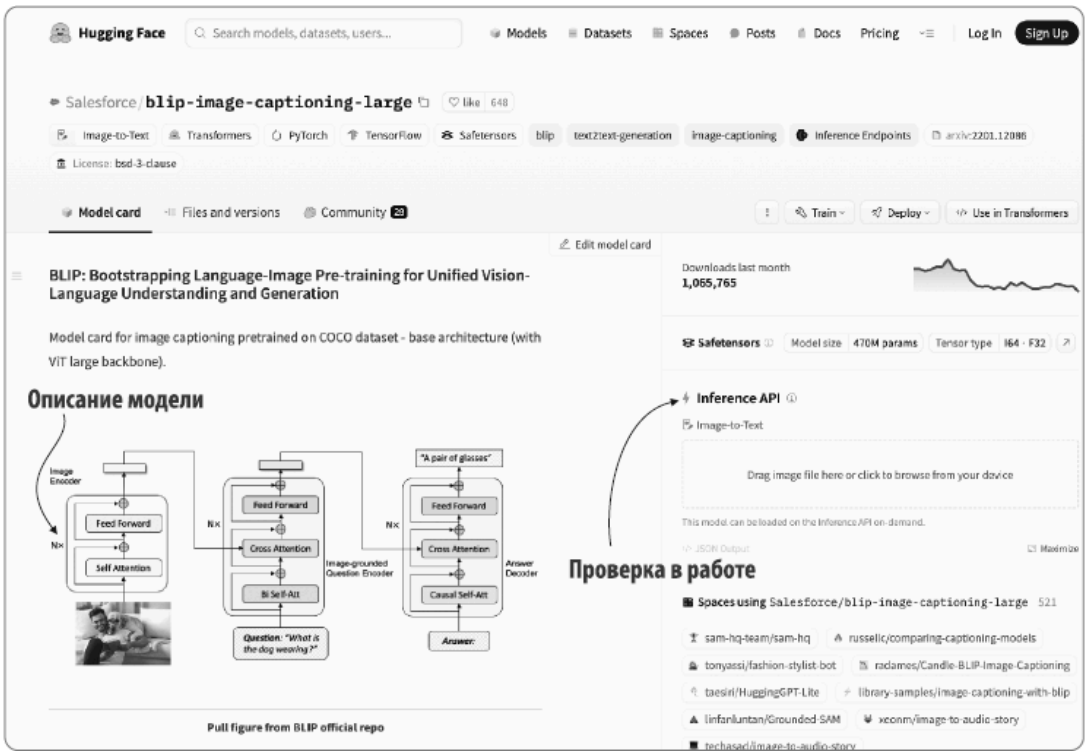


Рис. 8.5. Подробное описание модели BLIP от компании Salesforce. Можно прочитать ее описание слева или попробовать поработать с моделью, используя интерфейс справа

8.4.2. Библиотека для Python

Теперь запустим модели-трансформеры на нашей локальной инфраструктуре. Производительность, конечно, будет зависеть от характеристик используемого оборудования. Однако даже если оно имеет умеренную вычислительную мощность, вы сможете работать с моделями, которые мы собираемся опробовать. Но сначала необходимо установить библиотеку Hugging Face Transformers. Введите в терминале следующую команду:

```
pip install transformers==4.36
```

Основу библиотеки Transformers составляет PyTorch — популярная платформа для машинного обучения. Если вы еще не установили PyTorch, то введите в терминале такую команду

(в противном случае при попытке запустить код вы получите сообщение об ошибке):

```
pip install torch==2.1.2
```

Вот и вся настройка! Теперь мы можем использовать библиотеку Hugging Face Transformers, которую импортируем с помощью команды `import transformers`. Библиотека Transformers предлагает множество функций и различные способы использования моделей из ее репозитория (или, если на то пошло, варианты обучения ваших моделей). В текущем разделе мы рассмотрим лишь небольшую их часть, но этого достаточно, чтобы составить первое впечатление.

Для примера возьмем модель Roberta, которая классифицирует тональность сообщений. Она создана в Университете Кардиффа (доступна по адресу **https://mng.bz/rKoX**). По сравнению с GPT-4o и большинством других моделей, обсуждаемых в этом разделе, Roberta совсем небольшая. Однако она специализируется на анализе текста и определении его тональности. Roberta гораздо менее универсальна, чем GPT-4o и другие подобные модели, однако выполняет одну задачу и делает это довольно хорошо. Если вам, например, нужно классифицировать отзывы, то эта модель, скорее всего, подойдет.

Самый простой способ использования модели через библиотеку Transformers — это конвейер. С помощью команды, указанной ниже, создается конвейер для классификации по тональности на основе модели Roberta:

```
sentiment_pipeline = transformers.pipeline(
            model='cardiffnlp/twitter-roberta-base-
sentiment-latest')
```

Как видите, мы указываем модель, используя последнюю часть ее URL: имя учетной записи, предоставляющей модель (`cardiffnlp`), за которым следует идентификатор самой модели. При первом использовании кода библиотека Transformers автоматически загрузит модель из своего общедоступного репозитория. Обратите внимание: код работает отчасти потому, что модель, на которую мы ссылаемся, связана с определенным классом задач. Для других моделей в качестве отдельного входного параметра, возможно, придется указать типы задач, которые требуется решить.

Мы создали конвейер! Теперь с его помощью можно выполнить классификацию текста — например, вот так (предположим, что переменная `text_to_classify` содержит, как вы догадались, текст для классификации):

```
result = sentiment_pipeline(text_to_classify)
```

У нас есть все необходимое для создания простого приложения, которое будет классифицировать отзывы. Соответствующий код представлен в листинге 8.4 (доступен по ссылке **Hugging Face**, размещенной на сайте книги, в разделе дополнительных материалов к главе).

Листинг 8.4. Классификация сообщений по тональности с помощью платформы Hugging Face Transformers

```
import argparse
import transformers

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
           parser.add_argument('review',  type=str,
help='Текст отзыва')
```

```
    args = parser.parse_args()

     sentiment_pipeline = transformers.pipeline( ❶
```

Создает конвейер

```
            model='cardiffnlp/twitter-roberta-base-
sentiment-latest')

     result = sentiment_pipeline(args.review)      ❷
```

Применяет конвейер к входным данным

```
    print(result)                                  ❸
```

Выводит результат классификации

Можно заметить отличие от кода, с которым мы работали ранее: не требуется указывать ключ доступа! Модели Hugging Face находятся в открытом доступе, и мы запускаем их на собственной инфраструктуре, поэтому предоставлять учетные данные не нужно. Вместо этого единственным входным параметром является текст отзыва, который необходимо классифицировать.

Код объединяет фрагменты, о которых мы говорили ранее. Он создает конвейер ❶ , использует его для классификации входного текста ❷ и, наконец, выводит результат ❸ . Можно протестировать его, перейдя в соответствующую папку в терминале и введя, например, такую команду, содержащую отзыв о фильме:

```
python huggingface.py "Фильм действительно был
ужасен!"
```

При первом запуске кода, возможно, придется подождать несколько минут, пока библиотека Transformers скачает модель, на которую вы ссылаетесь. Но не беспокойтесь: библиотека кэширует скачанную модель, и вам не придется ждать при повторном запуске кода. После завершения обработки должен быть выведен примерно такой результат:

```
[{'label': 'negative', 'score': 0.9412825107574463}]
```

Он определенно верен для этого примера: отзыв краткий и на 100 % негативный. Протестируйте модель еще на нескольких отзывах и сравните результат с результатами вывода моделей типа GPT-4o. В большинстве случаев результаты классификации должны быть схожими. Конечно, GPT-4o — гораздо более универсальная модель, ее можно использовать для решения множества других задач. Но пока вас интересует классификация отзывов, модель от Hugging Face предлагает интересный компромисс между качеством и стоимостью.

Резюме

- Помимо компании OpenAI, есть и другие поставщики, предлагающие большие языковые модели. В основном они предлагают закрытые модели через облачный API. Модели различаются по своей универсальности, качеству вывода и цене.

- Для доступа к языковым моделям большинство поставщиков предлагают библиотеку для Python.

- Платформа Hugging Face Transformers предлагает различные модели, которые можно скачать бесплатно.

Литература

1. *Bai Y., Kadavath S., Kundu S. et al.* Constitutional AI: Harmlessness from AI Feedback // CoRR abs, 2022. — № 2212.0. — Pp. 1–32.

2. *Vaswani A., Shazeer N., Parmar N. et al.* Attention is All You Need // Advances in Neural Information Processing Systems, 2017. — Pp. 5999–6009.

9. Оптимизация стоимости и качества

В этой главе

- Выбор и настройка модели.
- Промпт-инжиниринг.
- Дообучение (fine-tuning) моделей.

Анализ данных с помощью больших языковых моделей — отличный способ быстро потратить деньги. Если вы уже пользовались моделью GPT-4 (или аналогичной ей), то, вероятно, замечали, с какой скоростью растут траты, заставляя вас регулярно пополнять счет. Но всегда ли требуется самая большая (и дорогая) модель? Неужели нельзя заставить более мелкие модели работать почти так же эффективно? Как получить максимальную отдачу от вложенных средств?

В этой главе речь пойдет о том, как сэкономить деньги при использовании языковых моделей на больших наборах данных. К счастью, есть несколько вариантов.

- Во-первых, больших языковых моделей множество. Выбор модели, которая является настолько маленькой (или, скорее, настолько дешевой), насколько это возможно, и при этом все еще хорошо справляется с аналитической задачей, может иметь большое значение для балансировки бюджета.
- Во-вторых, у моделей обычно есть различные параметры настройки, позволяющие регулировать все: от общей стратегии генерации текста до способа (де)приоритизации

конкретных токенов. Эти настройки нужно оптимизировать, чтобы превратить небольшие модели в альтернативы GPT-4 для определенных задач.

- В-третьих, можно использовать промпт-инжиниринг и изменить формулировку вопросов. Иногда это кардинально меняет результаты вывода модели!

- И наконец, если ни один из этих методов не подойдет, можно создать собственные узкоспециализированные модели, обученные исключительно для решения конкретной задачи. Разумеется, учитывая, что тратить миллионы на предварительное обучение не хочется, мы не станем начинать обучение новых моделей с нуля. Вместо этого мы, скорее всего, дообучим существующие модели на нескольких сотнях примеров. Этого часто бывает достаточно для значительного повышения производительности по сравнению с базовой моделью.

Безусловно, то, какой метод окажется наиболее эффективным, зависит от задачи, которую необходимо решить, а также от свойств данных. К счастью, при работе с крупными массивами данных логично потратить небольшую сумму на тестирование разных настроек модели на контрольной выборке. Велика вероятность, что эти первоначальные вложения окупятся по завершении анализа всего набора данных!

В этой главе мы применим вышеперечисленные варианты настройки в рамках примера.

9.1. Пример использования

Вы снова в компании Banana и вновь пытаетесь классифицировать отзывы пользователей. На сайте компании пользователи могут оставлять текстовые отзывы в свободном формате, описывающие их опыт использования товаров Banana. Вам нужно выяснить, являются ли эти отзывы положительными (то есть пользователь доволен продуктом) или отрицательными (то есть их чтение отпугнет потенциальных клиентов!). Конечно, для этой задачи можно использовать языковые модели (как вы видели в главе 4). Например, модель GPT-4 (на момент написания книги она была самой большой OpenAI-моделью для обработки текста). Предоставьте GPT-4 отзыв вместе с инструкциями по его классификации (включая описание возможных меток классов, таких как «положительный» и «отрицательный») — и для большинства отзывов вывод должен быть корректным.

Однако анализ данных с помощью модели GPT-4 стоит примерно 6 центов за 1000 токенов. Сумма (6 центов) может показаться небольшой, но компания Banana получает тысячи отзывов о продуктах ежедневно! Допустим, средний отзыв содержит около 100 токенов (примерно 400 символов). Далее предположим, что компания Banana получает примерно 10 000 отзывов в день. Это означает, что вы собираете 100 × 10 000 токенов в день = около 1 миллиона токенов в день и 365 миллионов токенов в год. В какую сумму обойдется анализ годового объема комментариев? Примерно 365 000 000 × 0,06 / 1000 = 21 900 долларов.

Это может ударить по бюджету! Нельзя ли уменьшить сумму? Например, на момент написания книги стоимость работы с моделью GPT-3.5 Turbo составляла всего 0,0005 доллара за тысячу токенов (токены оцениваются по-разному в зависимости от того, считывает их модель или генерирует, но сейчас мы этот нюанс опустим, чтобы упростить расчеты).

Следовательно, анализ годового объема комментариев обойдется всего в 365 000 000 × 0,0005 / 1000 = 182,5 доллара. Намного лучше! Однако, возможно, придется приложить дополнительные усилия, чтобы обеспечить удовлетворительное качество вывода и гарантировать оптимальное использование модели.

> СОВЕТ Вместо GPT-3.5 Turbo можно использовать и другие модели, такие как GPT-4o mini (идентификатор модели — gpt-4o-mini), из приведенных ниже примеров.

Именно этим мы и займемся в новом примере. Начав с самой наивной реализации классификатора, мы постепенно будем его совершенствовать и опробуем все варианты, обсуждавшиеся во введении к этой главе!

9.2. Классификатор без настройки

Начнем с базовой версии классификатора. Цель — загрузить в него отзыв и определить, как его классифицировать: как положительный или отрицательный. Для этого используем следующий шаблон промпта:

```
[Отзыв]
Отзыв положительный или отрицательный?
Ответ ("pos"/"neg"):
```

В этом шаблоне промпта `[Отзыв]` — это заполнитель, который заменяется фактическим текстом отзыва. Например, после подстановки промпт может выглядеть так (первые две строки соответствуют сокращенной версии отзыва для классификации, видимо, нового фильма, который показывают на Banana TV и который не понравился пользователю):

```
В научно-фантастическом фильме я готов закрыть
глаза на многие огрехи, но это было уже почти
невыносимо.
Отзыв положительный или отрицательный?
Ответ ("pos"/"neg"):
```

В идеале, отправив этот промпт модели GPT, мы ожидаем получить в качестве ответа вариант `pos` или `neg` (в данном случае — `neg`). Полный код Python показан в листинге 9.1.

Листинг 9.1. Классификация отзывов на положительные или отрицательные, базовая версия

```
import argparse
import openai
import pandas as pd
import time

client = openai.OpenAI()

def create_prompt(text):                                    ❶
```

Генерирует промпты

```
    """ Создать промпт для классификации отзывов по
тональности.

    Аргументы:
        text: текст для классификации.

    Выводит:
        Промпт для классификации текста.
    """
    task = Отзыв положительный или отрицательный?'
    answer_format = 'Ответ ("pos"/"neg")'
```

```
    return f'{text}\n{task}\n{answer_format}:'


def call_llm(prompt):                                    ❷
```

Вызывает большую языковую модель

```
    """ Отправить запрос языковой модели и вывести
ответ.
    Аргументы:
        prompt: введите запрос к языковой модели.

    Выводит:
          Ответ языковой модели и общее количество
токенов.
    """
    for nr_retries in range(1, 4):
        try:
```

❸ Генерирует ответ

```
                                        response   =
client.chat.completions.create(
                model='gpt-3.5-turbo',
                messages=[
                                      {'role':'user',
'content':prompt}
                    ],
                temperature=0
            )
```

❹ Извлекает ответ и количество использованных токенов

```
                                         answer   =
response.choices[0].message.content
            nr_tokens = response.usage.total_tokens
            return answer, nr_tokens
```

```
        except Exception as e:
            print(f'Исключение: {e}')
            time.sleep(nr_retries * 2)

    raise Exception('Не удается выполнить запрос к
модели OpenAI!')

if __name__ == '__main__':

        parser  =  argparse.ArgumentParser()      ❺
```

Выполняет разбор аргументов

```
        parser.add_argument('file_path',  type=str,
help='Путь к входному файлу')
    args = parser.parse_args()

    df = pd.read_csv(args.file_path)

    nr_correct = 0
    nr_tokens = 0

      for  _,  row  in  df.iterrows():            ❻
```

Просматривает отзывы

```
          text  =  row['text']                    ❼
```

Классифицирует отзывы

```
        prompt = create_prompt(text)
        label, current_tokens = call_llm(prompt)
```

```
                ground_truth = row['sentiment']    ❽
```

Обновляет счетчики

```
        if label == ground_truth:
            nr_correct += 1
        nr_tokens += current_tokens

            print(f'Метка: {label}; Истинная метка:
{ground_truth}')

                    print(f'Количество правильных
меток      :\t{nr_correct}')
                print(f'Количество использованных
токенов:\t{nr_tokens}')
```

Этот код похож на код классификаторов, представленных в главе 4. Функция `create_prompt` ❶ создает экземпляр шаблона промпта для конкретного отзыва (хранящегося во входном параметре `text`). Результатом является промпт, который можно отправить языковой модели с помощью функции `call_llm` ❷ . Здесь мы вызываем языковую модель GPT-3.5 Turbo ❸ (экономя деньги). Кроме того, параметру `temperature` задано значение `0`, а значит, случайность при генерации вывода минимизирована. Следовательно, при многократном запуске кода должен выводиться один и тот же результат. Вдобавок можно заметить, что вызов функции `call_llm` в листинге 9.1 немного длиннее, чем в версиях, рассмотренных в предыдущих листингах. Дело в том, что мы извлекаем не только сгенерированный языковой моделью ответ, но и количество использованных токенов ❹ . Подсчет количества

токенов позволит определить затраты на вызов для выборки данных.

Предположим, что отзывы для классификации хранятся в файле формата `.csv`. Пользователь должен указать путь к этому файлу в качестве аргумента командной строки ❺. После чтения `.csv`-файла отзывы просматриваются ❻ в том порядке, в котором они расположены в файле с входными данными. Для каждого отзыва извлекается связанный с ним текст ❼ (предполагается, что он находится в столбце `text`), создается промпт для классификации и отправляется вызов языковой модели. Результат — текст ответа, сгенерированный языковой моделью (будем надеяться, что это одна из двух меток классов — `pos` или `neg`), а также количество использованных токенов.

Наша цель — опробовать разные способы составления запросов к языковой модели, а затем сравнить качество результатов и затраты. Чтобы оценить качество вывода, предположим, что входной `.csv`-файл содержит не только текст отзыва, но и метку эталонных данных. Иными словами, каждый отзыв уже соотнесен с корректной меткой класса, хранящейся в столбце `sentiment` (поскольку наши два класса описывают тональность отзыва). Получив ответ языковой модели, мы сравниваем его с эталонной меткой ❽ и обновляем количество верно классифицированных отзывов (переменная `nr_correct`). Одновременно мы суммируем общее количество использованных токенов (так как стоимость обработки пропорциональна ему) и сохраняем в счетчике `nr_tokens`. Когда все отзывы будут обработаны, код, показанный в листинге 9.1, выводит итоговое количество корректных классификаций и количество использованных токенов.

9.3. Настройка модели

Код из листинга 9.1 доступен по ссылке **Untuned Classifier**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе. Мы повторно используем отзывы о фильмах из главы 4 — найдите в дополнительных материалах к ней ссылку **Reviews.csv** и скачайте файл. Он содержит десять отзывов вместе с соответствующими истинными метками (ground-truth label). Предположим, что код из листинга 9.1 и файл с отзывами хранятся на диске в одной папке. Откройте терминал, перейдите в эту папку и выполните следующую команду:

```
python basic_classifier.py reviews.csv
```

Должен отобразиться вывод:

```
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: pos; Истинная метка: pos
Метка: pos; Истинная метка: neg                    ❶
```

Некорректная метка

```
Метка: pos; Истинная метка: neg
Метка: negative; Истинная метка: neg               ❷
```

Несуществующая метка

```
Метка: negative; Истинная метка: pos
Метка: neg; Истинная метка: neg
Количество правильных меток      :    6
Количество использованных токенов:    2228
```

Первые десять строк описывают результаты для каждого отзыва. У нас есть метка, сгенерированная языковой моделью,

а также истинная метка (взятая из входных данных). В конце указано количество верно классифицированных отзывов и количество использованных токенов.

Из десяти отзывов правильно были классифицированы шесть. Это больше 50 %, но результат все равно не впечатляет. Что пошло не так? Изучив вывод, можно кое-что понять. Бывают случаи ❶, когда языковая модель просто выбирает неверную метку класса. Подобное не является неожиданностью. Однако есть и ситуации ❷, когда модель выбирает метку, которой даже не существует! Правда, она не так уж далека от истины (`negative` вместо `neg`), и это, похоже, легко исправить.

Мы сосредоточимся на (вероятно) самом простом и очевидном улучшении — заставим языковую модель генерировать только одну из двух возможных меток классов. Как это сделать? С помощью параметра `logit_bias`. Он позволяет пользователям изменять вероятность выбора определенных токенов (мы кратко обсуждали его и другие параметры моделей GPT в главе 3). В данном конкретном случае требуется существенно увеличить вероятность токенов, связанных с двумя метками классов (`neg` и `pos`). Параметр `logit_bias` задается в виде словаря Python, который сопоставляет идентификаторы токенов со значением смещения. Положительное смещение означает, что необходимо увеличить вероятность генерации соответствующего токена моделью. Отрицательное смещение означает, что вероятность генерации связанного токена нужно уменьшить.

В данном случае необходимо повысить шансы, что модель GPT-3.5 выберет один из двух токенов, представляющих метки классов. Поэтому мы задаем большое значение смещения для этих двух идентификаторов токенов. Значения смещения лежат в диапазоне от –100 до +100. Мы возьмем максимум и

присвоим смещение +100 токенам, представляющим метки классов. Сначала нужно найти их идентификаторы. Языковые модели представляют текст в виде последовательности идентификаторов токенов. Чтобы изменить смещение токенов, требуется сослаться на идентификаторы интересующих нас токенов.

Токенизатор — это компонент, преобразующий текст в идентификаторы токенов. Токенизаторы для всех моделей GPT можно найти по адресу **https://platform.openai.com/tokenizer**. Мы используем модель GPT-3.5, поэтому выберите токенизатор с пометкой **GPT-4&GPT-3.5 (legacy)**. На рис. 9.1 показан его веб-интерфейс.

Можно ввести текст в поле и нажать кнопку **Token IDs** (Идентификаторы токенов), чтобы увидеть соответствующие идентификаторы. Используя токенизатор, мы узнаем, что идентификатор токена `pos` — `981`, а токена `neg` — `29875`. Теперь мы готовы добавить смещение к вызову модели следующим образом:

```
import openai
client = openai.OpenAI()

response = client.chat.completions.create(
    model='gpt-3.5-turbo',
    messages=[
        {'role':'user', 'content':prompt}
        ],
          logit_bias  =  {981:100,  29875:100},  ❶
```

❶ Определяет смещение

```
    temperature=0
    )
```

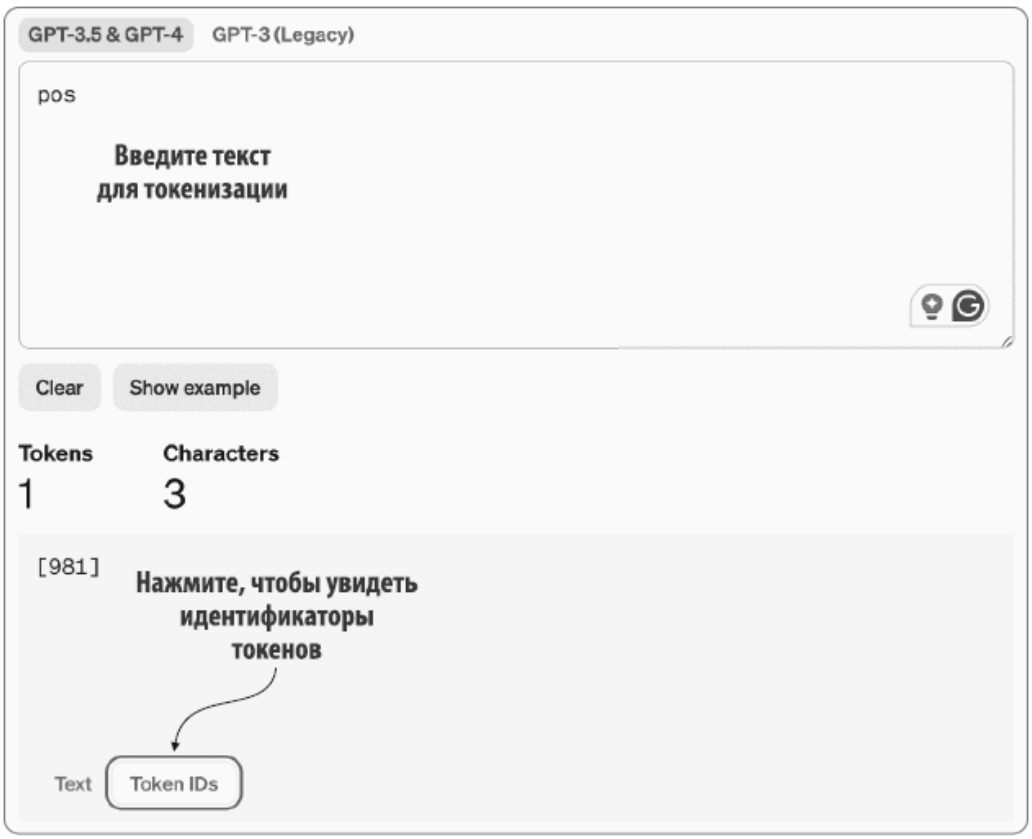


Рис. 9.1. Токенизатор GPT на сайте https://platform.openai.com/tokenizer: введите текст, чтобы узнать соответствующие идентификаторы токенов

По сравнению с предыдущим вызовом (см. листинг 9.1) мы добавляем смещение логитов ❶, сопоставляя идентификаторы двух интересующих нас токенов (`pos` с ID `981` и `neg` с ID `29875`) с максимально возможным значением смещения `100`. Это должно решить проблему генерации токенов, не соответствующих меткам классов, не так ли?

> ВНИМАНИЕ! Приведенный ниже код некорректен: он требует много времени на выполнение и серьезных финансовых затрат. Не запускайте его без исправления, представленного в конце этого раздела!

Итак, можно добавить смещение логитов к коду из листинга 9.1. Как вариант, далее в этой главе я представлю

настраиваемую версию классификатора, которая позволит пробовать разные комбинации параметров настройки (включая смещение логитов). Запустив классификатор с добавленным смещением, скорее всего, вы увидите вывод, похожий на приведенный ниже (на самом деле, поскольку выполнение кода занимает много времени и требует немалых затрат, думаю, вы просто поверите мне на слово):

❶ Несуществующие метки для каждого ввода:

```
Метка: negnegnegnegnegnegnegnegneg ...; Истинная
метка: neg
Метка: negposnegnegnegnegnegnegneg ...; Истинная
метка: neg
Метка: negposnegnegnegnegnegnegneg ...; Истинная
метка: neg
Метка: negposnegposnegnegnegnegneg ...; Истинная
метка: neg
Метка: posnegposnegposnegposnegposneg ...; Истинная
метка: pos
Метка: posnegpospospospospospospospos ...; Истинная
метка: neg
Метка: posnegpospospospospospospospos ...; Истинная
метка: neg
Метка: negposnegposnegposnegposnegpos ...; Истинная
метка: neg
Метка: negposnegposnegposnegposnegpos ...; Истинная
метка: pos
Метка: negposnegposnegnegnegnegneg ...; Истинная
метка: neg
Количество правильных меток      :   0
Количество использованных токенов:     2318      ❷
```

Возросло использование токенов

О нет — ни одной правильной классификации! Что произошло? Сравнение сгенерированных «меток» с эталонными выявляет проблему ❶: мы генерируем только два возможных токена (это отлично!), но слишком много раз (а это уже хуже!). В результате увеличивается потребление токенов ❷ (заметьте, что длина вывода при генерации примера была ограничена; в противном случае потребление токенов было бы намного выше), но главная проблема в другом: вывод не соответствует ни одной из меток класса.

Почему модель генерирует так много токенов

По сути, мы ограничили модель генерацией текста с использованием только двух токенов. Именно их мы хотим видеть в выводе. Однако мы забыли разрешить модели генерировать токен, указывающий на конец вывода! Вот почему модель не может остановить генерацию.

Существует несколько способов решения этой проблемы. Конечно, можно добавить этап последующей обработки для извлечения только первого токена из вывода языковой модели. Это (в основном) решило бы проблему с метками классов. Изучив вывод, вы увидите, что использование первого токена приводит к верному результату в семи из десяти случаев. Однако такой подход чреват другой проблемой: мы платим за генерацию токенов, которые в итоге не используем! Нам он точно не подходит. Поэтому продолжим настраивать модель, ограничив также длину вывода. Нам нужен всего один токен (этот способ работает только потому, что две возможные метки классов могут быть представлены одним токеном). С этим справляется параметр `max_tokens`. Используем его при вызове языковой модели:

```
response = client.chat.completions.create(
    model='gpt-3.5-turbo',
    messages=[
        {'role':'user', 'content':prompt}
        ],
    logit_bias = {981:100, 29875:100},   Определяет смещение
    temperature=0, max_tokens=1
)
```

При тестировании (которое должно пройти быстро и стоить дешево) вы должны увидеть следующий вывод:

```
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: pos; Истинная метка: pos
Метка: pos; Истинная метка: neg
Метка: pos; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: pos
Метка: neg; Истинная метка: neg
Количество правильных меток       :     7          ❶
```

Улучшает базовый классификатор

```
Количество использованных токенов:      2228       ❷
```

Сокращает использование токенов

Намного лучше! Количество корректно обработанных случаев выросло с шести (для версии без настройки) до семи ❶. Возможно, результат не слишком впечатляет. Но в масштабах всего набора данных он фактически означает, что мы повысили точность с 60 до 70 %: то есть теперь тысячи дополнительных отзывов будут классифицированы правильно! Конечно, есть нюанс. В реальности, вероятно, следует использовать гораздо более крупную выборку. Из-за случайных колебаний

показатели точности на тестовой выборке могут не отражать показатели точности на всем наборе данных. Чтобы не усложнять (и не тратить лишних денег при тестировании), мы ограничиваемся здесь десятью образцами. Дополнительный плюс — потребление токенов снова сократилось ❷ (на самом деле разрыв в потреблении токенов по сравнению с версией без ограничения размера вывода, вероятно, будет намного больше). Заметьте, что два рассмотренных здесь параметра — лишь малая часть доступных вариантов настройки. Более подробную информацию о соответствующих параметрах вы найдете в главе 3. Каждый раз, настраивая модель для выполнения новой задачи, обязательно учитывайте все параметры, которые могут оказаться потенциально важными. Затем поэкспериментируйте со значениями параметров на небольшой выборке данных и оцените, какой вариант дает наилучший результат.

9.4. Выбор модели

Предположим, что возможности повышения производительности путем настройки текущей модели исчерпаны. Что еще можно сделать? Конечно, можно выбрать другую модель. В предыдущей главе описывалось несколько альтернатив моделям GPT. Наличие модели, специально обученной для выполнения интересующей вас задачи (например, классификации текста), — повод обратить на нее внимание. К прочим факторам, способным повлиять на выбор модели, относятся конфиденциальность данных, которые планируется обрабатывать, и допустимость отправки этих данных конкретным поставщикам языковых моделей.

Получить информацию об относительной производительности разных моделей можно на сайте

https://crfm.stanford.edu/helm/lite/latest/[8]. Там содержатся результаты теста всесторонней оценки языковых моделей Стэнфордского университета — HELM. Тест сравнивает языковые модели по различным сценариям и предоставляет как результаты выполнения конкретных задач, так и данные о средней производительности, агрегированные из различных сценариев.

Полагаю, с ними стоит ознакомиться, чтобы понять, какие модели могут быть вам интересны. Однако, поскольку на производительность языковой модели влияет множество факторов, оценивать разные модели лучше на конкретной интересующей вас задаче.

Для простоты возьмем только модель GPT-4 в качестве альтернативы модели GPT-3.5 Turbo (использовавшейся до сих пор). При вызове языковой модели замените ее название:

```
response = client.chat.completions.create(
    model='gpt-4',
    messages=[
        {'role':'user', 'content':prompt}
        ],
    logit_bias = {981:100, 29875:100},  ◄── Определяет смещение
    temperature=0, max_tokens=1
)
```

При выполнении получившегося кода должен появиться такой вывод:

```
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: pos; Истинная метка: pos
Метка: pos; Истинная метка: neg
Метка: pos; Истинная метка: neg
Метка: neg; Истинная метка: neg
```

```
Метка: pos; Истинная метка: pos                    ❶
```

Верный результат классификации

```
Метка: neg; Истинная метка: neg
Количество правильных меток        :     8         ❷
```

Лучший результат на данный момент

```
Количество использованных токенов:       2228      ❸
```

То же самое количество токенов

По сравнению с предыдущей версией GPT-4 правильно решает на один тестовый пример больше ❶ ! Точность повышается до 80 % ❷ , в то время как потребление токенов остается неизменным ❸ . Это, кстати, не гарантируется при смене модели. Разные модели могут использовать разные токенизаторы, поэтому представление одного и того же текста может требовать разного количества токенов в зависимости от модели. В данном конкретном случае количество токенов не меняется, поскольку GPT-4 и GPT-3.5 используют один и тот же токенизатор.

Означает ли это, что сумма оплаты останется прежней? Не совсем. Поскольку модель GPT-4 взимает гораздо более высокую плату за токен, мы заплатим примерно в 120 раз больше, чем раньше (относительная разница между ценами за токен у моделей GPT-4 и GPT-3.5 Turbo). Именно поэтому мы постараемся добиться от модели GPT-3.5 максимально возможной производительности, чтобы не пришлось использовать GPT-4.

В некоторых случаях при выборе и настройке модели стоит самостоятельно просмотреть тестовые данные. Это позволит получить более полное представление о сильных сторонах и ограничениях различных моделей и оценить, являются ли

репрезентативными тестовые случаи, на которых модель работает плохо. Например, модель GPT-4 обрабатывает этот отзыв корректно, а GPT-3.5 — нет:

> Если вы хотели увидеть фильм со Стэном Лорелом из комедийного дуэта Лорела и Харди, то вы не угадали. Стэну потребуется еще два года, чтобы хотя бы приблизиться к характерной игре и ритму тех фильмов. Но если вы хотите посмотреть качественную пародию на "Кровь и песок" Рудольфа Валентино, снятый годом ранее, этот фильм для вас. Здесь использованы все возможные приемы, как в самой комедии, так и в титрах. И пусть фильм держится не на персонажах, зато в нем используется сюжет фильма Валентино — ну, почти.

В отзыве содержатся как положительные (к концу), так и отрицательные (в начале) аспекты. Хотя окончательный вердикт положительный, можно сделать вывод, что тратить дополнительные средства на тщательный анализ пограничных случаев, подобных этому отзыву, не стоит.

9.5. Промпт-инжиниринг

Помимо смены, что еще можно сделать для улучшения производительности модели? Мы пока не рассмотрели одну область — составление промпта, используемого для классификации. Изменение шаблона промпта может сильно повлиять на качество результата. Тот факт, что настройка промпта часто имеет решающее значение, даже привел к появлению специального термина *промпт-инжиниринг*, который описывает процесс поиска оптимальных шаблонов промптов. Более того, сложные задачи создания промптов стали причиной разработки множества платформ,

предлагающих шаблоны промптов для огромного количества различных задач. Если у вас закончились идеи вариантов промптов, то посетите **https://promptbase.com/**, **https://prompthero.com/** и другие подобные платформы. Бизнес-модель таких платформ заключается в предоставлении пользователям возможности покупать и продавать шаблоны промптов, оптимизирующие производительность определенных моделей для выполнения конкретных задач.

Чтобы понять, какой промпт наиболее эффективен, обычно приходится экспериментировать. Далее мы сосредоточимся на основах работы с промптами и рассмотрим классический метод повышения качества вывода за счет изменения промпта. Речь идет об обучении на небольшом количестве примеров: мы помогаем модели, предоставляя ей несколько образцов решения задачи. Концепция знакома нам из повседневной жизни: часто бывает трудно понять новую задачу или подход, имея лишь их описание. Разобраться в них гораздо проще, если есть несколько примеров. Так, в предыдущих разделах можно было бы просто обсудить семантику нескольких релевантных параметров настройки модели. Но разве не лучше было изучить их настройку на конкретном практическом примере?

Конечно, лучше. Языковые модели «чувствуют» то же самое, и добавление нескольких полезных примеров часто может повысить их производительность. Как же показать им примеры? Легко: указать их как часть промпта. Например, в сценарии классификации требуется, чтобы языковые модели классифицировали отзывы. Примером будет отзыв вместе с эталонной меткой класса.

Используем этот шаблон, чтобы интегрировать в промпт один пример:

```
[Пример отзыва]
Отзыв положительный или отрицательный?
```

```
Ответ ("pos"/"neg"):[Пример решения]
[Классификация отзывов]
Отзыв положительный или отрицательный?
Ответ ("pos"/"neg"):
```

Подставив вместо меток-заполнителей пример отзыва, эталонный ответ к нему и целевой отзыв для классификации, мы получим примерно такой промпт:

```
Не стану отрицать, что, покупая этот товар на eBay,
я ожидал многого...                          ❶ Пример отзыва
Отзыв положительный или отрицательный?       ❷ Инструкция
Ответ ("pos"/"neg"):neg                      ❸ Эталонный ответ
В научно-фантастическом фильме я готов закрыть глаза на многие огрехи,
но это было уже почти невыносимо...          ❹ Отзыв для классификации
Отзыв положительный или отрицательный?       ❺ Инструкция
Ответ ("pos"/"neg"):
```

Вы видите пример отзыва ❶, инструкцию ❷ и эталонный ответ для примера отзыва ❸. Затем идет отзыв, который нужно классифицировать ❹, и инструкция по классификации (снова) ❺, но теперь без ответа (ведь его-то мы и хотим получить от языковой модели). В этом промпте модели предоставляется только один пример правильно решенной задачи. Он может помочь ей лучше понять, что требуется сделать.

Безусловно, существует много вариантов добавления примеров в промпт. Мы выбрали, пожалуй, самое простое решение — использовать одну и ту же структуру промпта дважды для двух отзывов. Поскольку используемая структура не меняется, наш промпт избыточен: мы повторяем инструкцию к задаче (❷ и ❺), указывая две возможные метки классов. Хотя мы не будем этого делать здесь, было бы интересно поэкспериментировать и посмотреть, можно ли интегрировать примеры в промпт иначе, убрав избыточность и сократив его длину (тем самым уменьшив количество обрабатываемых токенов и в конечном счете стоимость обработки).

До сих пор мы рассматривали добавление только одного примера. Но порой его недостаточно. Поэтому целесообразно предоставить языковой модели несколько примеров. Предположим, они у нас есть: отзывы с относящимися к ним метками классов, которые хранятся в структуре данных под названием `samples`. Для генерации промптов, интегрирующих эти примеры, можно использовать такой код:

```
def  create_single_text_prompt(text,  label):      ❶
```

Создает промпт для одного отзыва

```
      """ Создать промпт для классификации одного
текста.

    Аргументы:
        text: текст для классификации.
          label: корректная метка класса (оставить
пустым, если недоступна).

    Выводит:
        Промпт для классификации текста.
    """
    task = 'Отзыв положительный или отрицательный?'
    answer_format = 'Ответ ("pos"/"neg")'
         return  f'{text}\n{task}\n{answer_format}:
{label}'


def create_prompt(text, samples):                  ❷
```

Генерирует промпт для всех отзывов

```
    """ Генерирует промпт для классификации отзывов
по тональности.
```

```
    Аргументы:
        text: классифицировать этот текст.
        samples: добавить эти образцы в промпт.

    Выводит:
        Ввод данных для LLM.
    """
    parts = []
    for _, row in samples.iterrows():                    ❸
```

Интегрирует примеры

```
        sample_text = row['text']
        sample_label = row['sentiment']
                                              prompt     =
create_single_text_prompt(sample_text,
sample_label)
        parts += [prompt]


    prompt = create_single_text_prompt(text, ")    ❹
```

Добавляет отзыв для классификации

```
    parts += [prompt]
    return '\n'.join(parts)
```

Функция `create_single_text_prompt` ❶ создает экземпляр следующего шаблона для одного отзыва:

```
[Отзыв]
Отзыв положительный или отрицательный?
Ответ ("pos"/"neg"):[Метка]
```

Мы используем одну и ту же функцию, чтобы указать как примеры отзывов, так и сам отзыв вместе с задачей

классификации, которую модель должна решить за нас. Если пример отзыва указан, то метка-заполнитель `[Метка]` заменяется эталонной меткой класса для соответствующего отзыва. Обозначая задачу, которую должна решить языковая модель, мы еще не знаем верную метку класса. В этом случае мы заменяем заполнитель `[Метка]` пустой строкой. Задача языковой модели — дополнить промпт фактической меткой класса.

Функция `create_prompt` ❷ генерирует полный промпт, содержащий все примеры отзывов, а также отзыв, который требуется классифицировать. Сначала ❸ она перебирает примеры отзывов. Мы предполагаем, что структура `samples` хранит текст отзыва в столбце `text`, а связанные метки классов — в столбце `sentiment`. Мы добавляем часть промпта ❹ для примера отзыва, используя функцию `create_single_text_prompt` (обсуждавшуюся ранее). Наконец, мы добавляем инструкции для классификации интересующего нас отзыва.

Теперь вернемся к работе с моделью GPT-3.5 Turbo. Однако на этот раз применим новую функцию генерации промптов. Пока ограничимся в промпте одним примером отзыва. По ссылке **Reviews Training** (см. сайт книги, раздел дополнительных материалов к этой главе) доступны обучающие отзывы с корректными метками классов (файл `train_reviews.csv`). Отзывы в этом файле не совпадают с отзывами в файле `reviews.csv` (с помощью которого мы тестируем свой подход). После того как в промпты будет добавлен в качестве примера только первый отзыв из файла `train_reviews.csv`, должен отобразиться следующий вывод:

```
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
```

```
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: pos; Истинная метка: pos
Метка: pos; Истинная метка: neg
Метка: pos; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: pos; Истинная метка: pos
Метка: neg; Истинная метка: neg
Количество правильных меток          :   8        ❶
```

Эквивалентно результату GPT-4

```
Количество использованных токенов:      4078    ❷
```

Использование токенов возросло примерно в два раза

Ура! Мы повысили точность до 80 % ❶ . Именно такой показатель был получен при использовании модели GPT-4 с исходными промптами (без примеров отзывов). В то же время использование токенов увеличилось ❷. А точнее, поскольку мы добавляем к каждому промпту второй отзыв (то есть у нас есть один пример отзыва и отзыв для классификации), то потребление токенов выросло почти вдвое по сравнению с последней версией. Но если сравнивать с использованием модели GPT-4 на более коротких промптах, то наш текущий подход все еще примерно в 60 раз дешевле (поскольку использование модели GPT-4 приблизительно в 120 раз дороже, чем GPT-3.5 Turbo).

9.6. Настраиваемый классификатор

Теперь, после изучения нескольких вариантов настройки, у вас может возникнуть соблазн попробовать новые комбинации. Например, нужно ли по-прежнему добавлять смещение (по

сути, ограничивая вывод двумя возможными метками классов), если добавлены примеры? Можно ли добиться большей точности, используя в промпте для более крупной модели несколько примеров? Менять код для проверки новых сочетаний быстро надоедает. Но не волнуйтесь, есть вариант и на такой случай. Он показан в листинг 9.2 (код доступен по ссылке **Tunable Classifier**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе).

Листинг 9.2. Настраиваемая версия классификатора тональности отзывов

```
import argparse
import openai
import pandas as pd
import time

client = openai.OpenAI()

def create_single_text_prompt(text, label):
      """ Создать промпт для классификации одного
текста

    Аргументы:
        text: текст для классификации.
          label: корректная метка класса (оставить
пустым, если недоступна).

    Выводит:
        Промпт для классификации текста.
    """
    task = 'Отзыв положительный или отрицательный?'
    answer_format = 'Ответ ("pos"/"neg")'
```

```
        return  f'{text}\n{task}\n{answer_format}:
{label}'

def  create_prompt(text,  samples):                    ❶
```

Генерирует промпты с примерами

```
    """ Генерирует промпт для классификации отзывов
по тональности.

    Аргументы:
        text: классифицировать этот текст.
        samples: добавить эти образцы в промпт.

    Выводит:
        Ввод данных для LLM.
    """
    parts = []
    for _, row in samples.iterrows():
        sample_text = row['text']
        sample_label = row['sentiment']
                                          prompt     =
create_single_text_prompt(sample_text,
sample_label)
        parts += [prompt]

    prompt = create_single_text_prompt(text, ")
    parts += [prompt]
    return '\n'.join(parts)
```

❷ Вызывает языковые модели, используя параметры

```
def call_llm(prompt, model, max_tokens, out_tokens):
    """ Отправить запрос большой языковой модели и вывести ответ.

    Аргументы:
        prompt: введите запрос к языковой модели.
            model: назовите, какую модель OpenAI выбрать.
          max_tokens: максимальная длина вывода в токенах.
            out_tokens: укажите приоритет ID этих токенов в выводе.

    Выводит:
            В ответе укажи языковую модель и общее количество токенов.
    """
    optional_parameters = {}
    if max_tokens:
                optional_parameters['max_tokens'] = max_tokens
    if out_tokens:
            logit_bias = {int(tid):100 for tid in out_tokens.split(',')}
                optional_parameters['logit_bias'] = logit_bias

    for nr_retries in range(1, 4):
        try:
                                        response = client.chat.completions.create(
```

```
                model=model,
                messages=[
                                        {'role':'user',
'content':prompt}
                    ],
                                **optional_parameters,
temperature=0
                )

                                            answer    =
response.choices[0].message.content
            nr_tokens = response.usage.total_tokens
            return answer, nr_tokens

        except Exception as e:
            print(f'Исключение: {e}')
            time.sleep(nr_retries * 2)
    raise Exception('Не удается выполнить запрос к
модели OpenAI!')

if __name__ == '__main__':

        parser  =  argparse.ArgumentParser()      ❸
```

Парсирует параметры командной строки

```
         parser.add_argument('file_path',  type=str,
help='Путь к входному файлу')
            parser.add_argument('model',   type=str,
help='Название модели OpenAI')
        parser.add_argument('max_tokens',  type=int,
help='Максимальный размер вывода')
```

```
        parser.add_argument('out_tokens',  type=str,
help='Токены, которым отдать приоритет')
        parser.add_argument('nr_samples',  type=int,
help='Количество примеров')
       parser.add_argument('sample_path',  type=str,
help='Путь к примерам')
    args = parser.parse_args()

    df = pd.read_csv(args.file_path)


      samples  =  pd.DataFrame()                    ❹
```

Читает примеры с диска

```
    if args.nr_samples:
        samples = pd.read_csv(args.sample_path)
        samples = samples[:args.nr_samples]

    nr_correct = 0
    nr_tokens = 0

    for _, row in df.iterrows():


          text  =  row['text']                      ❺
```

Классифицирует отзыв

```
        prompt = create_prompt(text, samples)
        label, current_tokens = call_llm(
            prompt, args.model,
            args.max_tokens,
            args.out_tokens)
```

```
            ground_truth = row['sentiment']    ❻
```

Обновляет счетчики

```
        if label == ground_truth:
            nr_correct += 1
        nr_tokens += current_tokens

            print(f'Метка: {label}; Истинная метка:
{ground_truth}')
```

❼ Выводит счетчики

```
                print(f'Количество правильных
меток:\t{nr_correct}')
            print(f'Количество использованных
токенов  :\t{nr_tokens}')
```

Эта реализация позволяет опробовать все варианты настройки, задав корректные параметры командной строки.

Генерация промптов ❶ работает так, как описано в предыдущем разделе. Функция `create_prompt` в качестве входных данных принимает текст отзыва для классификации и примеры отзывов. Примеры добавляются в промпт, потенциально помогая языковым моделям классифицировать интересующий нас отзыв. Обратите внимание, что можно по-прежнему посмотреть, как работает языковая модель без примеров (не указав их). Классификация без примеров — это частный случай.

Функция `call_llm` ❷ объединяет все упомянутые ранее параметры настройки.

- Во-первых, это имя вызываемой модели (параметр `model`).

- Во-вторых, можно указать максимальное количество токенов вывода (`max_tokens`).

- Наконец, можно указать смещение: токены, которым следует отдать приоритет при генерации вывода. Параметр `out_tokens` позволяет пользователям задавать список идентификаторов токенов, разделенных запятыми, которым мы присваиваем высокий приоритет (по сути, ограничивая вывод одним из этих токенов). Имя модели указывать обязательно, но, задав значение `0` для параметра `max_tokens` и пустую строку для параметра `out_tokens`, можно будет не менять заданные по умолчанию настройки OpenAI.

Настраиваемый классификатор использует довольно много параметров командной строки ❸ . Рассмотрим параметры в том порядке, в котором их нужно указывать:

- `file_path` — путь к файлу `.csv`, содержащему отзывы для оценки языковой моделью;

- `model` — имя языковой модели, которую следует использовать (например, `gpt-3.5-turbo`);

- `max_tokens` — максимальное количество выходных токенов для генерации в ответе на один отзыв;

- `out_tokens` — список токенов, разделенных запятыми, для приоритизации при генерации вывода;

- `nr_samples` — количество примеров отзывов с ответами, которое будет добавлено в каждый промпт;

- `sample_path` — путь к файлу в формате `.csv`, содержащему отзывы с корректными метками классов для использования в качестве примеров (может быть пустым, если для параметра `nr_samples` задано значение `0`).

> **ПРЕДУПРЕЖДЕНИЕ**! Почти всегда рекомендуется ограничивать количество токенов вывода. В частности, это необходимо делать всякий раз, если вы задаете смещение в сторону определенных токенов и не добавляете при этом стоп-токены (указывающие на конец вывода).

После обработки входных параметров классификатор считывает примеры с диска ❹ и классифицирует отзывы ❺, одновременно обновляя счетчики ❻, данные которых в итоге выводит ❼.

Посмотрим, как можно воспроизвести все версии классификатора, которые мы обсуждали до сих пор. Введя указанную ниже команду, вы должны получить версию классификатора без настроек, при условии, что файл `reviews.csv` находится в том же каталоге, что и сам код:

```
python tunable_classifier.py reviews.csv
    gpt-3.5-turbo 0 "" 0 ""
```

Обратите внимание, что мы не указываем токены, которым требуется отдать приоритет (оставляем пустую строку), не ограничиваем длину вывода (заданное значение `0` говорит об отсутствии ограничений) и задаем количество примеров в промпте равным `0` (следовательно, путь к файлу с примерами можно не прописывать).

Приведенная ниже команда, напротив, создаст версию, которая ограничит длину вывода, одновременно приоритизируя токены, соответствующие нашим меткам классов:

```
python tunable_classifier.py reviews.csv
    gpt-3.5-turbo 1 "981,29875" 0 ""
```

Наконец, последний вариант, который мы обсуждали, — с одним примером в каждом промпте и теми же настройками модели — можно запустить с помощью следующей команды (при условии, что файл `train_reviews.csv` находится в том же репозитории, что и код):

```
python tunable_classifier.py reviews.csv
              gpt-3.5-turbo   1   "981,29875"   1
"train_reviews.csv"
```

Смело тестируйте новые комбинации, которые мы не рассматривали!

9.7. Дообучение, или тонкая настройка

До сих пор мы делали все возможное, чтобы добиться максимальной производительности существующих моделей. Эти модели обучались для решения задач, которые, возможно, похожи на нашу, но *не точно такие же*. Правда, было бы чудесно получить модель, настроенную специально для решения нашей задачи? Это возможно, если применить дообучение. Посмотрим, как реализовать дообучение на практике с моделями OpenAI.

Дообучение означает, что мы берем существующую модель, например GPT-3.5 Turbo от OpenAI, и специализируем ее для

решения интересующей нас задачи. Конечно, можно было бы обучить модель с нуля. Но обычно это непомерно дорого, к тому же зачастую сложно найти объем целевых данных, необходимый для обучения большой модели. Поэтому гораздо эффективнее использовать дообучение.

Обычно дообучение — последнее, что стоит пробовать при максимизации производительности для конкретной задачи. Причина в том, что дообучение требует определенных первоначальных вложений времени и денег. Во время дообучения мы платим компании OpenAI за создание версии одной из ее базовых моделей, настроенной специально для решения нашей задачи. Цена зависит от размера обучающих данных и количества раз, которое эти данные считываются (то есть количества *эпох*). Например, на момент написания книги дообучение модели GPT-3.5 Turbo стоило около 0,8 цента за 1000 токенов обучающих данных за эпоху. Кроме того, после дообучения мы платим за использование дообученной модели. Цена за ее токен выше, чем в базовой версии. Это логично: ведь, по крайней мере теоретически, дообученная модель должна эффективнее решать нашу конкретную задачу.

Одно из возможных преимуществ дообучения — повышение точности вывода модели. Другой вероятный плюс заключается в возможности сократить длину промпта. При использовании универсальной модели в промпт необходимо добавить описание задачи, которую нужно выполнить (вместе со всеми релевантными данными). Дообученная модель, напротив, должна быть специализирована для выполнения одной задачи и хорошо с ней справляться. Если модель предназначена для решения лишь одной задачи, то ее описание можно вообще исключить из промпта, поскольку оно подразумевается. Помимо описания задачи, можно опустить и другую информацию, полезную для универсальной модели, но не

являющуюся необходимой для специализированной. Например, в промпт для универсальной модели может потребоваться добавить примеры, которые позволят получить вывод надлежащего качества, а при работе с дообученной версией это делать не обязательно.

В нашем конкретном сценарии задача состоит в том, чтобы сопоставлять отзывы с меткой класса, отражающей их тональность (отношение автора отзыва). Ранее мы прописывали задачу классификации в промпте (и даже предоставляли несколько полезных примеров). Сейчас при дообучении модели, полагаю, эти инструкции можно опустить. Точнее, теперь мы можем обойтись без промптов наподобие этого (содержит примеры отзывов ❶ с инструкциями ❷ и примерами ответов ❸, а также отзыв для классификации ❹ и соответствующие инструкции ❺):

```
Не стану отрицать, что, покупая этот товар на eBay,
я ожидал многого...                                    ❶ Пример отзыва
Отзыв положительный или отрицательный?                 ❷ Инструкция
Ответ ("pos"/"neg"):neg                                ❸ Эталонный ответ
В научно-фантастическом фильме я готов закрыть глаза
на многие огрехи, но это было уже почти невыносимо...  ❹ Отзыв для классификации
Отзыв положительный или отрицательный?                 ❺ Инструкция
Ответ ("pos"/"neg"):
```

Вместо этого предположим, что модель уже знает, что она должна классифицировать отзывы и какие метки классов доступны. Тогда промпт можно упростить:

```
В научно-фантастическом фильме я готов закрыть
глаза на многие огрехи, но это было уже почти
невыносимо....
```

В этом промпте содержится только отзыв, который необходимо классифицировать. Мы предполагаем, что вся другая информация, характерная для задачи (например, инструкции и примеры), уже известна модели по умолчанию. Как вы, конечно, заметили, этот промпт намного короче

предыдущего. Следовательно, мы *можем* сэкономить деньги, используя вместо базовой модели дообученную. Однако помните, что при использовании дообученной модели стоимость токена выше, чем в базовой версии. Соответствующие расчеты мы произведем позже, а пока посмотрим, удастся ли вообще, используя дообучение, заставить такие лаконичные промпты работать на практике.

9.8. Генерация обучающих данных

Сначала необходимо сгенерировать обучающие данные для дообучения. Для этого мы используем отзывы со связанными метками классов, содержащиеся в файле `train_reviews.csv`, доступном по ссылке **Review Training**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе. Компания OpenAI требует, чтобы обучающие данные для дообучения предоставлялись в специальном формате. Прежде чем приступать к дообучению, придется преобразовать наши данные из формата `.csv` в требуемый формат.

Обучающие данные для дообучения чат-моделей OpenAI обычно представлены в виде успешных взаимодействий с моделью (то есть примеров, когда она выдает вывод, который мы в идеале хотим получить). В случае чат-моделей OpenAI такие взаимодействия описываются с помощью истории сообщений. Каждому сообщению соответствует объект словаря Python. Например, в коде ниже показан успешный ответ модели на основе упомянутого ранее примера отзыва:

```
{'messages':[
       {'role':'user', 'content':'я готов закрыть
глаза на многие огрехи...'},
    {'role':'assistant', 'content':'neg'}
```

```
]}
```

Отзыв отрицательный (то есть его автор не рекомендует фильм), и поэтому в идеале хотелось бы, чтобы модель сгенерировала сообщение, содержащее единственный токен `neg`. Такое взаимодействие и показано в коде.

Чтобы дообучение было результативным, обычно требуется использовать от 50 до нескольких тысяч примеров. Добавление большего количества примеров для дообучения может повысить производительность, но и стоит дороже. С другой стороны, это однократная трата, ведь одну и ту же дообученную модель можно повторно использовать для потенциально большого набора данных (и плата за использование дообученной модели не зависит от объема обучающих данных, представленных для дообучения). Файл примера (`reviews_train.csv`) содержит 100 примеров и, следовательно, соответствует диапазону объемов данных, в котором дообучение может стать результативным.

Компания OpenAI требует предоставлять данные для дообучения в формате JSON-lines (в именах таких файлов обычно есть окончание `.jsonl`). Файлы, соответствующие этому формату, по сути, содержат в каждой строке один словарь Python. В данном случае каждая строка описывает одно успешное взаимодействие с моделью (используя тот же формат, что и в предыдущем примере). Чтобы упростить работу с файлами JSON-lines из Python, мы используем библиотеку `jsonlines`. Сначала перейдите в терминал и установите библиотеку, введя команду:

```
pip install jsonlines==4.0
```

Теперь с помощью этой библиотеки можно преобразовать данные из формата `.csv` в формат, требуемый OpenAI (листинг

9.3).

Листинг 9.3. Генерация обучающих данных для дообучения

```
import argparse
import jsonlines
import pandas

def get_samples(df):                                    ❶
```

Генерирует обучающие данные

```
    """ Сгенерировать образцы из фрейма данных.

    Аргументы:
        df: фрейм данных, содержащий образцы.

    Выводит:
            Список образцов в формате OpenAI для
дообучения.
    """
    samples = []
    for _, row in df.iterrows():
```

❷ Создает сообщение пользователя

```
        text = row['text']
                  user_message  =  {'role':'user',
'content':text}
```

❸ Создает сообщение ассистента

```
        label = row['sentiment']
           assistant_message = {'role':'assistant',
'content':label}
```

```
                sample  =  {'messages':[user_message,
assistant_message]}
        samples += [sample]

    return samples

if __name__ == '__main__':
```

❹ Разбирает аргументы командной строки

```
    parser = argparse.ArgumentParser()
          parser.add_argument('in_path',  type=str,
help='Путь к входным данным')
         parser.add_argument('out_path',  type=str,
help='Путь к выходным данным')
    args = parser.parse_args()

    df = pandas.read_csv(args.in_path)
    samples = get_samples(df)
```

❺ Сохраняет обучающие данные в новом формате

```
       with  jsonlines.open(args.out_path,  'w')  as
file:
        for sample in samples:
            file.write(sample)
```

В этом коде подготовка примеров в требуемом формате выполняется с помощью функции `get_samples` ❶. Входным параметром является объект `DataFrame` из библиотеки `pandas` (параметр `df`), который хранит обучающую выборку в стандартном виде (предполагается, что столбец `text` содержит отзывы, а столбец `sentiment` — связанные с ними метки

классов). Мы превращаем каждый пример в успешный обмен сообщениями с моделью. Сначала создаем сообщение, отправляемое пользователем ❷ и содержащее только текст отзыва. Затем добавляем желаемое ответное сообщение, которое должна сгенерировать модель (связанное с ролью «ассистента») ❸ . Полный набор обучающих примеров представляет собой список обменов сообщениями, каждое из которых подготовлено в формате JSON-lines.

В качестве входных данных должен быть представлен путь к файлу формата `.csv` с обучающими примерами, а также путь к файлу вывода ❹ . Файл вывода соответствует формату JSON-lines, поэтому мы, как правило, задаем выходной путь с окончанием `.jsonl`. После преобразования входного файла из формата `.csv` в формат для дообучения с помощью библиотеки `jsonlines` мы записываем преобразованные примеры в файл формата JSON-lines ❺.

Как обычно, вводить код листинга вручную не требуется. Его можно найти по ссылке **Prepare Fine-Tuning** на сайте книги, в разделе дополнительных материалов к главе. Запустите код из терминала с помощью следующей команды (предполагается, что файл `train_reviews.csv` находится в том же репозитории, что и код):

```
python prep_fine_tuning.py train_reviews.csv
train_reviews.jsonl
```

Возможно, вы решите вручную проверить файл `train_reviews.jsonl`, который (надеюсь) был сгенерирован после запуска этой команды. В каждой строке должен отобразиться один обучающий пример, представленный в виде словаря Python.

9.9. Запуск задачи дообучения

Теперь, когда у нас есть обучающие данные в корректном формате, можно создать задачу дообучения на платформе OpenAI. Конечно, поскольку модель хранится только на этой платформе, нам не удастся выполнить дообучение самостоятельно. Вместо этого мы отправим свои обучающие данные в компанию OpenAI и попросим использовать их для создания настроенной модели. Чтобы получить в итоге настроенную модель, сначала необходимо выбрать базовую. В данном случае мы начнем с модели GPT-3.5 Turbo (что упростит сравнение с результатами, которые мы получали ранее).

Можно создать задачу дообучения, используя следующий фрагмент кода (предположим, что `in_path` — это путь к файлу, содержащему обучающие данные):

```
import openai
client = openai.OpenAI()

reply = client.files.create(
    file=open(in_path, 'rb'), purpose='fine-tune')
```

Объект `reply` будет содержать объект Python с метаданными о задаче дообучения (при условии успешного создания задачи). Самое важное: в поле `reply.id` мы получаем идентификатор только что созданной задачи. Для выполнения задач дообучения обычно требуется некоторое время (например, выполнение описываемой здесь задачи занимает около 15 минут). Следовательно, придется подождать, пока будет создана дообученная модель. Идентификатор задачи позволяет проверять ее статус и получить идентификатор вновь

созданной модели, как только она станет доступна. Статус можно узнать, используя следующий фрагмент кода на Python:

```
reply = client.fine_tuning.jobs.retrieve(job_id)
```

Поле `reply.status` сообщает статус задачи дообучения, который в конечном счете достигнет значения `succeeded`. По завершении этого этапа идентификатор дообученной модели будет доступен в поле `reply.fine_tuned_model`.

Код, показанный в листинге 9.4, запускает процесс дообучения, ждет, пока соответствующая задача завершится, и, наконец, выводит идентификатор сгенерированной модели. При наличии пути к файлу с обучающими данными код сначала загружает этот файл ❶. Он получает идентификатор файла, присвоенный компанией OpenAI, и использует его для создания задачи дообучения ❷. Затем выполняются итерации, пока задача не завершится успешно ❸. На каждой итерации выводится таймер (измеряющий секунды с начала задачи дообучения) и проверяются обновления статуса задачи ❹. Наконец, мы получаем идентификатор модели и выводим его ❺.

Листинг 9.4. Дообучение модели GPT с использованием обучающих данных

```
import argparse
import openai
import time

client = openai.OpenAI()

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
```

```
          parser.add_argument('in_path',  type=str,
help='Путь к входному файлу')
    args = parser.parse_args()


     reply = client.files.create(                    ❶
```

Передает обучающие данные в компанию OpenAI

```
                    file=open(args.in_path,  'rb'),
purpose='fine-tune')
    file_id = reply.id


       reply  =  client.fine_tuning.jobs.create(  ❷
```

Создает задачу дообучения

```
            training_file=file_id, model='gpt-3.5-
turbo')
    job_id = reply.id
    print(f'Job ID: job_id')

    status = None
    start_s = time.time()


     while not (status == 'succeeded'):            ❸
```

Выполняет итерации, пока задача не завершится

```
        time.sleep(5)
        total_s = time.time() - start_s
                print(f'Fine-tuning  since  total_s
seconds.')
```

❹ Получает статус задачи

```
                                                reply      =
client.fine_tuning.jobs.retrieve(job_id)
        status = reply.status
        print(f'Status: {status}')
```

❺ Получает идентификатор дообученной модели

```
print(f'Fine-tuning is finished!')
model_id = reply.fine_tuned_model
print(f'Model ID: {model_id}')
```

Этот код доступен по ссылке **Start Fine-Tuning**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе. Скачайте файл и запустите код, используя команду ниже (где `train_reviews.jsonl` — ранее сгенерированный файл):

```
python fine_tune.py train_reviews.jsonl
```

Если позволить скрипту работать до завершения, то отобразится вывод, подобный указанному ниже (это, конечно, лишь часть вывода; многоточиями обозначены пропущенные строки):

```
Job ID: ...
Fine-tuning since 5.00495171546936 seconds.
Status: validating_files
...
Fine-tuning since 46.79299879074097 seconds.
Status: running
...
Fine-tuning since 834.6565797328949 seconds.
Status: succeeded
Fine-tuning is finished!
```

```
Model ID: ft:gpt-3.5-turbo-0613...
```

После вывода идентификатора задачи мы будем получать регулярные обновления о статусе задачи, который обычно меняется с `validating_files` на `running` и (надеемся) на `succeeded`. Проблема в том, что завершение задачи может занять некоторое время (в предыдущем примере около 14 минут). Если вы не хотите, чтобы скрипт работал непрерывно (например, вам нужно выключить компьютер), то его можно остановить после того, как задача дообучения начнет выполняться (вы узнаете о начале этого процесса, так как в этот момент скрипт выведет идентификатор задачи). Задача дообучения продолжится по плану на серверах компании OpenAI. В зависимости от настроек, возможно, вам даже придет электронное письмо с уведомлением о ее завершении. В ином случае можно периодически запускать скрипт, проверяющий статус задачи (листинг 9.5).

Листинг 9.5. Проверка статуса задачи дообучения

```
import argparse
import openai

client = openai.OpenAI()

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
           parser.add_argument('job_id',   type=str,
help='ID задачи дообучения')
    args = parser.parse_args()
```

❶ Получает и выводит метаданные

```
                                        job_info        =
client.fine_tuning.jobs.retrieve(args.job_id)
    print(job_info)
```

При наличии идентификатора задачи (полученного из вывода листинга 9.4), скрипт получает и выводит метаданные задачи ❶ , в том числе ее статус и идентификатор получившейся в итоге модели (после успешного завершения задачи).

9.10. Использование дообученной модели

Поздравляю! Вы создали специализированную модель, дообученную для выполнения интересующей вас задачи (классификации отзывов). Как ее использовать? К счастью, это несложно сделать с помощью библиотеки компании OpenAI. Вместо имени одной из стандартных моделей (например, `gpt-3.5-turbo`) теперь указывается идентификатор дообученной модели — метка-заполнитель `[Fine-tuned model ID]` заменяется фактическим идентификатором модели:

```
import openai
client = openai.OpenAI()

response = client.chat.completions.create(
    model='[Fine-tuned model ID]',
    messages=[
        {'role':'user', 'content':prompt}
        ]
    )
```

Как и раньше, мы предполагаем, что в переменной `prompt` содержится текст промпта. Однако промпты для нашей дообученной модели различаются. Ранее мы предоставляли в промпте задачу классификации и текст отзыва. Теперь мы обучили модель сопоставлять только текст отзыва с соответствующим классом. Это означает, что функция генерации промптов упрощается (фактически можно утверждать, что создавать специальную функцию больше не нужно):

```
def create_prompt(text):
    """ Сгенерировать промпт для классификации отзывов по тональности.

    Аргументы:
        text: текст для классификации.

    Выводит:
        Промпт для классификации текста.
    """
    return text
```

Вместо генерации многокомпонентных промптов мы возвращаем текст отзыва для классификации. Полагаю, вам любопытно, что произойдет, если отправить упрощенный промпт в исходную модель (`gpt-3.5-turbo`). Вы получите примерно такой вывод:

```
Метка: Мне понятна ваша озабоченность по поводу сцен курения в кинофильмах, особенно в предназначенных для детской и подростковой аудитории. Курение в фильмах может повлиять на молодых зрителей, и такое поведение потенциально
```

```
может начать казаться им нормальным. Однако важно
отметить, что не все случаи демонстрации сцен
курения в фильмах являются результатом
целенаправленной рекламы и не всегда их спонсорами
выступают табачные компании…
```

Очевидно, модель не понимает наши намерения — то есть того, каких действий с поступающими в качестве входных данных отзывами мы ожидаем от нее. Вместо генерации корректных меток классов она пишет развернутые анализы, комментируя основные моменты, затронутые в отзывах. Это вполне ожидаемо. Представьте, что вы получили текст отзыва без каких-либо дальнейших инструкций. Как бы вы узнали, что его требуется классифицировать, не говоря уже о корректных метках возможных классов? Сделать это было бы почти невозможно, и то же самое относится к языковым моделям.

Но если мы возьмем дообученную модель и предоставим ей в качестве входных данных те же самые промпты, то получим следующий вывод:

```
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: pos; Истинная метка: pos
Метка: pos; Истинная метка: neg
Метка: pos; Истинная метка: neg
Метка: neg; Истинная метка: neg
Метка: neg; Истинная метка: pos
Метка: neg; Истинная метка: neg
Количество правильных меток      :     7      ❶
```

```
Количество использованных токенов:        2085    ❷
```

Сокращенное потребление токенов

Обратите внимание, что, даже не задавая параметры настройки (и не предоставляя примеры в промпте), мы теперь получаем точность 70 % ❶ , а не 60, как в исходной версии! Кроме того, количество использованных токенов по сравнению с исходной версией сократилось примерно на 200 ❷ . Все потому, что мы опускаем во всех промптах инструкции (и метки классов).

Итак, мы выяснили, что можно дообучить модель для точной классификации отзывов, одновременно сократив длину промпта. Но остается вопрос: стоило ли оно того? Проведем расчеты, чтобы это выяснить. Не будем учитывать стоимость создания дообученной модели, ведь ее требуется дообучить лишь однажды (а в примере предполагается, что нужно проанализировать отзывы за год). Без дообучения можно добиться той же точности (70 %), используя универсальную модель и параметры настройки (задав смещение и ограничение на количество токенов вывода). В этом случае для десяти тестовых отзывов потребуется 2228 токенов. После дообучения для тестовых отзывов нужно всего 2085 токенов. Однако, работая с универсальной моделью, мы платим 0,05 цента за 1000 входных токенов. А при использовании дообученной модели мы платим 0,3 цента за 1000 токенов. Следовательно, после дообучения цена за токен выросла в шесть раз! В этом конкретном сценарии небольшое сокращение количества обрабатываемых токенов не окупает более высокую плату за токен.

В целом дообучение может помочь существенно повысить эффективность модели и, возможно, снизить стоимость. Но имейте в виду, что оно влечет за собой определенные затраты.

Прежде чем использовать дообученную модель в производственной среде, оцените ее в тестовой, проведите расчеты и убедитесь, что ее применение целесообразно!

Резюме

- Настройка параметров модели может влиять на ее производительность и стоимость.
- Можно ограничивать длину вывода и вводить смещение логитов токенов.
- Не всегда стоит использовать самую большую доступную модель, так как это увеличивает затраты.
- Определите оптимальную модель для своей задачи, оценив ее на примерах.
- Структура промпта может значительно повлиять на производительность модели.
- Для обучения на нескольких примерах добавьте в промпт примеры правильно решенных задач.
- Дообучение позволяет настроить базовые модели для решения интересующих вас задач. Благодаря такой специализации можно сократить длину промпта.
- Дообучение требует затрат, которые пропорциональны объему данных для обучения. Кроме того, оно увеличивает стоимость токена при использовании полученной модели.

8 Или для русскоязычных моделей на сайте https://mera.a-ai.ru/ru/text. — *Примеч. науч. ред.*

10. Фреймворки

В этой главе

- Создание приложений с помощью LangChain.
- Решение сложных задач с помощью агентов.
- Запрос данных с помощью LlamaIndex.

До сих пор для взаимодействия с языковыми моделями мы в основном пользовались библиотекой Python компании OpenAI. Она предоставляет базовый функционал для отправки промптов и получения ответов от GPT и других моделей OpenAI (а также возможности настройки и дообучения). Библиотеки других провайдеров, таких как Anthropic и Cohere, предлагают схожий функционал. Пока ваши задачи анализа данных просты, этого, скорее всего, будет достаточно. Но что, если для анализа данных потребуется сложный многоступенчатый конвейер, потенциально объединяющий множество разных форматов данных?

Тогда вы, пожалуй, захотите перейти на более мощный фреймворк. В настоящее время активно развиваются несколько высокоуровневых фреймворков для создания сложных приложений на основе языковых моделей. В этой главе мы обсудим два наиболее популярных из них: LangChain и LlamaIndex. Первый представляет собой универсальный фреймворк для создания приложений с использованием больших языковых моделей. Кроме того, он поставляется с множеством полезных встроенных компонентов, реализующих популярные сценарии использования языковых моделей.

LlamaIndex, в свою очередь, специализируется на сценариях, где языковым моделям необходимо взаимодействовать с большими наборами данных.

Чтобы понять, как работает LangChain, сначала напишем простой конвейер классификации текста с использованием этого фреймворка. Затем рассмотрим некоторые его расширенные возможности. В частности, мы обсудим механизм работы *агентов*, построенных на базе языковых моделей. Создание агента означает, что мы даем модели право самостоятельно выбирать методы выполнения задачи, используя набор инструментов, предоставленный пользователем. С помощью таких агентов вы научитесь самостоятельно решать сложные аналитические задачи, комбинируя разные инструменты для работы с данными из различных источников. Далее вы увидите, как LlamaIndex легко поглощает большие объемы данных в различных форматах и делает их пригодными для использования языковыми моделями. В основе работы этого фреймворка лежат недорогие модели, которые переводят фрагменты данных и задачи анализа в векторное представление, после чего сопоставляют задачи с данными на основе сходства векторов. Наконец, мы сравним два фреймворка и проанализируем их преимущества и недостатки по сравнению с библиотеками OpenAI и иных разработчиков языковых моделей.

10.1. Знакомство с LangChain

Создать сложное приложение на основе языковых моделей можно с помощью LangChain. Фреймворк был запущен в октябре 2022 года и быстро набирает популярность (что привело к созданию соответствующего стартапа в апреле 2023 года). На момент написания книги LangChain все еще

стремительно развивался. Обязательно запускайте код из этого раздела в корректной версии LangChain (поскольку в будущих версиях интерфейсы могут измениться).

Название LangChain говорит само за себя: оно отсылает к языковым моделям (Lang) и цепочкам (Chain). В терминологии LangChain *цепочка* — это просто последовательность действий. Каждое из них может соответствовать вызову языковой модели, обработке данных или вызову произвольного инструмента. Важный момент: мы больше не рассчитываем, что один вызов языковой модели решит нашу проблему (как было в большинстве случаев, которые мы обсуждали в этой книге). Вместо этого мы полагаем, что потребуется сложная сеть связанных компонентов. Именно в таких сценариях раскрываются сильные стороны LangChain.

Чтобы использовать этот фреймворк, его сначала необходимо установить. Перейдите в терминал и выполните следующую команду:

```
pip install langchain==0.1.13
```

Как уже говорилось, для запуска приведенных ниже примеров кода необходимо установить подходящую версию LangChain. Фреймворк меняется так быстро, что в другой версии код может не работать.

Помимо ядра LangChain, возможно, потребуется установить библиотеки, поддерживающие языковые модели от конкретных провайдеров. В разделах ниже мы будем использовать модели OpenAI. Выполните в терминале следующую команду (и еще раз убедитесь, что используете указанную версию):

```
pip install langchain-openai==0.1.1
```

Поддержка других провайдеров, таких как Anthropic и Cohere, также доступна.

После выполнения этих команд вы готовы запускать примеры проектов.

10.2. Классификация отзывов с помощью LangChain

Одним из первых наших проектов был анализ текста с помощью языковых моделей. Помните главу 4? Мы использовали модели, чтобы классифицировать отзывы на основе их тональности (это рекомендация или предупреждение?). То же самое будем делать и в этом проекте, но в коде станем использовать LangChain. Сравнив код для этого фреймворка с исходным, вы сразу увидите, насколько он способен упростить создание приложений с помощью языковых моделей.

10.2.1. Обзор

Мы создадим цепочку для классификации текстовых документов. Вообще, цепочка LangChain может состоять из множества действий, каждое из которых реализуется путем вызова языковой модели или общей функции Python (например, для приведения результатов вызовов языковой модели к заданному формату). На самом деле термин «цепочка» немного вводит в заблуждение. Цепочку можно представить как последовательность действий, однако цепочки LangChain этим не ограничиваются. Например, они могут содержать параллельные шаги, а также условное выполнение. Однако для простого приложения классификации текста нам не понадобятся такие расширенные возможности. Вместо этого

мы ограничимся простой цепочкой, состоящей всего из нескольких действий.

Она будет интегрировать несколько стандартных компонентов, предлагаемых фреймворком LangChain. Первый компонент — шаблон промпта. Как и в главе 4, в нем описывается задача классификации и ожидаемый формат вывода. Наверное, вам любопытно, что изменилось по сравнению с предыдущей версией кода. В конце концов, мы ведь постоянно обсуждаем шаблоны промптов. Разница в том, что LangChain вводит специальный класс для представления этих шаблонов. Он предлагает различные удобные функции: например, для создания и заполнения таких шаблонов. В то же время в LangChain есть специальное хранилище — хаб, где можно выкладывать свои шаблоны промптов, а также скачивать чужие (и не только). В нашем простом сценарии эти расширенные возможности не потребуются. Достаточно будет просто создать экземпляр шаблона, передав в качестве входных данных один параметр (текст для классификации).

Второе действие в цепочке — вызов языковой модели. Мы так уже работали с моделями ранее, но LangChain предлагает ряд удобных возможностей, которые надстраиваются над стандартным объектом модели. К примеру, можно вести журнал всех обращений к модели, а еще фреймворк предлагает удобные функции для различных сценариев вызова (например, пакетную и потоковую обработку). Опять же, в данном случае мы не будем применять эти расширенные возможности. Просто используем промпт (первое действие в нашей цепочке), чтобы модель могла сгенерировать ответ.

Третье действие в цепочке — использование парсера, который извлекает строку ответа из вывода, сгенерированного языковой моделью. Из главы 3 вы наверняка помните, что языковые модели компании OpenAI генерируют подробные

ответы, содержащие один или несколько вариантов, а также различные типы метаданных (например, информацию об использовании токенов). Парсер автоматически извлекает строку ответа, которую мы ищем, из объекта результата (это работает как для моделей OpenAI, так и для всех других провайдеров). Результатом конвейера становится единственный токен, указывающий, является ли входной отзыв рекомендацией. Все три этапа работы конвейера показаны на рис. 10.1.

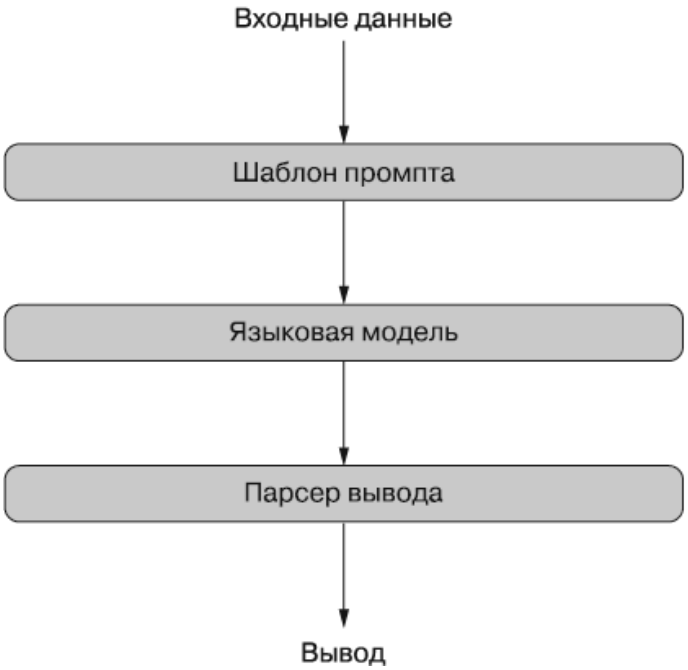


Рис. 10.1. Компоненты в цепочке классификации LangChain

10.2.2. Создание цепочки классификации

Пора реализовать нашу цепочку на Python! Прежде всего нам потребуется шаблон промпта. Возьмем тот же шаблон, что и в главе 4, но на этот раз укажем класс `ChatPromptTemplate`, используемый LangChain:

```
from langchain_core.prompts.chat import
ChatPromptTemplate
prompt = ChatPromptTemplate.from_template(
    '{text}\n'                                    ❶
```

Метка-заполнитель для текста

```
    'Отзыв положительный или отрицательный?\n'
    'Ответ ("Положительный"/"Отрицательный")\n')
```

Вы могли заметить отсылку к чат-моделям (в конце концов, создаваемый класс называется `ChatPromptTemplate`). Как обсуждалось в главе 3, чат-модели обрабатывают историю предыдущих сообщений, а не одно входное сообщение. Многие из недавно выпущенных моделей являются чат-моделями. В LangChain чат-моделям требуется специализированный шаблон промпта (который создает экземпляр последовательности сообщений, а не одного текста). Именно такой тип шаблона мы здесь создаем. Такой же шаблон использовался в главе 4. Он содержит метку-заполнитель ❶ для входного текста, который нужно классифицировать. Для обозначения меток-заполнителей в шаблонах промптов обычно используются фигурные скобки ({}), которые заменяются конкретными значениями при заполнении промпта.

Далее, нам нужна языковая модель для обработки промптов. Этот код создает экземпляр модели GPT-4o компании OpenAI:

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(
    model='gpt-4o', temperature=0,
    max_tokens=1)
```

Класс `ChatOpenAI` охватывает все чат-модели компании OpenAI. Он импортируется из пакета `langchain_openai`, который содержит функционал для поддержки использования моделей OpenAI в LangChain. У других провайдеров, таких как Anthropic и Cohere, есть собственные пакеты, предлагающие сопоставимый функционал для их моделей (обратите внимание, что эти пакеты необходимо устанавливать отдельно через систему управления пакетами pip). Параметры в конструкторе `ChatOpenAI` могут показаться знакомыми: мы выбираем модель (`gpt-4o`), задаем значение температуры (`temperature`), равное `0` (чтобы уменьшить степень рандомизации в выводе), и ограничиваем максимальное количество выходных токенов одним (поскольку обе возможные метки классов, `Положительный` и `Отрицательный`, состоят из одного токена).

Теперь необходимо извлечь строку ответа из (расширенного) вывода языковой модели. Это легко сделать с помощью класса `StrOutputParser`. Парсеры вывода LangChain позволяют выполнить широкий спектр преобразований над результатом вызова модели. В данном случае требуется очень простое преобразование — извлечение нужной строки ответа. Этот фрагмент кода создает соответствующий парсер:

```
from langchain_core.output_parsers.string import StrOutputParser
parser = StrOutputParser()
```

И наконец, объединим все компоненты в цепочку. Для этого можно использовать язык выражений LangChain (LangChain Expression Language, LCEL). Если вы пользователь Linux, то этот синтаксис наверняка вам знаком:

```
from langchain_core.runnables.passthrough import RunnablePassthrough
chain = ({'text':RunnablePassthrough()} | prompt | llm | parser)
```

Чтобы использовать вывод одной операции в качестве входных данных для следующего действия, соединим их с помощью символа |. Команда создает цепочку, связывающую ранее упомянутые компоненты. Дополнительно она указывает формат входных данных, требуемых цепочкой. В нашем случае шаблон промпта содержит метку-заполнитель для текста, который требуется классифицировать.

В начале цепочки мы помечаем этот параметр как `RunnablePassthrough`. Так мы получаем больше возможностей при передаче входных параметров в цепочку. Например, в коде ниже показано, как обработать список входных данных с помощью ранее созданной цепочки:

```
inputs = ['Это великолепный фильм!', 'Это плохой фильм!']
outputs = chain.batch(inputs)
```

10.2.3. Объединение компонентов

Пора закончить код для классификации текста. Код в листинге 10.1 принимает в качестве входных данных путь к файлу в формате `.csv`, содержащему столбец `text`. После запуска кода генерируется файл вывода с дополнительным столбцом `class`, содержащим результаты классификации. Другими словами, код делает в точности то же самое, что и код из главы 4, но на этот раз с использованием LangChain.

Листинг 10.1. Классификация отзывов по тональности с помощью LangChain

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts.chat import
ChatPromptTemplate
from langchain_core.output_parsers.string import
StrOutputParser
from langchain_core.runnables.passthrough import
RunnablePassthrough

import argparse
import pandas as pd

def create_chain():                                    ❶
```

Создает цепочку

```
    """ Создает цепочку для классификации текста.

    Выводит:
        цепочка для классификации текста.
    '''''
      prompt = ChatPromptTemplate.from_template(  ❷
```

Создает шаблон промпта

```
        '{text}\n'
        'Отзыв положительный или отрицательный?\n'
                                            'Ответ
("Положительный"/"Отрицательный")\n')
    llm = ChatOpenAI(                                  ❸
```

Создает объект LLM

```
        model='gpt-4o', temperature=0,
        max_tokens=1)
```

```
    parser = StrOutputParser()                    ❹
```

Создает парсер вывода

❺ Создает цепочку

```
        chain = ({'text':RunnablePassthrough()} |
prompt | llm | parser)
    return chain

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
        parser.add_argument('file_path', type=str,
help='Путь к входному файлу')
    args = parser.parse_args()

    df = pd.read_csv(args.file_path)              ❻
```

Считывает данные

```
    chain = create_chain()                        ❼
```

Создает цепочку

```
    results = chain.batch(list(df['text']))       ❽
```

Использует ее

```
    df['class'] = results                         ❾
```

Сохраняет вывод

```
    df.to_csv('result.csv')
```

Функция `create_chain` ❶ реализует шаги, которые мы обсуждали в предыдущем подразделе. Она генерирует шаблон промпта для классификации ❷ , затем чат-модель ❸ и, наконец, парсер вывода ❹ . Результатом является цепочка, соединяющая все эти компоненты ❺.

После считывания параметров командной строки код считывает входные данные ❻ , создает соответствующую цепочку ❼ и, наконец, применяет цепочку к списку входных текстов ❽. Результаты классификации добавляются к входным данным и сохраняются на диске ❾.

10.2.4. Проверка в работе

Как обычно, код из листинга 10.1 доступен по ссылке **Listing 1**, размещенной на сайте книги, в разделе дополнительных материалов к этой главе. Скачайте код (файл `listing1.py`) и (по желанию) файл, содержащий отзывы для классификации (например, `reviews.csv` из главы 4). Откройте терминал и перейдите в папку с кодом. При условии, что файл `reviews.csv` находится в той же папке, выполните следующую команду:

```
python listing1.py reviews.csv
```

Проверьте папку с кодом. В ней должен появиться новый файл `result.csv` с желаемыми результатами классификации. Пока что мы лишь убедились, что с помощью LangChain можно решать те же задачи, что и напрямую через библиотеки OpenAI (хотя, пожалуй, код фреймворка LangChain аккуратнее). В следующем разделе мы убедимся, что возможности LangChain этим не ограничиваются.

10.3. Агенты: большая языковая модель в роли управляющего компонента

Полагаю, до сих пор вы считали языковые модели инструментами (хоть и очень сложными). На основе предоставленных вами входных данных модель генерирует вывод. Когда возможностей модели недостаточно для обработки данных, разработчику приходится добавлять необходимую инфраструктуру самостоятельно. Представьте, что создаете набор вопросов и ответов по математике. Понимая, что языковые модели плохо справляются с вычислениями (звучит иронично, ведь они являются компьютерными программами, но это действительно так), вы можете использовать следующий подход: на основе вопроса пользователя языковая модель переводит ввод в математическую формулу. Затем простой инструмент-калькулятор разбирает эту формулу и вычисляет результат, который и отправляется пользователю.

Пока неплохо. Ситуация усложняется, когда у вас не один, а несколько математических инструментов. Допустим, один инструмент решает дифференциальные уравнения, а другой — вычисляет простые арифметические выражения. В таких случаях можно расширить подход, добавив этап классификации, сопоставляющий пользовательский ввод с наиболее подходящим инструментом. Однако этот подход перестает работать в ситуациях, когда для ответа на вопрос пользователя может потребоваться не один инструмент, а разные, причем их придется вызывать многократно, возможно используя результат работы одного инструмента в качестве входных данных для последующих вызовов. В таких случаях прописывать вручную каждую возможную последовательность необходимых вызовов инструментов просто нереально.

Именно для подобных сценариев использования и нужны *агенты*. Это сравнительно новый способ применения больших языковых моделей. В его основе лежит смена перспективы. Вместо того чтобы рассматривать языковую модель как инструмент, используемый на определенном этапе в конвейере, спроектированном разработчиком, мы даем ей возможность действовать самостоятельно. Мы не пытаемся выстроить порядок применения модели и других шагов обработки (как было в предыдущем разделе), а предоставляем ей самой решать, какие этапы обработки применять и в каком порядке. Преимущество этого подхода в том, что он гораздо более гибкий и освобождает нас, разработчиков, от необходимости предвидеть каждый возможный сценарий и создавать соответствующую ветвь в логике обработки.

Агенты могут быть полезны при решении сложных задач анализа данных, где заранее не ясно, какие источники данных или методы обработки потребуются для удовлетворения запроса пользователя.

В подходе с агентами центральными являются два понятия: *агент* и его *инструменты*.

Инструмент может инкапсулировать какую угодно функциональность. Это функция, которую языковая модель может использовать, если сочтет необходимым. Когда мы применяем для работы с агентами LangChain или подобные фреймворки, инструмент обычно реализуется как функция Python. Каждый инструмент должен быть связан с описанием на естественном языке. Оно предоставляется языковой модели как элемент промптов. На основе описания модель решает, будет ли инструмент полезен в текущем контексте. Для использования инструмента ей требуется описание входных параметров и семантики вывода. Языковые модели, как и программисты-люди, эффективнее работают с инструментами,

если у параметров понятные названия, а к самому инструменту прилагается четкое описание. Агенты реализуются через языковые модели, поэтому полное описание всех доступных инструментов обычно предоставляется как часть входного промпта.

Агенты используют инструменты всякий раз, когда требуется решить сложную задачу, поставленную пользователем. Агенты реализуются через языковые модели. Дообучение, конечно, может сделать модель более эффективным агентом, но и обычные модели, в принципе, должны работать. Секрет превращения языковых моделей в агенты заключается не столько в самой модели, сколько в том, как формулируются промпты к ней.

Как минимум в соответствующих промптах содержатся следующие компоненты.

- Краткое описание задачи, которую должен решить агент (предоставляется пользователем).

- Список доступных инструментов вместе с описанием их функциональности, а также их входных и выходных параметров.

- Описание ожидаемого формата вывода. Оно позволяет сопоставить вывод языковой модели с вызовами инструментов.

При наличии подобного промпта языковая модель может сама решить, какие инструменты вызвать, и сформировать соответствующий вывод. Инфраструктура, реализующая агентный подход, разбирает вывод, сопоставляет его с соответствующими инструментами и значениями входных параметров и получает результат вызова. При следующей

итерации результат вызова инструмента добавляется во входной промпт. Таким образом, языковая модель, по сути, может *получить доступ* к результатам вызовов инструментов. На основе этого она может либо применить дополнительные инструменты (возможно, используя результаты предыдущих вызовов как входные данные), либо завершить работу, если готовый ответ уже получен.

Этот процесс обобщенно показан на рис. 10.2. Поставленная пользователем задача вместе с подробным описанием всех инструментов формирует входные данные для языковой модели. Вывод, сгенерированный ею, разбирается и преобразуется в действие: вызов инструмента (в этом случае команда вызова содержит значения для всех входных параметров инструмента) или завершение (в этом случае команда завершения содержит данные, которые модель считает ответом на входную задачу). Когда действием является вызов инструмента, производится соответствующий запуск. Результат добавляется в промпт, используемый при следующей итерации. Итерации продолжаются, пока модель не решит завершить работу (или пока не будет достигнут заданный пользователем лимит на количество итераций).

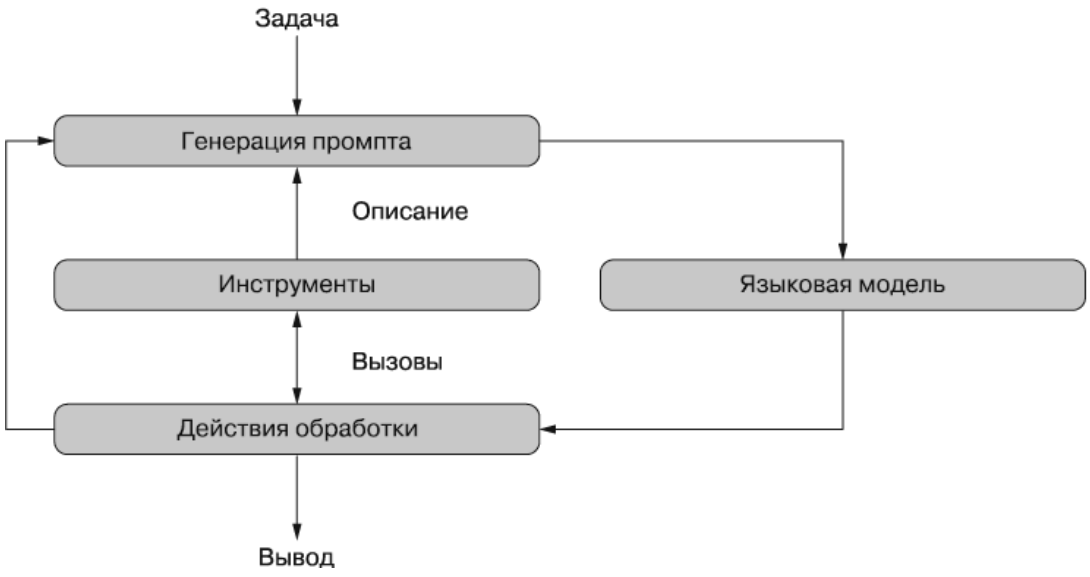


Рис. 10.2. Использование языковых моделей в качестве агентов. На основе промпта, описывающего задачу и доступные инструменты, языковая модель принимает решения о завершении и вызовах инструментов. Результаты вызовов инструментов добавляются в промпт, используемый для следующей итерации

Сейчас вам, наверное, любопытно, как выглядят соответствующие промпты. Рассмотрим стандартный шаблон, используемый для агентов в LangChain. Его можно скачать из хаба LangChain. Для этого сначала установите хаб, введя в терминале следующую команду:

```
pip install langchainhub==0.1.15
```

Затем запустите приведенный ниже код на Python, чтобы вывести стандартный шаблон для одного из самых популярных типов агентов:

```
from langchain import hub
prompt = hub.pull('hwchase17/react')
print(prompt.template)
```

Должен отобразиться следующий вывод:

```
❶ Общий сценарий

Ответь на следующие вопросы как можно точнее.
У тебя есть доступ к следующим инструментам:


{tools}                                                    ❷
Описание инструментов


Используй следующий формат:                                ❸
Описание формата
```

```
Вопрос: входной вопрос, на который ты должен ответить
Мысль: ты всегда должен обдумывать свои действия
Действие: действие, которое нужно предпринять, должно быть одним из [{tool_names}]
Входные данные действия: входные данные для действия
Наблюдение: результат действия
... (эта последовательность Мысль/Действие/Входные данные действия/Наблюдение может повторяться N раз)
Мысль: теперь я знаю окончательный ответ
Окончательный ответ: окончательный ответ на исходный входной вопрос

Начнем!

Вопрос: {input}                    ❹ Ввод пользователя
Мысль: {agent_scratchpad}          ❺ Предыдущие результаты
```

В этом шаблоне промпта описываются общий сценарий ❶ (есть вопрос, на который модель должна ответить), доступные инструменты ❷ и процесс решения задачи ❸ . Шаблон содержит несколько меток-заполнителей, представляющих собой описания инструментов ❷ , ввод от пользователя ❹ и результаты предыдущих итераций ❺ . Как мы увидим в следующих разделах, фреймворк LangChain предлагает различные удобные функции для создания и выполнения агентов на основе этого и похожих шаблонов промптов.

10.4. Создание агента для анализа данных

В этом разделе мы используем фреймворк LangChain, чтобы создать агент для анализа данных. Он сможет получать доступ к различным источникам, содержащим как структурированные, так и неструктурированные данные. Более того, агент будет сам решать, к каким из этих источников обращаться и в каком порядке. Он сможет даже использовать информацию, полученную из одного источника, для запроса ко второму источнику (например, получить доступ к структурированной базе данных о продажах видеоигр, чтобы определить самую продаваемую игру в конкретном году, а затем использовать название игры для поиска дополнительной информации в Интернете).

10.4.1. Обзор

Наш агент для анализа данных реализует подход, который мы обсуждали в предыдущем разделе. Он использует языковую модель, которая определяет, какие инструменты вызывать, их порядок и значения входных параметров. В примере мы предоставим агенту инструменты для доступа к реляционной базе данных (а также для получения информации о ее структуре, такой как имена доступных таблиц). Мы также дадим агенту инструмент, позволяющий выполнять поиск в Интернете (используя существующие поисковые системы в фоновом режиме). В совокупности мы получим агент, который может отправлять запросы к реляционной базе данных и использовать Интернет для получения информации, связанной с ее содержимым.

Начнем с более подробного описания инструментов, предоставляемых агенту. У него будет доступ к следующим пяти инструментам:

- `sql_db_list_tables` — перечисляет все таблицы в реляционной базе данных;
- `sql_db_schema` — возвращает SQL-схему таблицы по ее имени;
- `sql_db_query_checker` — позволяет агенту проверить SQL-запрос;
- `sql_db_query` — выполняет SQL-запрос и возвращает результат;
- `search` — позволяет агенту выполнять поиск в Интернете по ключевым словам, возвращая текст из сети.

Первые четыре инструмента помогают агенту получить доступ к реляционной базе данных. Последний инструмент позволяет ему извлекать информацию из Интернета. Получив задачу от пользователя, агент (используя базовую языковую модель) определяет, какие из этих инструментов вызывать и в каком порядке. Этот сценарий показан на рис. 10.3.

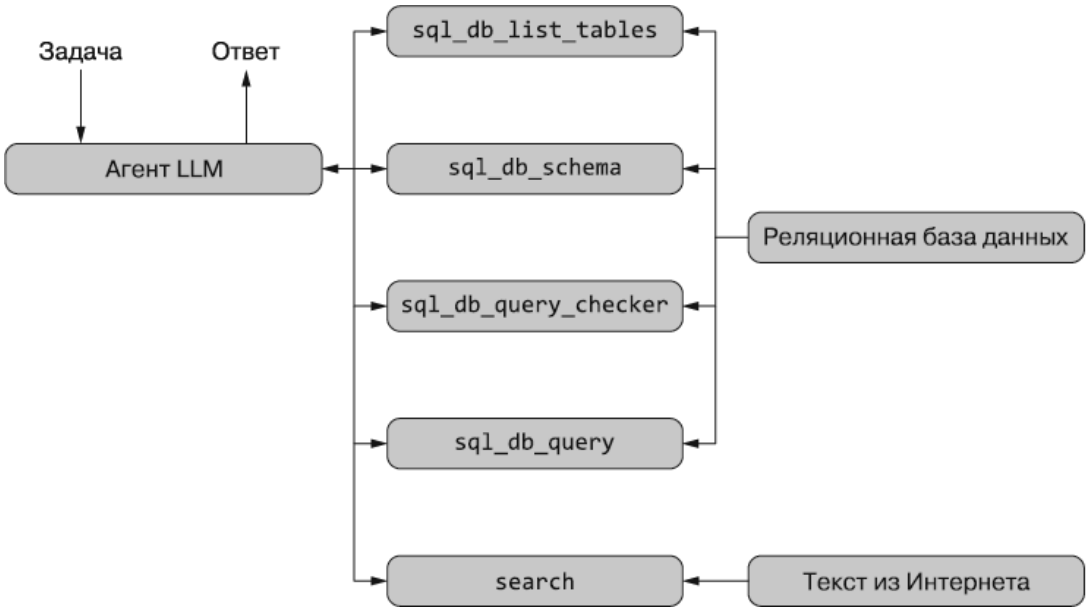


Рис. 10.3. Агент использует несколько инструментов для изучения структуры и выполнения запросов к реляционной базе данных. Кроме того, он может получать текст из Интернета с помощью инструмента веб-поиска

10.4.2. Создание агента с помощью фреймворка LangChain

LangChain позволяет быстро создавать агенты. И даже предлагает специализированные конструкторы для агентов, которые обращаются к структурированной базе данных. Мы используем эти возможности в коде ниже.

Агенты работают на базе языковых моделей. Для создания агента сначала необходимо создать объект модели:

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(
    temperature=0, model='gpt-4o')
```

Мы создадим чат-модель OpenAI. Точнее, опять обратимся к модели GPT-4o.

Затем мы создадим объект, представляющий реляционную базу данных. Запросы будут выполняться к базе данных SQLite,

хранящейся на диске. Предположим, что параметр `dbpath` содержит путь к соответствующему файлу базы данных (обычно такие файлы имеют расширение `.db`). Можно создать объект базы данных с помощью такого кода:

```
from langchain_community.utilities.sql_database import SQLDatabase
db = SQLDatabase.from_uri(f'sqlite:///{dbpath}')
```

Мы говорили о четырех инструментах для доступа к реляционной базе данных. К счастью, все они будут созданы автоматически из ее объекта. Однако нам еще нужно создать инструмент для веб-поиска.

Применим встроенный компонент LangChain — инструмент SerpAPI. Чтобы его использовать, сначала необходимо зарегистрировать учетную запись на сайте SerpAPI. Откройте браузер, перейдите по адресу **https://serpapi.com/**, нажмите кнопку **Register** (Регистрация) и создайте соответствующую учетную запись. Для выполнения приведенного ниже кода вам потребуется получить ключ доступа API (доступен на сайте **https://serpapi.com/dashboard**). Кроме того, необходимо установить расширение LangChain для активизации инструмента веб-поиска. Перейдите в терминал и выполните следующую команду:

```
pip install google-search-results==2.4.2
```

После этого достаточно следующего фрагмента кода на Python, чтобы создать инструмент для веб-поиска (при условии, что параметр `llm` содержит ранее созданный объект языковой модели, а `serpaikey` — ключ доступа SerpAPI):

```
from langchain.agents.load_tools import load_tools
extra_tools = load_tools(
```

```
            ['serpapi'],  serpapi_api_key=serpaikey,
llm=llm)
```

С помощью функции `load_tools` можно загрузить стандартные инструменты, передавая имена желаемых инструментов в качестве параметров. В данном случае нам нужен только инструмент веб-поиска, и мы передаем в списке имен инструментов (`serpapi`) только одну запись. После вызова функции `load_tools` мы сохраняем результат в параметре `extra_tools`: это список инструментов, содержащий одну запись (инструмент веб-поиска). Теперь у нас есть все компоненты, необходимые для создания агента с помощью фреймворка LangChain.

Предположим, что параметр `db` содержит созданный ранее объект базы данных, а `llm` — сгенерированную прежде языковую модель. С помощью кода ниже инициализируем агент, который обеспечивает доступ к данным на основе SQL:

```
from    langchain_community.agent_toolkits.sql.base
import create_sql_agent
agent = create_sql_agent(
    llm=llm, db=db, verbose=True,
    agent_type='openai-tools',
    extra_tools=extra_tools)
```

Команда `create_sql_agent` — это удобная функция, предлагаемая LangChain и позволяющая создавать агенты для доступа к данным на основе SQL. Четыре ранее упомянутых инструмента для доступа к реляционной базе данных (используемые для получения имен таблиц, показа схем таблиц, проверки SQL-запросов и в конечном счете их выполнения) добавляются автоматически, их не нужно указывать отдельно. Помимо этих ориентированных на SQL

инструментов, нам нужен еще только один — для выполнения веб-поиска. Подобные инструменты указываются в списке через входной параметр `extra_tools`. Установка параметра `verbose` в значение `True` позволяет следить за «ходом мыслей», приводящим агента к вызову конкретных инструментов (пример вывода будет показан ниже). Тип агента (в данном случае `openai-tools`) задает сразу два параметра: какой промпт отправлять модели и как именно разбирать ее ответ, чтобы понять, какой инструмент вызывать.

Создав агент, вызываем его для решения конкретной задачи, введя следующий код (предположим, что переменная `task` хранит описание задачи, которую требуется решить, на естественном языке):

```
agent.invoke({'input':task})
```

10.4.3. Полный рабочий код агента для анализа данных

В листинге 10.2 объединены все элементы, о которых мы говорили выше: считав ключ доступа API SerpAPI, путь к файлу базы данных и вопрос из командной строки, код создает объект языковой модели ❶, затем базу данных ❷, инструмент веб-поиска ❸ и, наконец, агент ❹. Он запускает агент ❺, передавая ему вопрос на обработку. Вывод, произведенный агентом, завершается ответом на этот вопрос (или сообщением об ошибке, если агенту не удалось найти ответ).

Листинг 10.2. Агент для анализа данных с возможностью веб-поиска

```
import argparse

from langchain.agents.load_tools import load_tools
```

```
from     langchain_community.utilities.sql_database
import SQLDatabase
from    langchain_community.agent_toolkits.sql.base
import create_sql_agent
from langchain_openai import ChatOpenAI

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
        parser.add_argument('serpaikey',  type=str,
help='Ключ доступа к SERP API')
          parser.add_argument('dbpath',   type=str,
help='Путь к базе данных SQLite')
         parser.add_argument('question',  type=str,
help='Вопрос, на который нужно ответить')
    args = parser.parse_args()

     llm = ChatOpenAI(                                  ❶
```

Создает клиент LLM

```
        temperature=0, model='gpt-4o')
```

❷ Создает объект базы данных

```
                                          db          =
SQLDatabase.from_uri(f'sqlite:///{args.dbpath}')
      extra_tools = load_tools(                        ❸
```

Добавляет инструмент веб-поиска

```
                                           ['serpapi'],
serpapi_api_key=args.serpaikey, llm=llm)
```

```
    agent = create_sql_agent(                                    ❹
```

Создает агент

```
        llm=llm, db=db, verbose=True,
        agent_type='openai-tools',
        extra_tools=extra_tools)
    agent.invoke({'input':args.question})                        ❺
```

Вызывает агент с помощью входных данных

10.4.4. Проверка в работе

Посмотрим, как код работает на практике. Скачайте код листинга 10.2, используя ссылку **Listing 2** на сайте книги. Помимо кода, для тестирования агента потребуется база данных SQLite. Мы будем использовать базу SQLite из главы 5, хранящую информацию о видеоиграх (файл доступен по ссылке **Games SQLite**, размещенной на сайте книги, в разделе дополнительных материалов к главе 5).

Откройте терминал и перейдите в каталог, содержащий код. Предположим, что файл базы данных, `games.db`, находится в том же каталоге. Запустите приведенный ниже код (заменив `[SerpAPI key]` своим ключом поиска, который доступен по ссылке **https://serpapi.com/dashboard**):

```
python listing2.py [SerpAPI key] games.db
⇨ 'What was the most sold game in 2016, and how is it played?'
```

Какая игра была самой продаваемой в 2016 году и как в нее играть?

Вывод будет похож на приведенный ниже (ваш вывод может немного отличаться из-за изменений веб-контента, небольших модификаций модели GPT-4o и некоторых других факторов):

```
[1m> Entering new SQL Agent Executor chain...[0m
[32;1m[1;3m
```

❶ Агент получает список таблиц

```
Invoking:         ``sql_db_list_tables`         with
`{'tool_input': "}`

[0m[38;5;200m[1;3mgames[0m[32;1m[1;3m
```

❷ Агент получает схему таблицы

```
Invoking:  `sql_db_schema`  with  `{'table_names':
'games'}`

[0m[33;1m[1;3m

CREATE TABLE games (
    rank INTEGER,
    name TEXT,
    platform TEXT,
    year INTEGER,
    genre TEXT,
    publisher TEXT,
    americasales NUMERIC,
    eusales NUMERIC,
    japansales NUMERIC,
    othersales NUMERIC,
    globalsales NUMERIC
)

/*

3 строки из таблицы игр:
```

```
rank    name    platform    year    genre    publis
her
americasales    eusales    japansales    othersales
    globalsales
1                                                  Wii
Sports    Wii    2006    Sports    Nintendo    41.4
900000000
29.0200000000    3.7700000000    8.4600000000    82
.7400000000
2     Super Mario Bros. NES 1985 Platform Nintendo
29.0800000000
3.5800000000    6.8100000000    0.7700000000    40.
2400000000
3                                    Mario        Kart
Wii    Wii    2008    Racing    Nintendo    15.8500
000000
12.8800000000    3.7900000000    3.3100000000    35
.8200000000
*/[0m[32;1m[1;3m
```

❸ Агент проверяет SQL-запрос

```
Invoking: `sql_db_query_checker` with `{'query':
'SELECT name
FROM games WHERE year = 2016 ORDER BY globalsales
DESC LIMIT 1'}`
responded: The games table contains the information
we need.
I will query for the game with the highest global
sales in 2016.

[0m[36;1m[1;3mSELECT name FROM games WHERE year =
2016
```

```
ORDER BY globalsales DESC LIMIT 1[0m[32;1m[1;3m
```

❹ **Агент запрашивает самую популярную игру**

```
Invoking: `sql_db_query` with `{'query': 'SELECT name
FROM games WHERE year = 2016 ORDER BY globalsales
DESC LIMIT 1'}`

[0m[36;1m[1;3m[('FIFA 17',)][0m[32;1m[1;3m
```

❺ **Агент ищет в Интернете информацию об игре FIFA 17**

```
Invoking: `Search` with `How to play FIFA 17`

[0m[33;1m[1;3m["A Beginner's Guide To Complete FIFA
17 Domination ... The
main steps you should take are to jump right in
with a quick play game. ...
EA Sports FIFA ...", 'Play FIFA 17 up to 5 days
before launch for a full
10 hours when you join EA Access on Xbox One and
Origin Access on PC.',
"1. Shield the ball in 360 degrees · 2. Use Driven
Shots and Driven Headers
.
3. Use set piece upgrades to score with style · 4.
Make Fifa 17's ...",
'Play FIFA 17 as much as you want with EA Access or
Origin Access
for only $4.99 per month. Now available in The
Vault.',
```

'Cautiously Start An Online Match. Score Early After Some Self-Proclaimed
Beautiful Build Up Play. Concede 4 Goals In A Row And Convince ...',
'FIFA 17 TUTORIALS & ULTIMATE TEAM ☒
Twitter: https://twitter.com/KrasiFIFA ☒
Instagram: http://instagram.com/KrasiFIFA How I record my ...',
"Draft mode is another way to play FIFA Ultimate Team,
giving you the ability to play with Players you don't own.
You'll have the opportunity to draft a random ..."]

❻ **Агент формулирует окончательный ответ**

[0m[32;1m[1;3mThe most sold game in 2016 was FIFA 17.

To play FIFA 17, you can follow these steps:

1. Jump right in with a quick play game.
2. Shield the ball in 360 degrees.
3. Use Driven Shots and Driven Headers.
4. Use set piece upgrades to score with style.
5. Start an online match cautiously.
6. Score early after some self-proclaimed beautiful build-up play.
7. Draft mode is another way to play FIFA Ultimate Team,
giving you the ability to play with players you don't own.

```
You'll have the opportunity to draft a random team.

Remember, practice makes perfect![0m

[1m> Finished chain.[0m
```

Напомню, что мы включили в агенте подробный режим вывода. Это означает, что в выводе содержится полная запись того, какие инструменты вызывал агент, а также процесс его рассуждений. Внимательнее изучим вывод, чтобы понять, что произошло.

Сначала агент получает список таблиц, доступных в реляционной базе данных (используя инструмент `sql_db_list_tables`) ❶. Определенно, это разумное действие при знакомстве с новой базой. Результат вызова инструмента показывает, что база содержит только одну таблицу (`games`). Агенту «любопытно» ее содержимое. Он вызывает инструмент `sql_db_schema`, чтобы получить дополнительную информацию о таблице `games` ❷. Обратите внимание, что этому инструменту нужны входные параметры, в частности имя исследуемой таблицы. В журнале показаны значения всех входных параметров для каждого вызова инструмента.

Вызов инструмента `sql_db_schema` возвращает SQL-команду, которая использовалась для создания таблицы `games`, вместе с небольшим фрагментом содержимого таблицы. Далее агент изучает SQL-запрос для получения соответствующей информации о входном вопросе о том, какая игра была самой продаваемой в 2016 году и как в нее играть. На первом этапе он проверяет, корректен ли запрос с точки зрения синтаксиса, вызывая инструмент `sql_db_query_checker` ❸:

```
SELECT name FROM games WHERE year = 2016 ORDER BY
globalsales DESC LIMIT 1
```

В то же время агент использует возможность «поразмышлять» о полезности рассматриваемого запроса, о чем свидетельствует вывод, в котором говорится, что в таблице `games` содержится нужная информация и инструмент выполнит запрос, чтобы найти игру с самыми высокими мировыми продажами в 2016 году. Может показаться странным, что языковая модель умеет извлекать пользу из такого рода монолога, вместо того чтобы просто напрямую вызвать нужный инструмент. Тем не менее было показано, что предоставление агентам возможности явно рассуждать о стоящей перед ними проблеме и действиях, которые они совершают для ее решения, может повысить их производительность [1]. То же самое происходит и здесь.

Далее агент решает использовать ранее проверенный запрос для получения информации из базы данных, используя инструмент `sql_db_query` ❹. SQL-запрос возвращает название игры, которая принесла наибольшую выручку в 2016 году: FIFA 17, симулятор игры в футбол компании Electronic Arts. Но вопрос во входных данных этим не ограничивался: «Какая игра была самой продаваемой в 2016 году и *как в нее играть*?» Ответ на вторую его часть не удастся получить из контента базы данных. Надо отдать должное агенту — он это понимает и пытается обратиться к Интернету: отправляет поисковый запрос с помощью инструмента `Search` для поисковой строки «Как играть в FIFA 17» ❺. Обратите внимание, что агент сумел автоматически сформулировать подходящую поисковую строку на основе результата SQL-запроса и входного вопроса. Результатом веб-поиска является коллекция текстовых фрагментов (показанных в выводе), которые содержат информацию о том, как играть в FIFA 17.

Наконец, агент использует информацию, полученную в результате веб-поиска (вместе с информацией из SQL-базы данных), чтобы сформулировать окончательный ответ ❻. В нем FIFA 17 называется самой популярной игрой в 2016 году и содержатся подробные инструкции, как в нее успешно играть. Мы видели, что агент может выполнять сложную последовательность вызовов инструментов, чтобы найти желаемый ответ, и для этого не нужно вручную прописывать необходимые этапы процесса. При желании вы можете задать агенту еще несколько, возможно, более сложных вопросов и посмотреть, сможет ли он ответить на них.

10.5. Добавление пользовательских инструментов

До сих пор мы использовали стандартные инструменты, предлагаемые LangChain, для наиболее распространенных сценариев использования. Но что произойдет, если появятся специализированные требования? Например, потребуется сделать так, чтобы источник данных был доступен через пользовательский API, или у вас есть специализированные аналитические функции, которые агент может применять к данным. В таких случаях можно определить собственные пользовательские инструменты и сделать их доступными для агента LangChain.

10.5.1. Конвертер валют

В предыдущем разделе мы анализировали набор данных о продажах видеоигр. Исходные данные сообщают о значениях продаж в долларах США. А сколько это в других валютах? Чтобы агенты могли анализировать продажи игр, используя несколько валют, мы добавим инструмент конвертации. Получив на вход

сумму в долларах США и название целевой валюты, этот инструмент возвращает значение, эквивалентное этой валюте.

В листинге 10.3 показано, как добавить инструмент конвертации валют к агенту данных.

Листинг 10.3. Агент анализа данных с инструментом конвертации валют

```
import argparse

from langchain.agents.load_tools import load_tools
from langchain.tools import tool
from langchain_community.utilities.sql_database import SQLDatabase
from langchain_community.agent_toolkits.sql.base import create_sql_agent
from langchain_openai import ChatOpenAI
from typing import Union

@tool     ❶ Превращает функцию в инструмент

❷ Сигнатура функции и типов параметров

def convert_currency(USD_amount: float, currency: str) -> Union[float, str]:

    ❸ Документация функции

    """ Конвертирует сумму в долларах США в другую валюту.

    Аргументы:
        USD_amount: сумма в долларах США.
        currency: название целевой валюты.
```

```
    Выводит:
        Ввести сумму в целевой валюте.
    """
    conversion_factors = {
        'Euro':0.93, 'Yen':151.28, 'Yun':0.14,
        'Pound':1.26, 'Won':0.00074, 'Rupee':0.012}

    if currency not in conversion_factors:
```

❹ Полезное сообщение об ошибке для агента

```
        error_message = (
            f'Неизвестная валюта: {currency}!'
                            f'Используйте  одну  из
{conversion_factors.keys()}')
        return error_message
```

❺ Конвертирует и возвращает результат

```
                            conversion_factor       =
conversion_factors[currency]
              converted_amount    =    USD_amount    *
conversion_factor
    return converted_amount

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
         parser.add_argument('serpaikey',  type=str,
help='Ключ доступа к SERP API')
           parser.add_argument('dbpath',   type=str,
help='Путь к базе данных SQLite')
```

```
        parser.add_argument('question',  type=str,
help='Вопрос, на который нужно ответить')
    args = parser.parse_args()

    llm = ChatOpenAI(
        temperature=0, model='gpt-4o')
                                        db          =
SQLDatabase.from_uri(f'sqlite:///{args.dbpath}')
    extra_tools = load_tools(
                                         ['serpapi'],
serpapi_api_key=args.serpaikey, llm=llm)
```

❻ Добавляет инструмент конвертации валют

```
    extra_tools.append(convert_currency)

    agent = create_sql_agent(
        llm=llm, db=db, verbose=True,
        agent_type='openai-tools',
        extra_tools=extra_tools)
    agent.invoke({'input':args.question})
```

По сути, инструмент — это не что иное, как функция Python. Наш конвертер валют запускается функцией `convert_currency` ❷. Как фреймворк LangChain узнает, что следует превратить функцию в инструмент? С помощью декоратора `@tool` ❶, который должен находиться непосредственно перед именем функции. Обычно для функций Python не требуется указывать типы параметров и возвращаемых значений (хотя это улучшает читаемость кода). Однако если вы планируете превратить функцию в инструмент, то все эти типы нужно обязательно прописать. Причина такова: чтобы правильно использовать функцию в качестве инструмента, агент должен

вызывать ее с параметрами корректного типа. Все типы, которые указываются в заголовке функции, будут доступны агенту как часть описания инструмента. Следовательно, сопоставление параметров с типами поможет агенту избежать ненужных ошибок вызова.

Помимо типов параметров, агент должен иметь информацию о том, что может делать ваш инструмент. Первая важная информация — имя функции. По умолчанию инструмент будет назван в соответствии с ней. Не называйте функцию `XYZ`, ведь это сильно затруднит понимание выполняемого действия! Имя функции, приведенной в листинге 10.3, `convert_currency`, должно вполне четко указывать, что она делает. Аналогичным образом имена входных параметров, `USD_amount` (типа `float`) и `currency` (типа `` `str` ``), говорят сами за себя (и это хорошо!). Вывод функции — это сумма, конвертированная в целевой валюте, или сообщение об ошибке, если запрошенная валюта не поддерживается (поэтому тип вывода — это `Union[float, str]` — либо число, либо строка). Простое правило: если вы собираетесь использовать функцию Python как инструмент, то оформляйте ее так, чтобы другой разработчик-человек мог понять, что она делает, даже не заглядывая в реализацию.

Помимо имени и параметров функции, агент «видит» ее документацию ❸. Опять же, убедитесь, что ваша документация хорошо структурирована, а также объясните семантику своего инструмента и связанных параметров. В данном случае документация описывает назначение инструмента, семантику входных параметров (даже с примером допустимого значения для второго параметра) и семантику вывода.

Функция `convert_currency` использует небольшую базу данных валют с соответствующими коэффициентами конвертации. Например, она содержит коэффициенты

конвертации для евро и иен, но в ней нет многих других валют. Создавая инструмент для своего агента, учитывайте случаи, когда агент будет использовать инструмент некорректно. Это может произойти, если описание инструмента неполное или если языковая модель допускает ошибку (что бывает даже с современными моделями). В этом случае мы добавляем специальную обработку для ситуации, когда целевая валюта не поддерживается (то есть отсутствует соответствующий коэффициент конвертации) ❹ . Если целевая валюта не поддерживается, то функция возвращает информативное сообщение об ошибке, содержащее полный список доступных валют. Это помогает агенту ограничить параметр набором допустимых вариантов для последующих вызовов. Если целевая валюта поддерживается, то функция возвращает конвертированную сумму ❺.

Создав инструмент на основе функции Python, мы просто должны сделать его доступным для агента. В листинге 10.3 показывается создание практически такого же агента, как в листинге 10.2, с той лишь разницей, что мы добавляем инструмент конвертации валют ❻. Мы снова используем SQL-агент, поэтому инструменты конвертера и веб-поиска добавлены в список дополнительных инструментов (добавляемых поверх стандартных для доступа к SQL, которые автоматически предоставляются агенту). По умолчанию имя инструмента совпадает с именем функции, на основе которой он создан. Следовательно, мы просто добавляем `convert_currency` в список дополнительных инструментов ❻, чтобы наделить агент возможностью конвертировать валюты.

10.5.2. Проверка в работе

Посмотрим, сможет ли наш агент использовать новый инструмент. Скачайте код листинга 10.3, доступный по ссылке

Listing 3, размещенной на сайте книги в разделе дополнительных материалов к главе. Можно взять тот же файл базы данных, что и раньше (предполагается, что файл `games.db` находится в одной папке с кодом). Затем откройте терминал и введите следующий код (вставив вместо `[SerpAPI key]` свой ключ доступа SerpAPI):

```
python listing3.py [SerpAPI key] games.db    'What revenue was generated by
computer games in 2015? How much is it in Yen?'
```

Какой доход был получен от компьютерных игр в 2015 году? Сколько это в йенах?

Очевидно, что для ответа на этот вопрос требуется инструмент конвертации валют. При запуске кода отобразится вывод, похожий на этот:

```
[1m> Entering new SQL Agent Executor chain...[0m
[32;1m[1;3m
```

❶ Агент получает таблицы базы данных

```
Invoking: `sql_db_list_tables` with `''tool_input':
"`


[0m[38;5;200m[1;3mgames[0m[32;1m[1;3m
```

❷ Агент запрашивает схему таблицы

```
Invoking: `sql_db_schema` with `''table_names':
'games'`


[0m[33;1m[1;3m
CREATE TABLE games (
    rank INTEGER,
    name TEXT,
    platform TEXT,
```

```
    year INTEGER,
    genre TEXT,
    publisher TEXT,
    americasales NUMERIC,
    eusales NUMERIC,
    japansales NUMERIC,
    othersales NUMERIC,
    globalsales NUMERIC
)

/*
3 строки из таблицы игр:
rank    name    platform    year    genre    publis
her    americasales
eusales    japansales    othersales    globalsales
1                                                Wii
Sports    Wii    2006    Sports    Nintendo    41.4
900000000
29.0200000000    3.7700000000    8.4600000000    82
.7400000000
2    Super Mario Bros.    NES    1985    Platform
Nintendo    29.0800000000
3.5800000000    6.8100000000    0.7700000000    40.
2400000000
3    Mario Kart Wii    Wii    2008    Racing
Nintendo    15.8500000000
12.8800000000    3.7900000000    3.3100000000    35
.8200000000
*/[0m[32;1m[1;3m
```

❸ Агент проверяет SQL-запрос

```
Invoking: `sql_db_query_checker` with `{'query':
'SELECT SUM(globalsales) as
total_revenue FROM games WHERE year = 2015'}`
responded: The "games" table contains the
information we need.
The "globalsales" column represents the global
revenue generated by
each game. We can sum this column for the games
released in 2015 to
get the total revenue. Let's write and check the
SQL query.

[0m[36;1m[1;3mSELECT      SUM(globalsales)      as
total_revenue
FROM games WHERE year = 2015[0m[32;1m[1;3m
```

❹ **Агент запрашивает продажи за 2015 год**

```
Invoking: `sql_db_query` with `{'query': 'SELECT
SUM(globalsales) as
total_revenue FROM games WHERE year = 2015'}`

[0m[36;1m[1;3m[(264.43999999999795,)][0m[32;1m[1;3m
```

❺ **Агент конвертирует валюты**

```
Invoking: `convert_currency` with `{'USD_amount':
264.43999999999795, 'currency': 'Yen'}`

[0m[38;5;200m[1;3m40004.48319999969[0m[32;1m[1;3m
```

❻ **Агент формулирует окончательный ответ**

```
The total revenue generated by computer games in 2015
was approximately $264.44 million.
In Japanese Yen, this is approximately ¥40,004,483,200.[0m

[1m> Finished chain.[0m
```

Аналогично предыдущему примеру агент сначала исследует базу данных, получая набор таблиц ❶, а затем, узнав, что она содержит только одну таблицу, получает схему для нее ❷. Агент делает верный вывод, что база содержит полезную информацию для входного вопроса, и сначала проверяет ❸, а затем выполняет ❹ соответствующий SQL-запрос. В результате возвращается общая стоимость продаж компьютерных игр в 2015 году, выраженная в долларах США. Чтобы ответить на вторую часть вопроса о стоимости в иенах, агент затем применяет инструмент конвертации валют ❺ . Обратите внимание, что он выбирает соответствующие значения двух входных параметров на основе описания функции и типов. Наконец, агент формулирует ответ на входной вопрос ❻.

10.6. Индексация мультимодальных данных с помощью LlamaIndex

LangChain вовсе не единственный фреймворк, способный упростить использование языковых моделей для анализа данных. В этом разделе мы обсудим еще один фреймворк, который появился недавно и быстро набирает популярность, — LlamaIndex.

10.6.1. Обзор

LlamaIndex в полной мере раскрывает свои возможности в тех сценариях, где языковым моделям необходимо обращаться к большим хранилищам данных, в том числе разноформатных. В таких случаях, как правило, нецелесообразно (или даже невозможно) напрямую загружать все данные в модель. Вместо этого нужен механизм, который быстро идентифицирует релевантные данные для конкретной задачи, передавая модели только их. Как следует из названия, LlamaIndex индексирует данные, чтобы быстро идентифицировать соответствующие подмножества. Точнее, платформа связывает фрагменты данных (например, части текста) с эмбеддингами. Мы кратко обсуждали такие векторы в главе 4. В двух словах: эмбеддинг представляет семантику текста в виде вектора, вычисляемого языковой моделью. Если у двух документов векторы похожи (расстояние между ними маленькое), то мы предполагаем, что они относятся к схожим темам.

Типичный конвейер обработки данных LlamaIndex состоит из следующих этапов. Сначала он загружает данные (возможно, в различных форматах) и выполняет предварительную обработку. Например, такая обработка может заключаться в разделении длинных текстовых документов на более мелкие фрагменты, с которыми удобнее работать. Далее LlamaIndex индексирует данные. Как я уже говорил, это означает сопоставление фрагментов данных с эмбеддингами. По умолчанию для вычисления эмбеддингов LlamaIndex использует небольшие языковые модели (например, ada от OpenAI). Благодаря этому этап индексации обходится недорого. Более того, LlamaIndex может сохранять сгенерированный индекс (эмбеддинги)[9] на диск, чтобы их не приходилось генерировать повторно для каждой новой задачи.

Фреймворк предлагает поддержку различных сценариев использования на основе сгенерированного индекса.

Например, он может использовать проиндексированные данные для ответов на вопросы на естественном языке. Получив вопрос в качестве входных данных, LlamaIndex сначала вычисляет эмбеддинг для текста вопроса. Затем сравнивает вектор, представляющий вопрос, с предварительно вычисленными векторами, представляющими фрагменты данных. Фреймворк выявляет элементы данных с наиболее похожими векторами. Связанные данные добавляются в промпт вместе с входным вопросом. Цель — сгенерировать ответ на вопрос, используя релевантные данные в качестве контекста. Для индексации используются маленькие модели, однако для генерации окончательного ответа обычно применяются модели большего размера. На рис. 10.4 показан такой конвейер обработки данных.

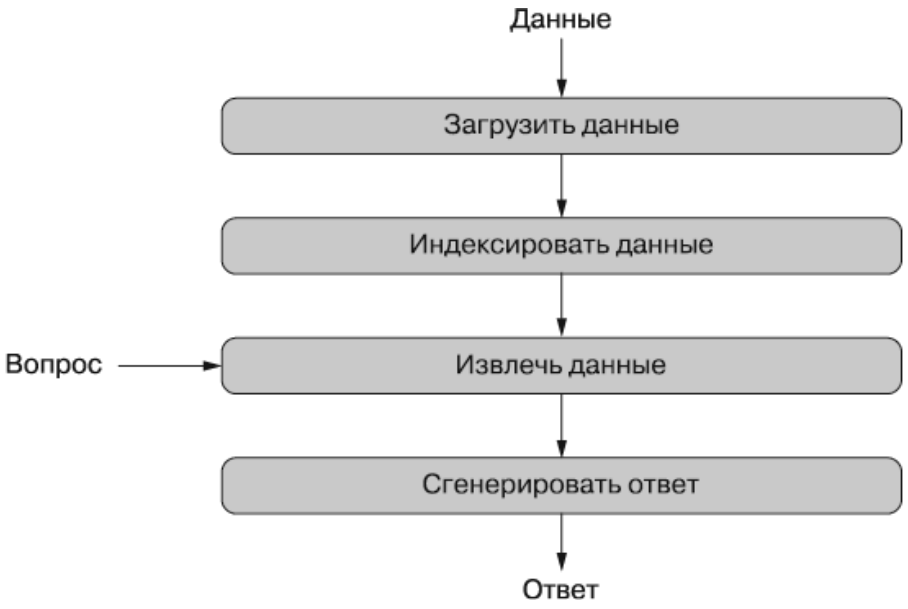


Рис. 10.4. Основные этапы типичного конвейера обработки данных LlamaIndex. Фреймворк загружает и индексирует данные, чтобы обеспечить быстрое извлечение. Получив вопрос, он идентифицирует релевантные элементы данных и передает их вместе с входным вопросом языковой модели для генерации ответа

10.6.2. Установка LlamaIndex

Давайте реализуем конвейер, описанный в предыдущем подразделе, на языке Python. Для работы с LlamaIndex сначала необходимо установить несколько пакетов. Перейдите в терминал и выполните следующую команду:

```
pip install llama-index==0.10.25
```

Она установит основные пакеты LlamaIndex. Однако вы будете использовать платформу для анализа множества разных форматов данных. Чтобы LlamaIndex мог корректно обращаться ко всем этим форматам и анализировать их, вы должны установить несколько дополнительных пакетов. В терминале введите следующие команды:

```
pip install torch==2.1.2
pip install transformers==4.36.0
pip install python-pptx==0.6.23
pip install Pillow==10.2.0
```

Эти библиотеки необходимы для анализа документов в формате `.pdf` и презентаций PowerPoint — и те и другие понадобятся для следующего проекта.

10.6.3. Реализация простой системы вопросов и ответов

Вы все еще работаете в компании Banana и столкнулись со сложной проблемой: так как компания международная, у нее есть множество филиалов. Ваш руководитель хочет, чтобы вы проанализировали данные из разных филиалов, например, в целях сравнения эффективности их работы. Однако разные филиалы используют совершенно разные форматы данных. Одни публикуют свои результаты в виде простых текстовых документов, тогда как другие регулярно выпускают подробные

презентации PowerPoint. Как объединить все эти форматы? К счастью, LlamaIndex упрощает эту задачу.

Ее можно решить с помощью всего нескольких строк кода на Python (листинг 10.4).

Листинг 10.4. Простая система вопросов и ответов с помощью фреймворка LlamaIndex

```
import argparse
import openai

from llama_index.core import VectorStoreIndex,
SimpleDirectoryReader

if __name__ == '__main__':

    ❶ Парсирует параметры командной строки

    parser = argparse.ArgumentParser()
          parser.add_argument('datadir', type=str,
help='Путь к папке с данными')
         parser.add_argument('question', type=str,
help='Вопрос, на который нужно ответить')
    args = parser.parse_args()

    ❷ Загружает данные из каталога

                                    documents =
SimpleDirectoryReader(args.datadir).load_data()

    ❸ Индексирует данные

                                       index =
VectorStoreIndex.from_documents(documents)
```

❹ Включает запросы по индексу

```
engine = index.as_query_engine()
```

❺ Генерирует ответ

```
answer = engine.query(args.question)
print(answer)
```

Код принимает следующие входные параметры.

- Ссылку на репозиторий данных. Этот репозиторий может содержать файлы различных типов.
- Вопрос, на который нужно ответить. LlamaIndex будет искать ответ на него среди данных в репозитории.

После получения этих параметров из командной строки ❶ загружаются данные из входного репозитория ❷. К счастью, в LlamaIndex это сделать очень просто: не нужно добавлять обработку разных типов файлов и т.п. Вместо этого достаточно передать путь к каталогу. Далее только что загруженные данные индексируются ❸ . По умолчанию для вычисления эмбеддингов LlamaIndex использует модели ada компании OpenAI. Преобразование данных и разделение на фрагменты (например, разбиение больших текстовых документов на части, достаточно мелкие для обработки моделями ada от OpenAI) — все это выполняется автоматически. Далее на основе индекса создается механизм, который будет отвечать на запросы ❹ . Он будет автоматически извлекать данные, связанные с входным вопросом, используя индекс. Наконец, применяется ранее сгенерированный механизм, чтобы ответить на входной вопрос ❺, и выводится результат.

Несмотря на то что фреймворк LlamaIndex предлагает различные способы настройки и специализации каждого шага данного конвейера (и создания других конвейеров), именно заданные по умолчанию параметры на каждом шаге позволяют добиться предельной краткости кода.

10.6.4. Проверка в работе

Протестируем наш конвейер на демонстрационных данных. Вы можете скачать код листинга 10.4, используя ссылку **Listing 4** на сайте книги. Кроме того, скачайте файл `bananareports.zip` (доступен по ссылке **Banana Reports**, размещенной там же на сайте, в разделе дополнительных материалов к этой главе) и распакуйте его в той же папке, что и код. В папке вы найдете (короткие) бизнес-отчеты в форматах текстовых, PDF-документов и презентаций PowerPoint. Пора ответить на несколько вопросов! Откройте терминал и перейдите в каталог, содержащий код и папку `bananareports` (после распаковки). Теперь выполните следующую команду, которая задает вопрос о прибыли филиала Plantain за 2023 год:

```
python listing4.py bananareports  'Сколько филиал Plantain заработал в 2023 году?'
```

Должен отобразиться примерно такой вывод:

```
За 2023 год филиал Plantain заработал 30 миллионов долларов США.
```

Просмотрите файл. Вы увидите, что филиал Plantain действительно заработал 30 миллионов долларов США. Спросите модель о других филиалах (Pisang Raja и Cavendish); должны отобразиться корректные результаты. Можно задавать более сложные вопросы (например, «Какой филиал принес

наибольшую прибыль в 2023 году?»), для ответа на которые требуется сравнить данные разных файлов. Опять же, система должна быть способна выдать правильный ответ.

10.7. Заключительные замечания

Итак, LangChain и LlamaIndex — два популярных фреймворка для работы с языковыми моделями. Мы убедились, что реализация даже сложных приложений, например на основе агентов, при использовании этих фреймворков выполняется довольно быстро. LangChain и LlamaIndex имеют схожую функциональность: к примеру, оба фреймворка предоставляют поддержку для реализации агентов на основе языковых моделей. LlamaIndex особенно эффективен в сценариях, где используется общий шаблон, описанный в предыдущем разделе (предоставление функциональности на основе индексов данных). LangChain имеет более общую направленность, помогая разработчикам создавать сложные приложения с использованием языковых моделей.

На момент написания книги оба фреймворка были относительно новыми, поэтому описанные выше характеристики могут не соответствовать будущим версиям. Кроме того, интерфейсы фреймворков быстро развиваются, и для работы с новейшими версиями, возможно, придется внести изменения в код. Если ваше приложение не требует сложной логики, то можете использовать низкоуровневые библиотеки наподобие библиотек компании OpenAI.

Конечно, в этой главе не приводилось подробное описание фреймворков. Я хотел показать их возможности, чтобы вы могли осознанно выбрать, какой из них стоит изучить более глубоко.

Резюме

- Фреймворки LangChain и LlamaIndex позволяют создавать сложные приложения на основе языковых моделей.
- LangChain и LlamaIndex упрощают процесс создания агентов. Агенты используют языковые модели для управления вызовами различных инструментов (стандартных, а также пользовательских). Они могут решать сложные задачи при наличии доступа к подходящим инструментам.
- В LangChain для превращения функций в инструменты следует использовать декоратор `@tool`.
- LlamaIndex индексирует различные типы данных, создавая эмбеддинги.
- LlamaIndex упрощает исследование и запрос проиндексированных данных.

Литература

1. *Wei J., Wang X., Schuurmans D. et al.* Chain-of-Thought Prompting Elicits Reasoning in Large Language Models // Advances in Neural Information Processing Systems, 2022. — № 35. — Pp. — 24 824–24 837.

---

9 Это не просто эмбеддинги. LlamaIndex строит индекс из набора данных, их эмбеддингов и дополнительных метаданных, которые используются для быстрого поиска. — *Примеч. науч. ред.*

Read IT Club

Комьюнити рецензентов
и переводчиков ИТ-литературы

Миссия участников клуба — обеспечить высокое качество профессиональной переводной литературы в России. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.

присоединиться к нам

Анализ данных с помощью GenAI и Python

Артур Гуджа, Марлена Сивяк, Мариан Сивяк

Анализ данных

с помощью GenAI и Python

Артур Гуджа • Марлена Сивяк • Мариан Сивяк



MANNING

Эта книга — практическое руководство для тех, кто превращает данные в решения. Шаг за шагом вы разберетесь, как современные методы искусственного интеллекта помогают не просто интерпретировать массивы информации, но и находить скрытые закономерности, прогнозировать будущее и создавать конкурентные преимущества для бизнеса. Вы получите системное понимание инструментов анализа, узнаете о лучших практиках применения машинного обучения и научитесь интегрировать ChatGPT в ежедневные рабочие процессы. Четкие инструкции научат правильно извлекать, интерпретировать и представлять данные, в результате вы сможете поднять навыки на совершенно новый уровень. Это не сухая теория — это рабочая методика, которая помогает аналитикам принимать решения быстрее, точнее и смелее. Вы получите проверенные стратегии анализа больших данных с помощью ИИ, практические инструменты для повышения точности прогнозов, кейсы из бизнеса и индустрии, которые можно адаптировать под свои задачи, понимание, как превратить аналитическую работу в драйвер роста компании.

КУПИТЬ

Машинное обучение на табличных данных: XGBoost, глубокое обучение и ИИ

Марк Райан, Лука Массарон

XGBoost, глубокое обучение и ИИ

Машинное обучение на табличных данных



Марк Райан
Лука Массарон

Предисловие Антонио Гулли

MANNING

Машинное обучение может ускорить выполнение повседневных бизнес-задач, таких как сверка счетов, прогнозирование спроса или автоматизация обслуживания клиентов, не говоря уже о более сложных задачах, например выявление мошенничества, прогнозное техническое обслуживание и персонализированный маркетинг. Извлекайте важную информацию из электронных таблиц, реестров, баз данных и других источников, используя градиентный бустинг, глубокое обучение и генеративный ИИ. «Машинное обучение на табличных данных» описывает практические методы машинного обучения для модернизации каждого этапа анализа бизнес-данных. Вас ждут примеры использования XGBoost и Keras для прогнозирования цен на краткосрочную аренду недвижимости, развертывание локальной модели машинного обучения с помощью Python и Flask и оптимизация рабочих процессов с использованием больших языковых моделей (LLM). Научитесь делать свои модели более мощными и интерпретируемыми.

КУПИТЬ

Строим LLM с нуля

Себастьян Рашка

С НУЛЯ

СТРОИМ LLM

Себастьян Рашка



MANNING

Говорят, что физик Ричард Фейнман однажды сказал: «Я не понимаю того, что не могу создать». Основываясь на этом же важном принципе, автор бестселлеров Себастьян Рашка шаг за шагом ведет вас к созданию LLM в стиле GPT, которую вы сможете запустить на своем ноутбуке. Это увлекательная книга, которая охватывает каждый этап процесса: от планирования и кодирования до обучения и тонкой настройки. «Строим LLM с нуля» — это чрезвычайно занимательное путешествие в основы генеративного ИИ. Не полагаясь на существующие библиотеки LLM, вы реализуете в коде базовую модель, превратите ее в классификатор текста и в конечном счете создадите чат-бот, который сможет следовать вашим инструкциям в диалоге. И вы действительно поймете LLM, потому что создали ее сами!

КУПИТЬ

Data Science в действии

Леонард Апельцин

реальные проекты на Python

Data Science в действии



Леонард Апельцин

MANNING

В проектах обработки и анализа данных много движущихся частей, и требуются практика и знания, чтобы создать гармоничную комбинацию кода, алгоритмов, наборов данных, форматов и визуальных представлений. Эта уникальная книга содержит описание пяти практических проектов, включая отслеживание вспышек заболеваний по заголовкам новостей, анализ социальных сетей и поиск закономерностей в данных о переходах по рекламным объявлениям. Автор не ограничивается поверхностным обсуждением теории и искусственными примерами. Исследуя представленные проекты, вы узнаете, как устранять распространенные проблемы, такие как отсутствующие и искаженные данные и алгоритмы, не соответствующие создаваемой модели. По достоинству оцените подробные инструкции по настройке и детальные обсуждения решений, в которых описываются типичные точки отказа, и обретите уверенность в своих навыках.

КУПИТЬ