Аймен Эль Амри

GPT-3: программирование на Python в примерах

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

OpenAI GPT For Python Developers

Aymen El Amri

Leanpub

GPT-3: программирование на Python в примерах

Аймен Эль Амри

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2023

УДК 004.04
ББК 32.371
А62

А62 Аймен Эль Амри
GPT-3: программирование на Python в примерах / пер. с англ. В. Яценкова. – М.: ДМК Пресс, 2023. – 218 с.: ил.

ISBN 978-5-93700-221-1

В книге рассказывается о том, как использовать генеративную текстовую модель (GPT) для создания приложений различного назначения, в числе которых медицинский чат-бот с пользовательской точной настройкой, интеллектуальный голосовой помощник, система предсказания категории новостей и многие другие. Вы узнаете, как управлять уровнем креативности моделей GPT, применять современные методы генерирования высококачественного текста, классифицировать изображения с помощью OpenAI CLIP. Примеры и практические упражнения помогут закрепить пройденный материал.

Издание предназначено для тех, кто владеет основами языка программирования Python и собирается использовать GPTв реальных сценариях для решения прикладных задач.

УДК 004.04
ББК 32.371

ISBN 978-5-93700-221-1 (рус.)

Оглавление

Предисловие

Когда люди спрашивают меня, чем я занимаюсь, мне всегда сложно дать им простой ответ. Моя карьера продвигалась извилистыми путями, и за эти годы я примерил много разных профессий. Я человек, который страстно любит учиться и пробовать что-то новое, оттого мне и довелось поработать в разных областях.

Я занимался разработкой программного обеспечения, рекламой, маркетингом, сетями и телекоммуникациями, системным администрированием, преподаванием, написанием технических текстов, ремонтом компьютеров и многими другими делами. Мной всегда двигало желание узнать больше и расширить свой кругозор. По мере того как я изучал новые технологии, знакомился с новыми людьми и исследовал новые концепции, мой разум становился более открытым, а кругозор расширялся. Я начал видеть связи и возможности, которых раньше не замечал.

Чем больше я узнавал сам, тем больше мне хотелось учить других, иногда даже бесплатно. Мне нравится это особое чувство, когда видишь, как в чьих-то глазах зажигается огонек понимания. Я всегда был учителем в душе, и мне всегда нравилось делиться своими знаниями с другими.

Именно эти соображения побудили меня написать руководство по работе с большими моделями Open AI.

Во время работы над этой книгой я постоянно думал о людях, которые будут ее читать. Я стремился создать доступное и простое руководство по NLP, GPT и смежным темам для людей, знающих Python, но владеющих ограниченными знаниями в этих областях. Моя цель состояла в том, чтобы предоставить практическую информацию, которую читатели смогут использовать для создания своих собственных интеллектуальных систем, не тратя многие часы на изучение теории, лежащей в основе этих концепций.

В этом практическом руководстве я делюсь своими знаниями и опытом работы с моделями OpenAI, в частности GPT-3 (но также и другими моделями), и тем, как программисты Python могут использовать их для создания собственных интеллектуальных приложений. Книга выстроена как пошаговое руководство, которое охватывает основные понятия и методы использования GPT-3 и дает читателю прочные и разнообразные практические навыки.

Мой почтовый ящик всегда полон, и я получаю много писем. Но наибольшее удовольствие мне доставляют письма с вопросами и пожеланиями от людей, которые прочитали мои онлайн-руководства и курсы и нашли их полезными. Пожалуйста, в любое время обращайтесь ко мне по адресу **aymen@faun.dev**. Мне важно узнать ваше мнение.

Надеюсь, вы получите от чтения этой книги такое же удовольствие, какое я получил от работы над ней.

Об авторе

Аймен Эль Амри – писатель, предприниматель, преподаватель и инженер-программист, который преуспел в различных профессиях в области информационных технологий, включая DevOps и Cloud Native, облачную архитектуру, Python, NLP, науку о данных и многое другое.

Аймен лично обучил сотни программистов и написал множество книг и курсов, которые прочитали тысячи других инженеров и разработчиков.

У Аймена Эль Амри практичный подход к обучению, который всегда находит отклик у его аудитории. Он разделяет сложные понятия на составные части, рассказывает о них понятным языком и иллюстрирует реальными примерами.

Он основал несколько проектов: FAUN[1], eralabs[2] и Marketto[3]. Вы можете найти Аймена в Twitter[4] и Linkedin[5].

---

1 **https://faun.dev.**

2 **https://eralabs.io.**

3 **https://marketto.dev.**

4 **https://twitter.com/@eon01.**

5 **https://www.linkedin.com/in/elamriaymen/.**

Об этой книге

Знания, которые вы получите из этой книги, относятся к текущей версии (GPT-3) и, вероятно, также пригодятся для GPT-4, если нам суждено дождаться ее выпуска[1].

OpenAI предоставляет API (интерфейс прикладного программирования) для доступа к моделям ИИ. Назначение API – абстрагировать базовые модели путем создания универсального интерфейса для всех версий, позволяющего пользователям использовать GPT независимо от его версии.

Цель этого книги – предоставить пошаговое руководство по использованию GPT-3 в ваших проектах с помощью API OpenAI. В руководстве рассмотрены и другие модели, такие как CLIP, DALL-E и Whispers.

Независимо от того, создаете ли вы чат-бот, ИИ-ассистента или веб-приложение, предоставляющее данные, сгенерированные ИИ, это руководство поможет реализовать ваши идеи.

Если вы владеете основами языка программирования Python и готовы изучить еще несколько инструментов, таких как объекты Dataframe библиотеки Pandas и некоторые методы NLP, значит, у вас есть под рукой все необходимое для создания интеллектуальных систем с использованием инструментов OpenAI.

Не беспокойтесь, вам не нужно быть специалистом по данным, инженером по машинному обучению или экспертом по искусственному интеллекту, чтобы понять концепции, методы и учебные примеры, представленные в этой книге. Все объяснения предельно просты для понимания, в них используется базовый код Python, примеры и практические упражнения.

Эта книга сфокусирована на практической части обучения и призвана помочь читателю создавать реальные приложения. Вы найдете здесь множество примеров кода, которые помогут усвоить базовые понятия и применить их в реальных сценариях для решения прикладных задач.

К концу обучения вы создадите такие приложения, как:

- медицинский чат-бот с пользовательской точной настройкой;
- интеллектуальная система подсказок любимого сорта кофе;
- интеллектуальная диалоговая система с памятью и контекстом;
- голосовой помощник с ИИ наподобие Alexa, но более умный;
- чат-бот, помогающий освоить команды Linux;
- семантическая поисковая система;
- система предсказания категории новостей;

[1] Во время работы над переводом этой книги продолжался прием заявок на раннее тестирование GPT-4, но автор и переводчик продолжали ожидать своей очереди, а сроки запуска модели в широкий доступ не были объявлены. – *Прим. перев.*

- интеллектуальная система распознавания изображений (изображение в текст);
- генератор изображений (текст в изображение);
- и многое другое!

Прочитав эту книгу и следуя примерам, вы освоите такие навыки, как:

- выбор подходящей модели из числа доступных, а также правильное использование каждой из них;
- генерация реалистичного текста для различных целей, таких как ответы на вопросы, создание контента и другие творческие применения;
- управление уровнем креативности моделей GPT и применение современных методов создания высококачественного текста;
- преобразование и редактирование текста для выполнения перевода, форматирования и других полезных задач;
- оптимизация производительности моделей GPT с помощью различных параметров и опций, таких как `suffix`, `max_tokens`, `temperature`, `top_p`, `n`, `stream`, `logprobs`, `echo`, `stop`, `presence_penalty`, `frequency_penalty`, `best_of` и др.;
- снижение стоимости использования API за счет сокращения и лемматизации текстов;
- применение заполнения контекста, создание смысловых цепочек и использование передовых технологий;
- использование встраивания текста аналогично тому, как это делают компании Tesla и Notion;
- применение семантического поиска и других передовых инструментов и концепций;
- создание алгоритмов прогнозирования и обучения без примеров и оценка их точности;
- использование обучения с несколькими примерами и улучшение его точности;
- использование тонкой настройки моделей, в том числе собственных;
- использование передового опыта для создания собственных моделей;
- применение различных методов обучения и классификации с использованием GPT;
- создание усовершенствованных моделей с помощью тонкой настройки;
- использование OpenAI Whisper и других инструментов для создания интеллектуальных голосовых помощников;
- классификация изображений с помощью OpenAI CLIP;
- создание и редактирование изображений с помощью OpenAI DALL-E;
- использование готовых изображений для создания собственных версий;

- обратное преобразование изображений из Stable Diffusion (изображение в текст).

Оставайтесь на связи

Если вы хотите быть в курсе последних тенденций в экосистемах Python и ИИ, присоединяйтесь к нашему сообществу разработчиков по адресу **www.faun.dev/join**. Мы отправляем еженедельные информационные бюллетени с важными и полезными учебными пособиями, новостями и идеями от экспертов в сообществе разработчиков программного обеспечения.

Глава 1

ChatGPT, GPT, GPT-3, DALL·E, Codex… Что это?

В декабре 2015 г. несколько блестящих новаторов объединилось для достижения общей цели: продвигать и развивать дружественный ИИ таким образом, чтобы он приносил пользу человечеству в целом.

Сэм Альтман, Илон Маск, Грег Брокман, Рид Хоффман, Джессика Ливингстон, Питер Тиль, Amazon Web Services (AWS), Infosys и YC Research объявили о создании компании OpenAI и пообещали выделить более 1 млрд долл. США на это предприятие. Новая организация сразу заявила, что будет свободно сотрудничать с другими компаниями и исследователями, делая свои патенты и исследования общедоступными.

Штаб-квартира OpenAI находится в здании Pioneer Building в районе Миссии в Сан-Франциско. В апреле 2016 г. OpenAI выпустила общедоступную бета-версию OpenAI Gym – своей платформы для исследований в области обучения с подкреплением. В декабре 2016 г. OpenAI выпустила Universe[1] – программную платформу широкого применения для измерения и обучения общего интеллекта ИИ в играх, веб-сайтах и других приложениях.

В 2018 г. Илон Маск покинул свое место в совете директоров, сославшись на потенциальный будущий конфликт интересов с разработкой искусственного интеллекта Tesla AI для беспилотных автомобилей, но остался инвестором. В 2019 г. OpenAI перешла от некоммерческой деятельности к бизнес-модели с ограниченной прибылью, при этом для любых инвестиций был установлен предел доходности 1 к 100. Компания распределила акции среди своих сотрудников и стала партнером Microsoft, которая инвестировала в проект 1 млрд долл. США. Затем OpenAI объявила о своем намерении лицензировать свои технологии на коммерческой основе.

В 2020 г. OpenAI анонсировала GPT-3 – языковую модель, обученную на триллионах слов из интернета, – и объявила, что API этой модели станет основой первого коммерческого продукта компании. Модель GPT-3 предназначена для ответов на вопросы на естественном языке, но она также

[1] **https://openai.com/blog/universe/.**

может выполнять перевод между языками и связно генерировать импровизированный текст. В 2021 г. OpenAI представила модель глубокого обучения DALL-E, способную генерировать цифровые изображения из описаний на естественном языке.

Перенесемся в декабрь 2022 г. После запуска бесплатной тестовой версии ChatGPT вокруг OpenAI разгорелась нешуточная шумиха в СМИ. По данным OpenAI, за первые пять дней на предварительное тестирование зарегистрировалось более миллиона человек. Согласно анонимным источникам, на которые ссылалось агентство Reuters в декабре 2022 г., OpenAI прогнозирует выручку в размере 200 млн долл. США в 2023 г. и 1 млрд долл. США в 2024 г. По состоянию на январь 2023 г. велись переговоры о следующем раунде финансирования, перед началом которого компания была оценена в 29 млрд долл.

Такова краткая история OpenAI, исследовательской лаборатории искусственного интеллекта, состоящей из коммерческой корпорации OpenAI LP и ее материнской компании, некоммерческой OpenAI Inc.

Большинство людей не знало о существовании OpenAI до того, как компания запустила свою невероятно популярную модель ChatGPT.

Основная цель ChatGPT заключалась в том, чтобы имитировать человеческое поведение и вести естественные диалоги с людьми. Кроме того, чат-бот может учиться на разговорах с разными пользователями. Этот ИИ не только общается с людьми на естественном языке, он также способен писать учебные пособия и код, сочинять музыку и выполнять другие задачи. Варианты использования ChatGPT весьма разнообразны и почти бесконечны; пользователи доказали это своими примерами. Одни пользователи занимались творчеством (например, сочинением рэпа), другие – вредоносной деятельностью (например, созданием вредоносного кода или команд), а третьи – бизнесом (например, контент-маркетингом, продажами по электронной почте, «холодными» рассылками электронных писем и повышением эффективности бизнеса).

Аббревиатура ChatGPT расшифровывается как Generative Pre-trained Transformer (генеративный предварительно обученный трансформер). Эта модель построена на основе семейства больших языковых моделей OpenAI GPT-3. Точную настройку чат-бота выполняли как с помощью методов обучения с учителем, так и с помощью обучения с подкреплением.

ChatGPT – это просто проект, использующий GPT-3 и добавляющий к базовой модели веб-интерфейс, память и другие удобные функции. Прочитав это руководство, вы сможете создать свой собственный чат-бот, возможно, лучше, чем ChatGPT, поскольку вы сможете настроить его под свои конкретные нужды.

К другим известным проектам, использующим GPT-3, среди прочих относятся:

- GitHub Copilot (с использованием модели OpenAI Codex, потомка GPT-3, настроенной для генерации кода);

- Copy.ai[1] и Jasper.ai[2] (генерация контента для маркетинговых целей);
- университет Дрекселя (выявление ранних признаков болезни Альцгеймера);
- Algolia (улучшение возможностей поисковой системы).

Вы наверняка уже слышали о GPT-4. Она является преемником GPT-3 и представляет собой нейронную сеть, также созданную OpenAI, но к ней пока не открыт широкий доступ.

Возможно, вам также довелось видеть диаграмму сравнения двух версий GPT, показывающую количество параметров в GPT-3 (175 млрд) и GPT-4 (100 трлн), как показано на рис. 1.1.

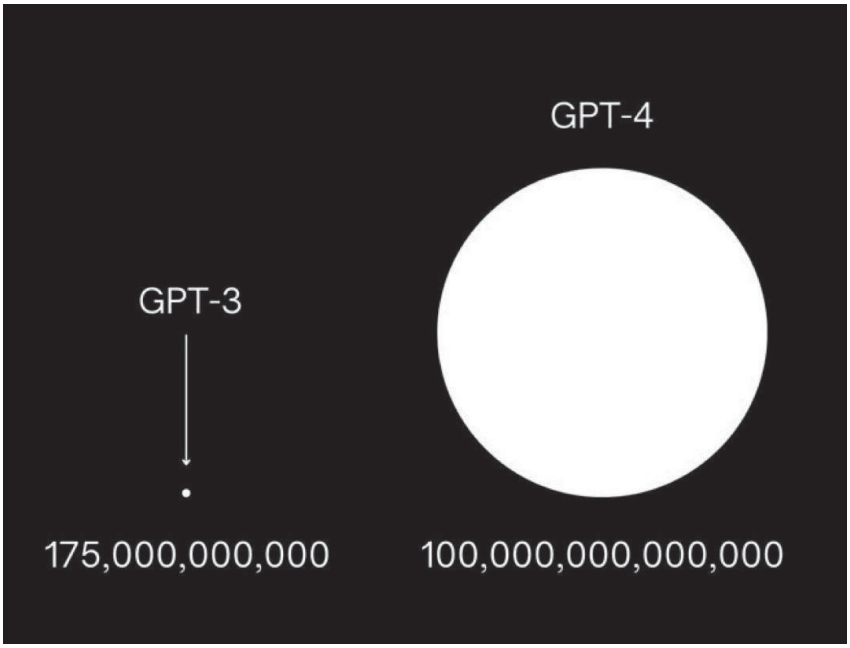


Рис. 1.1. Наглядное сравнение количества параметров моделей GPT-3 и GPT-4

Когда Сэма Альтмана спросили об этой вирусной иллюстрации, он назвал ее полной чушью:

> *«Невероятные слухи о GPT-4 – полная чушь. Я не знаю, кто все это придумывает. Некоторые люди упорно хотят разочароваться в собственных ожиданиях, и они обязательно разочаруются. Мы не изобрели подлинный искусственный интеллект, хотя от нас ждут именно этого».*

В том же интервью Альтман отказался подтвердить, будет ли модель вообще выпущена в 2023 г.

---

1 **http://copy.ai**.

2 **http://jasper.ai**.

Глава 2

Как работает GPT?

GPT – это *генеративная* текстовая модель. Такая модель способна создавать новый текст, предсказывая его продолжение на основе полученных входных данных.

GPT-3 – это просто обновленная и увеличенная модель, она заметно крупнее и производительнее, чем любая другая предыдущая модель GPT, включая GPT-1 и GPT-2.

Модель третьего поколения обучали на большом массиве текстов, таких как книги, статьи и общедоступные веб-сайты, такие как Reddit и другие форумы. Она использует эти обучающие данные для изучения закономерностей и взаимосвязей между словами и фразами.

Ключевое отличие GPT-3 заключается в ее впечатляющем размере – с ошеломляющими 175 млрд параметров, – что делает ее одной из самых массивных и мощных языковых моделей, когда-либо созданных человеком. Сочетание невиданного ранее количества параметров и обширного набора данных позволяет модели генерировать правдоподобные тексты и выполнять различные задачи обработки естественного языка с впечатляющим качеством.

GPT – это тип нейронной сети, относящейся к архитектуре Transformer, которую специально разработали для задач обработки естественного языка. В повседневной жизни модели с такой архитектурой часто называют просто трансформерами. Архитектура Transformer основана на последовательности блоков самовнимания, которые позволяют модели параллельно обрабатывать входной текст и взвешивать важность каждого слова или токена в зависимости от контекста.

Самовнимание (self-attention) – это механизм, используемый в моделях глубокого обучения для обработки естественного языка (NLP), который позволяет модели взвешивать важность различных частей предложения или нескольких предложений при прогнозировании. Являясь частью архитектуры Transformer, он позволяет нейронной сети достигать высокого качества работы, когда речь идет о задачах NLP.

Вот пример использования трансформеров, разработанных компанией Hugging Face для логического вывода GPT-2:

```
1 from transformers import pipeline
2 generator = pipeline('text-generation', model = 'gpt2')
```

```
generator("Привет, я языковая модель", max_length = 30, num_return_
sequences=3)
## [{'generated_text': "Привет, я языковая модель. Когда я писал это, когда
я встречался с женой или возвращался домой, она говорила мне, что мой"},
##  {'generated_text': "Привет, я языковая модель. Я пишу и поддерживаю
программы на Python. Я люблю программировать, и это включает в себя"}
...
```

По умолчанию модель не имеет памяти, это означает, что каждый ввод обрабатывается независимо, без переноса какой-либо информации из предыдущих вводов. Когда GPT генерирует текст, у нее *нет априорных представлений* о том, что должно быть дальше на основе предыдущих входных данных. Вместо этого она генерирует каждое слово на основе *вероятности* того, что оно будет следующим словом с учетом предыдущего ввода. Поэтому полученный текст иногда бывает неожиданным и не совсем осмысленным.

Вот еще один пример кода, использующего модель GPT для генерации текста на основе пользовательского ввода.

```
# Импорт необходимых библиотек
from transformers import GPT2Tokenizer, GPT2LMHeadModel

# Загрузка предварительно обученного токенизатора GPT-2 и модели
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2LMHeadModel.from_pretrained('gpt2')

# Настройка модели в режим оценки
model.eval()

# Определение начального запроса, за которым следует завершение
prompt = input("You: ")

# Токенизация запроса и генерация текста
input_ids = tokenizer.encode(prompt, return_tensors='pt')
output = model.generate(input_ids, max_length=50, do_sample=True)

# Декодирование сгенерированного текста и вывод в консоль
generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print("AI: " + generated_text)
```

GPT-3 целенаправленно разрабатывали как языковую модель общего назначения, поэтому ее можно использовать для различных задач обработки естественного языка, таких как языковой перевод, резюмирование текста и ответы на вопросы. Разработчики из OpenAI предварительно обучили GPT-3, но вы можете создать собственную модель с помощью процедуры тонкой настройки на собственных наборах данных. Это позволяет применять модель для более конкретных творческих задач в дополнение к задачам по умолчанию, таким как сочинение текста, стихов и рассказов. Настроенные модели также можно использовать для создания чат-ботов,

которые являются экспертами в определенной области, диалоговых интерфейсов и многого другого!

В этой книге будет подробно рассказано об OpenAI GPT-3, в том числе о том, как эффективно использовать API и инструменты, предоставляемые компанией и сообществом AI/ML, для создания прототипов мощных и креативных инструментов и систем. Это не только улучшит ваши навыки использования API GPT-3, но и расширит ваше понимание концепций и методов, используемых в обработке естественного языка и других смежных областях.

Благодаря огромному количеству параметров и впечатляющим результатам работы GPT-3 считается значительным достижением в обработке естественного языка. Однако некоторые эксперты выражают обеспокоенность по поводу того, что модель может создавать предвзятый или вредный контент. Как и в случае с любой технологией, здесь очень важно вовремя остановиться и подумать об этических последствиях ее использования. Но в этой книге мы не будем касаться этических вопросов, а сосредоточимся только на практических аспектах.

Приятного чтения!

Глава 3

Подготовка среды разработки

3.1. Установка Python, pip и виртуальной среды для разработки

Очевидно, что вам понадобится Python. В этой книге мы будем использовать Python 3.9.5.

Мы также будем использовать установщик пакетов pip. Вы можете использовать *pip* для установки пакетов из репозитория пакетов Python и других репозиториев.

Вам не нужно устанавливать Python 3.9.5 непосредственно в вашей системе, так как мы создадим виртуальную среду разработки. Какая бы у вас ни была установлена версия Python на рабочем компьютере, среда разработки будет изолирована от вашей системы и будет использовать Python 3.9.5.

Если Python не установлен, перейдите на страницу **www.python.org/downloads/**, загрузите и установите одну из версий Python 3.x. В зависимости от вашей операционной системы вам придется следовать разным инструкциям.

Для управления нашей средой разработки мы будем использовать *virtualenvwrapper*[1]. Инструкции по установке вы найдете в официальной документации[2].

Самый простой способ получить его – использовать *pip*:

```
pip install virtualenvwrapper
```

> *Примечание:* если вы привыкли к *virtualenv*, Poetry или любому другому диспетчеру пакетов, вы можете продолжать его использовать. В этом случае нет необходимости устанавливать *virtualenvwrapper*.

Если *pip* не установлен, проще всего установить его с помощью скрипта, представленного в официальной документации.

Загрузите скрипт с **https://bootstrap.pypa.io/get-pip.py**.

Откройте окно терминала / командной строки, перейдите в папку, содержащую файл `get-pip.py`, и выполните команду

---

[1] **https://github.com/python-virtualenvwrapper/virtualenvwrapper**.

[2] **https://virtualenvwrapper.readthedocs.io/en/latest/install.html**.

```
python get-pip.py
```

если вы пользователь MacOs или Linux, или команду

```
py get-pip.py
```

если вы пользователь Windows.

В итоге в вашей системе должны быть установлены следующие пакеты:

- Python,
- Pip,
- *virtualenvwrapper* (либо *virtualenv* или любой другой менеджер пакетов, который вы предпочитаете).

Я настоятельно рекомендую пользователям Windows создать виртуальную машину с операционной системой Linux, так как большинство примеров, представленных в этой книге, выполнялись и тестировались в системе Linux Mint.

Далее создадим виртуальную среду:

```
mkvirtualenv -p python3.9 chatgptforpythondevelopers
```

После создания виртуальной среды активируйте ее:

```
workon chatgptforpythondevelopers
```

3.2. Получение ключа API OpenAI

Следующим шагом является создание ключей API, которые позволят вам получить доступ к официальному API, предоставляемому OpenAI. Перейдите на **https://openai.com/api/** и создайте учетную запись.

> **Примечание к переводу:** сайт и сервисы OpenAI недоступны для пользователей с российским IP-адресом. Для создания учетной записи вам понадобится доступ через VPN с европейским IP и временный виртуальный телефонный номер в любой европейской стране, на который вы получите СМС с кодом подтверждения регистрации. В дальнейшем вам этот телефонный номер не понадобится, но доступ к API возможен только через VPN. Мы не будем здесь детально описывать использование виртуальных телефонных номеров, но отметим, что российским пользователям доступно множество сервисов, предоставляющих виртуальные телефонные номера для получения СМС в различных странах с оплатой российскими банковскими картами, включая «Мир», по очень доступной цене. Как показал опыт, для регистрации и входа на сайт OpenAI можно использовать российский аккаунт Google. Процедура регистрации на сайте OpenAI детально описана в различных блогах российских авторов.

Следуйте инструкциям по созданию учетной записи, а затем создайте ключи API по адресу: **https://beta.openai.com/account/api-keys.**

Ключ API должен принадлежать организации, вам будет предложено создать организацию. В этой книге мы будем называть ее LearningGPT.

Сохраните сгенерированный секретный ключ в надежном и доступном месте. Вы не сможете увидеть его снова через свою учетную запись OpenAI.

> **Примечание к переводу:** по состоянию на май 2023 г. новым пользователям на счет зачислялся грант в размере 18 долл., действующий в течение 40 дней, после чего неиспользованный остаток гранта «сгорает». Имейте это в виду при проведении экспериментов за счет гранта. Разумеется, условия предоставления гранта могут в любой момент измениться без предупреждения со стороны OpenAI. К сожалению, на данный момент для российских пользователей не существует простого способа оплаты сервисов OpenAI после исчерпания гранта. Необходимо иметь банковскую карту Visa или Mastercard, выпущенную западным банком, поэтому проблему оплаты каждый пользователь должен решать самостоятельно. Впрочем, начального гранта вполне достаточно для экспериментов и знакомства с API OpenAI.

3.3. Установка официальных средств интеграции Python

Вы можете взаимодействовать с API, используя HTTP-запросы с любого языка через официальные средства интеграции (binding, привязка) Python, или через официальную библиотеку Node.js, или через библиотеку, поддерживаемую сообществом.

В этой книге мы будем использовать официальную библиотеку, предоставленную OpenAI. Другой альтернативой является использование Chronology[1], неофициальной библиотеки, предоставленной OthersideAI. Однако похоже, что эта библиотека больше не обновляется.

Чтобы установить официальные средства интеграции Python, выполните следующую команду:

```
pip install openai
```

Убедитесь, что вы устанавливаете библиотеку в виртуальной среде, которую мы создали ранее.

3.4. Тестирование ключей API

Чтобы убедиться, что все работает правильно, выполним вызов через `curl`. Перед этим нужно сохранить ключ API и идентификатор организации в файле с именем `.env` следующего содержания:

```
cat << EOF > .env
API_KEY=xxx
ORG_ID=xxx
EOF
```

[1] **https://github.com/OthersideAI/chronology**.

Перед выполнением команды `curl` обязательно обновите переменные `API_KEY` и `ORG_ID` на их соответствующие значения из вашей учетной записи.

Теперь вы можете выполнить следующую команду (одна строка):

```
source .env
curl https://api.openai.com/v1/models
  -H 'Authorization: Bearer '$API_KEY''
  -H 'OpenAI-Organization: '$ORG_ID''
```

Вы можете обойтись без ссылки на переменные в файле и использовать их непосредственно в команде `curl`:

```
curl https://api.openai.com/v1/models
  -H 'Authorization: Bearer xxxx'
  -H 'OpenAI-Organization: xxxx'
```

Если в вашей учетной записи OpenAI есть только одна организация, можно выполнить ту же команду без указания идентификатора организации.

```
curl https://api.openai.com/v1/models -H 'Authorization: Bearer xxxx'
```

Команда `curl` должна вернуть вам список моделей, предоставляемых API, таких как `davinci`, `ada` и многие другие.

Чтобы протестировать API с помощью кода Python, вы можете выполнить следующий код:

```
import os
import openai

# Чтение переменных из файла .env, а именно API_KEY и ORG_ID
with open(".env") as env:
    for line in env:
        key, value = line.strip().split("=")
        os.environ[key] = value

# Инициализация ключа API и идентификатора организации
openai.api_key = os.environ.get("API_KEY")
openai.organization = os.environ.get("ORG_ID")

# Вызов API и получение списка моделей
models = openai.Model.list()
print(models)
```

Мы будем использовать примерно такой же подход в будущем, поэтому давайте создадим повторно используемую функцию для инициализации API. Она может выглядеть, например, так:

```
import os
import openai

def init_api():
```

```
        with open(".env") as env:
            for line in env:
                key, value = line.strip().split("=")
                os.environ[key] = value

        openai.api_key = os.environ.get("API_KEY")
        openai.organization = os.environ.get("ORG_ID")

init_api()

models = openai.Model.list()
print(models)
```

Глава 4

Доступные модели

4.1. Три основные модели

OpenAI предлагает следующие базовые модели или *семейства моделей*, если их можно так назвать:

- GPT-3,
- Codex,
- Content Filter (фильтр контента).

Наверняка вы встречали в интернете и другие названия моделей OpenAI, и это может сбивать с толку. Многие публикации относятся к более старым версиям, а не к поколению GPT-3.

Также помимо использования трех базовых моделей вы можете создавать и настраивать собственные модели.

4.2. GPT-3: обработка и генерация естественного языка

Модели поколения GPT-3 способны понимать человеческий язык и генерировать текст, который выглядит так, будто он написан человеком. Это семейство моделей состоит из четырех моделей. В списке ниже они отсортированы по убыванию скорости и качества:

- text-davinci-003,
- text-curie-001,
- text-babbage-001,
- text-ada-001.

Как уже было сказано, каждая из них имеет разные возможности, стоимость и точность.

OpenAI рекомендует сначала поэкспериментировать с моделью Davinci, а затем попробовать другие модели, способные выполнять большое количество аналогичных задач при гораздо меньших затратах.

`text-davinci-003`

Это самая мощная модель GPT-3, поскольку она может выполнять то же, что и все остальные модели, но делает это быстрее и зачастую

лучше. Это самая последняя модель на сегодняшний день, обученная на данных, датированных до июня 2021 г.

Одним из ее преимуществ является обработка запросов длиной до 4 тыс. токенов. Она также поддерживает вставку дополнений в текст.

Мы более подробно определим понятие *токена* позже. Пока просто знайте, что количество токенов определяет длину запроса пользователя.

`text-curie-001`

Модель `text-curie-001` является второй по производительности моделью GPT-3, поскольку она поддерживает до 2048 токенов. Ее преимущество в том, что она более экономична, чем `text-davinci-003`, но при этом обладает высокой точностью.

Она была обучена на данных, датированных октябрем 2019 г., поэтому немного менее точна, чем `text-davinci-003`. Она может быть хорошим вариантом для перевода, сложной классификации, анализа и резюмирования текста.

`text-babbage-001`

То же, что и у `text-curie-001: 2048` токенов и обучение на данных до октября 2019 г.

Эта модель подходит для более простых задач категоризации и семантической классификации.

`text-ada-001`

То же, что и у `text-curie-001: 2048` токенов и обучение на данных до октября 2019 г.

Эта модель очень быстра и экономична, и ее желательно использовать для простейшей классификации, извлечения и исправления текста.

4.3. Codex: понимание и создание компьютерного кода

OpenAI предлагает две модели Codex, способные понимать и создавать компьютерный код: `code-davinci-002` и `code-cushman-001`.

Codex – это модель, на которой основан сервис GitHub Copilot. Она владеет более чем дюжиной языков программирования, включая Python, JavaScript, Go, Perl, PHP, Ruby, Swift, TypeScript, SQL и Shell.

Codex может понимать основные инструкции, выраженные на естественном разговорном языке, и выполнять запрошенные задачи от имени пользователя.

Как сказано выше, на базе Codex доступны две модели:

- code-davinci-002,
- code-cushman-001.

code-davinci-002

Эта модель Codex является наиболее функциональной. Она превосходно переводит естественный язык в код. Она не только завершает код, но также поддерживает вставку дополнительных элементов. Модель может обрабатывать до 8000 токенов и обучена на данных, датированных июнем 2021 г.

code-cushman-001

Это мощная и быстрая модель. Даже если Davinci мощнее, когда дело доходит до анализа сложных задач, Cushman больше подходит для многих задач генерации кода.

Эта модель также быстрее и доступнее, чем Davinci.

4.4. Content Filter

Как следует из названия, это фильтр для конфиденциального контента.

С помощью этого фильтра вы можете обнаружить сгенерированный API текст, который содержит конфиденциальную или небезопасную информацию. Фильтр может классифицировать текст по трем категориям:

- safe (безопасный);
- sensitive (чувствительный);
- unsafe (небезопасный).

Если вы создаете приложение для сторонних пользователей, вы можете использовать фильтр, чтобы определить, возвращает ли модель какой-либо неприемлемый контент.

4.5. Получение списка всех доступных моделей

Используя конечную точку `models` API, вы можете получить список всех доступных моделей. Давайте посмотрим, как это работает на практике:

```
import os
import openai

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

models = openai.Model.list()
print(models)
```

Как видите, мы просто используем метод `list()` из модуля `Model` пакета `openai` и в результате получаем список наподобие следующего:

```
{
  "data": [
    {
      "created": 1649358449,
      "id": "babbage",
      "object": "model",
      "owned_by": "openai",
      "parent": null,
      "permission": [
        {
          "allow_create_engine": false,
          "allow_fine_tuning": false,
          "allow_logprobs": true,
          "allow_sampling": true,
          "allow_search_indices": false,
          "allow_view": true,
          "created": 1669085501,
          "group": null,
          "id": "modelperm-49FUp5v084tBB49tC4z8LPH5",
          "is_blocking": false,
          "object": "model_permission",
          "organization": "*"
        }
      ],
      "root": "babbage"
    },
                [...]
                [...]
                [...]
                  [...]
                  [...]
                  {
      "created": 1642018370,
      "id": "text-babbage:001",
      "object": "model",
      "owned_by": "openai",
      "parent": null,
      "permission": [
        {
          "allow_create_engine": false,
          "allow_fine_tuning": false,
          "allow_logprobs": true,
          "allow_sampling": true,
          "allow_search_indices": false,
          "allow_view": true,
          "created": 1642018480,
          "group": null,
          "id": "snapperm-7oP3WFr9x7qf5xb3eZrVABAH",
```

```
            "is_blocking": false,
            "object": "model_permission",
            "organization": "*"
          }
        ],
        "root": "text-babbage:001"
      }
    ],
    "object": "list"
}
```

Давайте напечатаем только идентификаторы моделей:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

models = openai.Model.list()
for model in models["data"]:
   print(model["id"])
```

Результат должен быть следующим (на момент написания книги):

```
babbage
ada
davinci
text-embedding-ada-002
babbage-code-search-code
text-similarity-babbage-001
text-davinci-003
text-davinci-001
curie-instruct-beta
babbage-code-search-text
babbage-similarity
curie-search-query
code-search-babbage-text-001
code-cushman-001
code-search-babbage-code-001
audio-transcribe-deprecated
code-davinci-002
text-ada-001
text-similarity-ada-001
text-davinci-insert-002
```

```
ada-code-search-code
text-davinci-002
ada-similarity
code-search-ada-text-001
text-search-ada-query-001
text-curie-001
text-davinci-edit-001
davinci-search-document
ada-code-search-text
text-search-ada-doc-001
code-davinci-edit-001
davinci-instruct-beta
text-similarity-curie-001
code-search-ada-code-001
ada-search-query
text-search-davinci-query-001
davinci-search-query
text-davinci-insert-001
babbage-search-document
ada-search-document
text-search-babbage-doc-001
text-search-curie-doc-001
text-search-curie-query-001
babbage-search-query
text-babbage-001
text-search-davinci-doc-001
text-search-babbage-query-001
curie-similarity
curie-search-document
curie
text-similarity-davinci-001
davinci-similarity
cushman:2020-05-03
ada:2020-05-03
babbage:2020-05-03
curie:2020-05-03
davinci:2020-05-03
if-davinci-v2
if-curie-v2
if-davinci:3.0.0
davinci-if:3.0.0
davinci-instruct-beta:2.0.0
text-ada:001
text-davinci:001
text-curie:001
text-babbage:001
```

Итак, вы уже знаете, что есть базовые модели `Ada`, `Babbage`, `Curie`, `Cushman` и `Davinci`.

Вы также знакомы с моделью `Codex`. Каждая модель, имеющая слово `code` в своем идентификаторе, является частью семейства Codex.

Модели поиска, такие как babbage-code-search-code, оптимизированы для поиска по коду, но некоторые из них в настоящее время являются устаревшими, включая ada-code-search-code, ada-code-search-text, babbage-code-search-code и babbage-code-search-text, и они были заменены соответственно на code-search-ada-code-001, code-search-ada-text-001, code-search-babbage-code-001 и code-search-babbage-text-001.

Другие модели поиска, оптимизированные для поиска текста, такие как ada-search-document, ada-search-query, babbage-search-document, babbage-search-query, curie-search-document и curie-search-query, также устарели и были заменены на text-search-ada-doc-001, text-search-ada-query-001, text-search-babbage-doc-001, text-search-babbage-query-001, text-search-curie-doc-001 и text-search-curie-query-001.

Модели «подобия» предназначены для поиска похожих документов. Модели ada-similarity, babbage-similarity, curie-similarity, davinci-similarity были заменены на text-similarity-ada-001, text-similarity-babbage-001, text-similarity-curie-001 и text-similarity-davinci-001.

Устаревшие модели davinci-instruct-beta-v3, curie-instruct-beta-v2, babbage-instruct-beta и ada-instruct-beta заменены на text-davinci-001, text-curie-001, text-babbage-001 и text-ada-001.

Обратите внимание, что davinci – это не то же самое, что text-davinci-003 или text-davinci-002. Все они мощные, но у них есть некоторые различия, такие как количество токенов, которые они поддерживают.

Многие из этих устаревших моделей раньше назывались «движками» (engine), но OpenAI отказался от термина «движок» в пользу «модели». Многие пользователи и онлайн-ресурсы используют эти термины как синонимы, но правильное название – «модель».

Запросы API со старыми именами по-прежнему будут работать, поскольку OpenAI обеспечивает обратную совместимость. Однако рекомендуется использовать обновленные имена.

Пример устаревшего запроса:

```
response = openai.Completion.create(
  model="davinci",
  prompt="В чем смысл жизни?",
)
```

Пример нового правильного запроса:

```
response = openai.Completion.create(
  model="text-davinci-002",
  prompt="В чем смысл жизни?",
)
```

4.6. Какую модель использовать?

Модели семейства Davinci, безусловно, самые лучшие, но и самые дорогие. Поэтому, если вы не очень ограничены в расходах и хотите сосредоточиться

на качестве, Davinci будет лучшим выбором. В частности, `text-davinci-003` – самая мощная модель.

По сравнению с `davinci`, `text-davinci-003` – это более новая, более мощная модель, разработанная специально для задач, требующих выполнения инструкций, и сценариев обучения без примеров (zero-shot learning). Концепцию обучения на ограниченном количестве примеров мы рассмотрим позже.

Однако имейте в виду, что для некоторых конкретных случаев использования модели Davinci не всегда подходят. Об этом мы также поговорим позже.

Другие модели, такие как Curie, являются хорошими альтернативами для оптимизации затрат, особенно если вы выполняете простые запросы, например резюмирование текста или извлечение данных.

Сказанное относится как к GPT-3, так и к Codex.

Фильтр контента является необязательным, но настоятельно рекомендуется, если вы создаете общедоступное приложение. Вы же не хотите, чтобы ваши пользователи получали недопустимые результаты?

4.7. Что дальше?

Подводя итог этой главе, можно сказать, что модели семейства Davinci, такие как `text-davinci-003`, являются лучшими на сегодняшний день. Модель `text-davinci-003` рекомендована OpenAI для большинства случаев использования. Другие модели, такие как Curie, могут работать очень хорошо в определенных случаях использования и обходятся примерно в 10 раз дешевле Davinci.

Модель ценообразования OpenAI понятна и проста. Тарифы можно найти на официальном сайте по адресу **https://openai.com/api/pricing/**. Кроме того, при регистрации в OpenAI вы получите грант в размере 18 долл., который можно использовать в течение 40 дней (правило, действовавшее в мае 2023).

В этом руководстве мы собираемся следовать рекомендациям OpenAI и использовать новые модели, а не устаревшие.

По мере того как мы будем осваивать эти модели, вы лучше поймете, как использовать API для создания, генерации и редактирования текста и изображений, используя либо модели, либо ваши собственные доработанные модели. Работая с практическими примерами, вы научитесь интуитивно соотносить токены и цены и к концу книги сможете создавать готовые к применению интеллектуальные приложения на основе ИИ.

Глава 5

Применение GPT для генерации текста

После успешной аутентификации вы можете начать использовать API OpenAI для генерации ответов на текстовые запросы. Для этого вам нужно использовать OpenAI Completion API.

Конечная точка Completion позволяет разработчикам получать доступ к наборам данных и моделям OpenAI, упрощая завершение текстового запроса.

Вы передаете модели начало предложения, а она предсказывает одно или несколько возможных завершений, каждое из которых будет иметь соответствующую оценку.

5.1. Базовый пример завершения текста

В качестве примера передадим API предложение *Once upon a time*[1], а модель должна вернуть возможное продолжение.

Активируйте среду разработки следующей командой:

```
workon chatgptforpythondevelopers
```

Создайте новый файл Python `app.py`, куда добавьте следующий код (или воспользуйтесь готовым файлом из архива книги):

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()
```

[1] Аналог русского «жили-были». – *Прим. перев.*

```
next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=7,
    temperature=0
)

print(next)
```

Теперь выполним этот код:

```
python app.py
```

API вернет текст наподобие следующего:

```
{
  "choices": [
    {
      "finish_reason": "length",
      "index": 0,
      "logprobs": null,
      "text": " there was a little girl named Alice"
    }
  ],
  "created": 1674510189,
  "id": "cmpl-6bysnKOUj0QWOu5DSiJAGcwIVMfNh",
  "model": "text-davinci-003",
  "object": "text_completion",
  "usage": {
  "completion_tokens": 7,
  "prompt_tokens": 4,
  "total_tokens": 11
  }
}
```

В приведенном выше примере у нас есть единственный выбор: `there was a little girl named Alice` (жила-была маленькая девочка по имени Алиса). Этот результат имеет индекс 0.

API также вернул `finish_reason` (причина завершения), в данном случае это `length` (длина).

Длина вывода определяется API на основе значения `max_tokens` (максимальное количество токенов), предоставленного пользователем. В нашем случае мы устанавливаем это значение равным `7`.

> **Примечание:** токены по определению представляют собой общие последовательности символов в выходном тексте. Вы можете запомнить, что один токен обычно содержит около четырех букв текста для обычных английских слов. Это означает, что 100 токенов примерно равны 75 словам. Понимание этого поможет вам оценить стоимость запроса. Позже в этой книге мы углубимся в детали ценообразования.

5.2. Управление количеством токенов на выходе

Проверим на более длинном примере, что означает большее количество токенов (15):

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
model="text-davinci-003",
prompt="Once upon a time",
    max_tokens=15,
temperature=0
    )

    print(next)
```

API возвращает более длинный текст:

```
{
  "choices": [
    {
      "finish_reason": "length",
      "index": 0,
      "logprobs": null,
      "text": " there was a little girl named Alice. She lived in a small
village with"
    }
  ],
  "created": 1674510438,
  "id": "cmpl-6bywotGlkujbrEF0emJ8iJmJBEEnO",
  "model": "text-davinci-003",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 15,
    "prompt_tokens": 4,
    "total_tokens": 19
  }
}
```

5.3. Параметр *logprobs*

Чтобы увеличить возможности модели, мы можем использовать параметр `logprobs`. Например, если задать для `logprobs` значение 2, будут возвращены две версии каждого токена.

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=7,
    temperature=0,
    logprobs=2
)

print(next)
```

Вот что вернул API:

```
{
  "choices": [
    {
      "finish_reason": "length",
      "index": 0,
      "logprobs": {
      "text_offset": [
        16,
        22,
        26,
        28,
        35,
        40,
        46,
        52,
        53,
        57,
        63,
        66,
        68,
```

```
      74,
      82
    ],
    "token_logprobs": [
      -0.9263134,
      -0.2422086,
      -0.039050072,
      -1.8855333,
      -0.15475112,
      -0.30592665,
      -1.9697434,
      -0.34726024,
      -0.46498245,
      -0.46052673,
      -0.14448218,
      -0.0038384167,
      -0.029725535,
      -0.34297562,
      -0.4261593
    ],
    "tokens": [
      " there",
      " was",
      " a",
      " little",
      " girl",
      " named",
      " Alice",
      ".",
      " She",
      " lived",
      " in",
      " a",
      " small",
      " village",
      " with"
    ],
    "top_logprobs": [
      {
        "\n": -1.1709108,
        " there": -0.9263134
      },
      {
        " lived": -2.040701,
        " was": -0.2422086
      },
      {
        " a": -0.039050072,
        " an": -3.403554
      },
      {
```

```
            " little": -1.8855333,
            " young": -2.02082
          },
          {
            " boy": -2.449015,
            " girl": -0.15475112
          },
          {
            " named": -0.30592665,
            " who": -1.7700866
          },
          {
            " Alice": -1.9697434,
            " Sarah": -2.9232333
          },
          {
            " who": -1.3002346,
            ".": -0.34726024
          },
          {
            " Alice": -1.2721952,
            " She": -0.46498245
          },
          {
            " lived": -0.46052673,
            " was": -1.7077477
          },
          {
            " in": -0.14448218,
            " with": -2.0538774
          },
          {
            " a": -0.0038384167,
            " the": -5.8157005
          },
          {
            " quaint": -4.941383,
            " small": -0.029725535
          },
          {
            " town": -1.454277,
            " village": -0.34297562
          },
          {
            " in": -1.7855972,
            " with": -0.4261593
          }
        ]
      },
      "text": " there was a little girl named Alice. She lived in a small
village with"
```

```
    }
  ],
  "created": 1674511658,
  "id": "cmpl-6bzGUTuc5AmbjsoNLNJULTaG0WWUP",
  "model": "text-davinci-003",
  "object": "text_completion",
  "usage": {
      "completion_tokens": 15,
      "prompt_tokens": 4,
      "total_tokens": 19
    }
  }
```

Как видите, каждый токен имеет связанную с ним вероятность или *рейтинг* токена. Например, между токенами `\n` и `there` API выберет `there`, поскольку `-1.1709108` меньше `-0.9263134`.

Аналогично API выберет `was` вместо `lived`, поскольку `-0.2422086` больше, чем `-2.040701`. Точно так же это работает и для других токенов:

```
{
  "\n": -1.1709108,
  " there": -0.9263134
},
{
  " lived": -2.040701,
  " was": -0.2422086
},
{
  " a": -0.039050072,
  " an": -3.403554
},
{
  " little": -1.8855333,
  " young": -2.02082
},
{
  " boy": -2.449015,
  " girl": -0.15475112
},
{
  " named": -0.30592665,
  " who": -1.7700866
},
{
  " Alice": -1.9697434,
  " Sarah": -2.9232333
},
{
  " who": -1.3002346,
  ".": -0.34726024
},
```

```
        {
          " Alice": -1.2721952,
          " She": -0.46498245
        },
        {
          " lived": -0.46052673,
          " was": -1.7077477
        },
        {
          " in": -0.14448218,
          " with": -2.0538774
        },
        {
          " a": -0.0038384167,
          " the": -5.8157005
        },
        {
          " quaint": -4.941383,
          " small": -0.029725535
        },
        {
          " town": -1.454277,
          " village": -0.34297562
        },
        {
          " in": -1.7855972,
          " with": -0.4261593
        }
```

Каждый токен имеет два возможных значения. API возвращает вероятность каждого из них и предложения, сформированного токенами с наибольшей вероятностью.

```
"tokens": [
" there",
" was",
" a",
" little",
" girl",
" named",
" Alice",
".",
" She",
" lived",
" in",
" a",
" small",
" village",
" with"
17 ],
```

Мы можем увеличить значение параметра `logprobs` до `5`, поскольку это максимальное значение согласно правилам OpenAI. Если вам недостаточно этого значения, обратитесь в службу поддержки и объясните свои потребности.

5.4. Управление креативностью: параметр *temperature*

Следующий параметр, который мы можем настроить, – это `temperature`. Его можно использовать, чтобы сделать сгенерированный текст более творческим, но высокая креативность связана с некоторыми рисками.

Чтобы дать модели больше творческой свободы, мы можем использовать более высокие значения `temperature`, такие как `0.2`, `0.3`, `0.4`, `0.5` и `0.6`. Максимальное значение параметра `temperature` равно `2`.

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=15,
    temperature=2,
)

print(next)
```

API вернет следующий текст:

```
{
  "choices": [
    {
      "finish_reason": "length",
      "index": 0,
      "logprobs": null,
      "text": " there lived\ntwo travellers who ravret for miles through
desert forest carwhen"
    }
```

```
  ],
  "created": 1674512348,
  "id": "cmpl-6bzRc4nJXBKBaOE6pr5d4bLCLI7N5",
  "model": "text-davinci-003",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 15,
    "prompt_tokens": 4,
    "total_tokens": 19
  }
}
```

В этом коде параметр `temperature` был установлен на максимальное значение, поэтому выполнение того же скрипта возвращает более креативный результат. Здесь в игру вступает творчество.

5.5. Использование параметра *top_p*

В качестве альтернативы мы могли бы использовать параметр `top_p`. Например, при значении `0.5` учитываются только токены с наибольшей вероятностной мерой, составляющей 50 %; `top_p=0.1` означает, что учитываются токены с наибольшей вероятностной мерой, составляющей 10 %.

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=15,
    top_p=9,
)

print(next)
```

Рекомендуется варьировать либо параметр `top_p`, либо параметр `temperature`, но не оба сразу. Параметр `top_p` также называется *ядерным семплированием* (nucleus sampling) *или выборкой с ограничением маловероятных токенов*.

5.6. Потоковая передача результатов

Еще один общий параметр, который мы можем использовать в OpenAI, – это `stream` (поток). Можно указать API возвращать поток токенов вместо блока, содержащего все токены. В этом случае API вернет генератор, который выдает токены в том порядке, в каком они были сгенерированы.

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=15,
    stream=True,
)

# возвращает <class 'generator'>
print(type(next))

# распаковывает генератор
print(*next, sep='\n')
```

Этот код должен вывести на печать тип `next`, представляющий собой `<class 'generator'>`, за которым следуют токены:

```
{
  "choices": [
    {
      "finish_reason": null,
      "index": 0,
      "logprobs": null,
      "text": " there"
    }
  ],
  "created": 1674594500,
  "id": "cmpl-6cKoeTcDzK6NnemYNgKcmpvCvT9od",
  "model": "text-davinci-003",
  "object": "text_completion"
}
```

```
{
  "choices": [
    {
      "finish_reason": null,
      "index": 0,
      "logprobs": null,
      "text": " was"
    }
  ],
  "created": 1674594500,
  "id": "cmpl-6cKoeTcDzK6NnemYNgKcmpvCvT9od",
  "model": "text-davinci-003",
  "object": "text_completion"
}
{
  "choices": [
    {
      "finish_reason": null,
      "index": 0,
      "logprobs": null,
      "text": " a"
    }
  ],
  "created": 1674594500,
  "id": "cmpl-6cKoeTcDzK6NnemYNgKcmpvCvT9od",
  "model": "text-davinci-003",
  "object": "text_completion"
}
{
  "choices": [
    {
      "finish_reason": null,
      "index": 0,
      "logprobs": null,
      "text": " girl"
    }
  ],
  "created": 1674594500,
  "id": "cmpl-6cKoeTcDzK6NnemYNgKcmpvCvT9od",
  "model": "text-davinci-003",
  "object": "text_completion"
}
{
  "choices": [
    {
      "finish_reason": null,
      "index": 0,
      "logprobs": null,
      "text": " who"
    }
  ],
```

```
  "created": 1674594500,
  "id": "cmpl-6cKoeTcDzK6NnemYNgKcmpvCvT9od",
  "model": "text-davinci-003",
  "object": "text_completion"
}
{
  "choices": [
    {
      "finish_reason": null,
      "index": 0,
      "logprobs": null,
      "text": " was"
    }
  ],
  "created": 1674594500,
  "id": "cmpl-6cKoeTcDzK6NnemYNgKcmpvCvT9od",
  "model": "text-davinci-003",
  "object": "text_completion"
}
{
  "choices": [
    {
      "finish_reason": "length",
      "index": 0,
      "logprobs": null,
      "text": " very"
    }
  ],
  "created": 1674594500,
  "id": "cmpl-6cKoeTcDzK6NnemYNgKcmpvCvT9od",
  "model": "text-davinci-003",
  "object": "text_completion"
}
```


Если вы хотите получить только текст, вы можете использовать код наподобие следующего:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
```

```
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=15,
    stream=True,
)

# Читает элементы генератора текста по одному
for i in next:
    print(i['choices'][0]['text'])
```

Этот код должен вывести на печать слова:

```
there
was
a
small
village
that
was
```

5.7. Контроль повторений: штрафы за частоту и наличие

Completions API имеет две опции, которые можно использовать для предотвращения слишком частого предложения одних и тех же слов. Эти функции изменяют вероятность выдачи определенных слов, добавляя бонус или штраф к *логитам* (числа на выходе модели, которые показывают, насколько вероятно, что слово будет предложено).

Эти опции можно применить с помощью двух параметров:

- `presence_penalty` может принимать значения от `-2.0` до `2.0`. Если значение положительное, это повышает вероятность того, что модель будет говорить о новых темах, потому что она будет оштрафована, если она использует слова, которые уже использовались;
- `frequency_penalty` тоже может принимать значения `-2.0` до `2.0`. Положительные значения снижают вероятность того, что модель будет повторять одну и ту же строку текста, которая уже использовалась.

Чтобы понять влияние этих параметров, давайте используем их в следующем коде:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value
```

```
    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=100,
    frequency_penalty=2.0,
    presence_penalty=2.0,
)

print("=== Штраф за частоту и повторение равен 2.0 ===")
print(next["choices"][0]["text"])

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=100,
    frequency_penalty=-2.0,
    presence_penalty=-2.0,
)

print("=== Штраф за частоту и повторение равен -2.0 ===")
print(next["choices"][0]["text"])
```

Как видите, первый запрос должен обеспечить большее разнообразие в тексте (`frequency_penalty=2.0` и `presence_penalty=2.0`), а второе сочетание параметров должно дать совершенно противоположный эффект (`frequency_penalty=-2.0` и `presence_penalty=-2.0`).

Выполнение приведенного выше кода даст следующий вывод:

```
=== Штраф за частоту и повторение равен 2.0 ===
there was a beautiful princess named Cinderella.
She lived with her wicked stepmother and two jealous stepsisters who treated
her like their servant. One day, an invitation arrived to the palace ball
where all of the  eligible young ladies in town were invited by the prince
himself! Cinderella's determination kept fueling when she noticed how cruelly
she been mistreated as if it gave her more strength to reach for what she
desired most - to attend that magnificent event alongside everyone else but not
just only that

(жила-была прекрасная принцесса по имени Золушка.
Она жила со своей злой мачехой и двумя ревнивыми сводными сестрами, которые
относились к ней как к своей служанке. Однажды пришло приглашение на дворцовый
бал, куда все достойные юные леди города были приглашены самим принцем!
Решимость Золушки становилась только сильнее, когда она замечала, как жестоко
с ней обращались, как будто это давало ей больше сил, чтобы достичь того, чего
она желала больше всего - посетить это великолепное событие вместе со всеми,
но не только это)
```

```
=== Штраф за частоту и повторение равен -2.0 ===
there lived a little girl named Lucy. She lived a very happy life with her
family. She lived a very simple life. She lived a very happy life. She lived
a very happy life. She lived a very happy life. She lived a very happy life.
She lived a very happy life. She lived a very happy life. She lived a very
happy life. She lived a very happy life. She lived a very happy life. She
lived a very happy life. She lived a very

(жила-была маленькая девочка по имени Люси. Она прожила очень счастливую жизнь
со своей семьей. Она жила очень простой жизнью. Она прожила очень счастливую
жизнь. Она прожила очень счастливую жизнь. Она прожила очень счастливую жизнь.
Она прожила очень счастливую жизнь. Она прожила очень счастливую жизнь. Она
прожила очень счастливую жизнь. Она прожила очень счастливую жизнь. Она
прожила очень счастливую жизнь. Она прожила очень счастливую жизнь. Она
прожила очень счастливую жизнь. Она прожила очень)
```

Как видите, во втором случае очень скоро начинает бесконечно повторяться строка «Она прожила очень счастливую жизнь».

5.8. Управление количеством выводимых результатов

Если вы хотите получить более одного результата, воспользуйтесь параметром `n`. Следующий пример даст два результата:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=5,
    n=2,
)

print(next)
```

Так выглядит пример вывода, созданного показанным выше кодом Python:

```
{
  "choices": [
    {
      "finish_reason": "length",
      "index": 0,
      "logprobs": null,
      "text": " there was a kind old"
    },
    {
      "finish_reason": "length",
      "index": 1,
      "logprobs": null,
      "text": ", there was a king"
    }
  ],
  "created": 1674597690,
  "id": "cmpl-6cLe6CssOiH4AYcLPz8eFyy53apdR",
  "model": "text-davinci-003",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 10,
    "prompt_tokens": 4,
    "total_tokens": 14
  }
}
```


5.9. Использование параметра *best_of*

Мы можем попросить модель сгенерировать возможные варианты выполнения задачи на стороне сервера и выбрать тот, который с наибольшей вероятностью окажется правильным. Это делается с помощью параметра `best_of`.

При использовании `best_of` нужно также указать число `n`.

Как было сказано ранее, `n` – это количество вариантов вывода модели, которые вы хотите увидеть.

> **Примечание:** убедитесь, что `best_of` больше `n`. Давайте взглянем на следующий пример:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")
```

```
init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=5,
    n=1,
    best_of=2,
)

print(next)
```

5.10. Управляемое ограничение вывода

В большинстве случаев полезно запретить API генерировать лишний текст.

Допустим, мы хотим сгенерировать один абзац и не более. В этом случае мы можем попросить API прекратить завершение текста при появлении новой строки (\n). Это можно сделать с помощью кода, аналогичного приведенному ниже:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=5,
    stop=["\n",],
)

print(next)
```

Параметр остановки может содержать до четырех стоп-слов. Обратите внимание, что модель не будет включать последовательность стоп-слов в результат.

Вот еще один пример:

```
import os
import openai
```

```
def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-003",
    prompt="Once upon a time",
    max_tokens=5,
    stop=["\n", "Story", "End", "Once upon a time"],
)

print(next)
```

5.11. Использование суффикса после вывода текста

Допустим, мы хотим создать словарь Python, содержащий список простых чисел от 0 до 9, который отформатирован, как показано ниже:

```
{
    "primes": [2, 3, 5, 7]
}
```

В этом случае API должен возвращать `2, 3, 5, 7`. Здесь мы можем использовать параметр `suffix`. Этот параметр настраивает суффикс, который следует после завершения вставленного текста.

Давайте рассмотрим два примера, чтобы лучше понять, как это работает.

В первом примере мы сообщаем API, что словарь должен начинаться с символов `{\n\t\"primes\":[`:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
```

```
    model="text-davinci-002",
    prompt= "Write a JSON containing primary numbers between 0 and 9 \n\n{\n\
t\"primes\": [",
)

print(next)
```

API должен вернуть такой текст:

```
1 2, 3, 5, 7]\n}
```

Как видите, API самостоятельно закрывает список и словарь.

Во втором примере мы не хотим, чтобы API закрывал структуру данных, вставляя `]\n}` в конце сгенерированного текста. Здесь пригодится параметр `suffix`:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-002",
    prompt= "Write a JSON containing primary numbers between 0 and 9 \n\n{\n\
t\"primes\": [",
    suffix= "]\n}",
)

print(next)
```

Теперь API должен вернуть:

```
2, 3, 5, 7
```

вместо предыдущего вывода (`2, 3, 5, 7]\n}`).

Это один из случаев, когда полезно использовать суффикс вывода.

5.12. Пример: извлечение ключевых слов

В этом примере мы хотим извлечь ключевые слова из текста. Воспользуемся текстом, который получен из «Википедии» по адресу **https://en.wikipedia.org/wiki/Programming_language_theory**:

```
The first programming language to be invented was Plankalkül, which was
designed by Konrad Zuse in the 1940s, but not publicly known until 1972 (and
not implemented until 1998). The first widely known and successful high-level
programming language was Fortran, developed from 1954 to 1957 by a team of IBM
researchers led by John Backus. The success of FORTRAN led to the formation
of a committee of scientists to develop a "universal" computer language;
the result of their effort was ALGOL 58. Separately, John McCarthy of MIT
developed Lisp, the first language with origins in academia to be successful.
With the success of these initial efforts, programming languages became an
active topic of research in the 1960s and beyond.
```

В этом примере мы будем использовать такой код:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()
prompt = "The first programming language to be invented was Plankalkül, which
was designed by Konrad Zuse in the 1940s, but not publicly known until 1972
(and not implemented until 1998). The first widely known and successful high-
level programming language was Fortran, developed from 1954 to 1957 by a
team of IBM researchers led by John Backus. The success of FORTRAN led to
the formation of a committee of scientists to develop a "universal" computer
language; the result of their effort was ALGOL 58. Separately, John McCarthy
of MIT developed Lisp, the first language with origins in academia to be
successful. With the success of these initial efforts, programming languages
became an active topic of research in the 1960s and beyond\n\nKeywords:"

keywords = openai.Completion.create(
   model="text-davinci-002",
   prompt=prompt,
   temperature=0.5,
   max_tokens=300,
)

print(keywords)
```

Запрос здесь выглядит следующим образом (в сокращенном виде):

```
The first programming language to be invented was Plankalkül
[...]
[...]
in the 1960s and beyond

Keywords:
```

Благодаря добавлению `Keywords:` в новую строку запроса модель понимает, что нам нужны ключевые слова, и вывод будет выглядеть примерно так:

```
programming language, Plankalkül, Konrad Zuse, FORTRAN, John Backus, ALGOL 58,
John McCarthy, MIT, Lisp
```

Вы можете поэкспериментировать с запросом и попробовать разные варианты, например:

```
The first programming language to be invented was Plankalkül
[...]
[...]
in the 1960s and beyond

Keywords:
-
```

Фактически запрос будет таким:

```
prompt = "The first programming language to be invented was Plankalkül, which
was designed by Konrad Zuse in the 1940s, but not publicly known until 1972
(and not implemented until 1998). The first widely known and successful high-
level programming language was Fortran, developed from 1954 to 1957 by a
team of IBM researchers led by John Backus. The success of FORTRAN led to
the formation of a committee of scientists to develop a "universal" computer
language; the result of their effort was ALGOL 58. Separately, John McCarthy
of MIT developed Lisp, the first language with origins in academia to be
successful. With the success of these initial efforts, programming languages
became an active topic of research in the 1960s and beyond\n\nKeywords:\n-"
```

В этом случае мы должны получить результат, который будет оформлен немного иначе:

```
- Plankalkül
- Konrad Zuse
- FORTRAN
- John Backus
- IBM
- ALGOL 58
- John McCarthy
- MIT
- Lisp
```

5.13. Пример: генерация твитов

Мы продолжим использовать предыдущий пример, но вместо `Keywords` подставим слово `Tweet`.

```
import os
import openai

def init_api():
```

```
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()
prompt = "The first programming language to be invented was Plankalkül, which
was designed by Konrad Zuse in the 1940s, but not publicly known until 1972
(and not implemented until 1998). The first widely known and successful high-
level programming language was Fortran, developed from 1954 to 1957 by a
team of IBM researchers led by John Backus. The success of FORTRAN led to
the formation of a committee of scientists to develop a "universal" computer
language; the result of their effort was ALGOL 58. Separately, John McCarthy
of MIT developed Lisp, the first language with origins in academia to be
successful. With the success of these initial efforts, programming languages
became an active topic of research in the 1960s and beyond\n\nTweet:"

tweet = openai.Completion.create(
    model="text-davinci-002",
    prompt=prompt,
    temperature=0.5,
    max_tokens=300,
)

print(tweet)
```

Запрос (в сокращенном виде) будет выглядеть так:

```
The first programming language to be invented was Plankalkül
[...]
[...]
in the 1960s and beyond

Tweet:
```

Должен получиться такой вывод:

```
The first programming language was Plankalk\u00fcl, invented by Konrad Zuse in
the 1940s.
```

Мы также можем создать твит и извлечь хештеги, используя модифицированный запрос, как показано в следующем примере:

```
The first programming language to be invented was Plankalkül
[...]
[...]
in the 1960s and beyond

Tweet with hashtags:
```

Вот как будет выглядеть код:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()
prompt = "The first programming language to be invented was Plankalkül, which
was designed by Konrad Zuse in the 1940s, but not publicly known until 1972
(and not implemented until 1998). The first widely known and successful high-
level programming language was Fortran, developed from 1954 to 1957 by a
team of IBM researchers led by John Backus. The success of FORTRAN led to
the formation of a committee of scientists to develop a "universal" computer
language; the result of their effort was ALGOL 58. Separately, John McCarthy
of MIT developed Lisp, the first language with origins in academia to be
successful. With the success of these initial efforts, programming languages
became an active topic of research in the 1960s and beyond\n\nTweet with
hashtags:"

tweet = openai.Completion.create(
   model="text-davinci-002",
   prompt=prompt,
   temperature=0.5,
   max_tokens=300,
)

print(tweet)
```

Вот как должен выглядеть результат:

```
#Plankalkül was the first #programming language, invented by Konrad Zuse in the
1940s. #Fortran, developed by John Backus and IBM in the 1950s, was the first
widely known and successful high-level programming language. #Lisp, developed
by John McCarthy of MIT in the 1960s, was the first language with origins in
academia to be successful.
```

Вы можете поэкспериментировать с `max_tokens`, чтобы изменить длину твита, но помните, что:

- 100 токенов ≈ 75 слов;
- твит не должен быть длиннее 280 символов;
- средняя длина слова на английском языке составляет 4.7 символа.

Специальный инструмент-токенизатор[1] поможет вам понять, как API токенизирует фрагмент текста и вычисляет общее количество токенов в нем.

5.14. Пример: сочинение песни в стиле рэп

В этом примере вы увидим, как с помощью модели можно сочинить песню в стиле рэп. Вы можете повторно использовать тот же пример для создания любого другого типа текста. Давайте посмотрим, как это делается:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

my_song = openai.Completion.create(
        prompt="Write a rap song:\n\n",
        max_tokens=200,
        temperature=0.5,
)

print(my_song.choices[0]["text"].strip())
```

Мы можем напрямую получить доступ к тексту и удалить специальные символы при помощи `my_song.choices[0]["text"].strip()`, чтобы распечатать только песню:

```
I was born in the ghetto
(Я родом из негритянском гетто)
Raised in the hood
(Вечно с капюшоном на голове)
My momma didn't have much
(Моя мама была беднячкой)
But she did the best she could
(Но трудилась из последних сил)
I never had much growing up
(Я никогда не был крепок телом)
But I always had heart
(Но я всегда был силен духом)
I knew I was gonna make it
(Я знал, что у меня получится)
```

[1] **https://platform.openai.com/tokenizer.**

```
I was gonna be a star
(И вот я стал звездой)
Now I'm on top of the world
(Теперь я стою на вершине мира)
And I'm never looking back
(И никогда не оглядываюсь)
```

Вы можете экспериментировать с «креативностью» модели и другими параметрами, тестировать, настраивать и пробовать разные комбинации.

5.15. Пример: составление списка дел

В этом примере мы попросим модель составить список дел для создания компании в США. Нам нужен список из пяти пунктов.

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
    model="text-davinci-002",
    prompt="Todo list to create a company in US\n\n1.",
    temperature=0.3,
    max_tokens=64,
    top_p=0.1,
    frequency_penalty=0,
    presence_penalty=0.5,
    stop=["6."],
)

print(next)
```

Отформатированный текст должен быть примерно таким:

```
1. Choose a business structure.
2. Register your business with the state.
3. Obtain a federal tax ID number.
4. Open a business bank account.
5. Establish business credit.
```

В нашем коде мы использовали следующие параметры:

```
model="text-davinci-002"
prompt="Todo list to create a company in US\n\n1."
temperature=0.3
max_tokens=64
top_p=0.1
frequency_penalty=0
presence_penalty=0.5
stop=["6."]
```

Давайте еще раз рассмотрим их по порядку:

- `model`: указывает модель, которую API должен использовать для генерации завершения текста. В данном случае используется `text-davinci-002`;
- `prompt`: это текст, который API использует в качестве отправной точки для создания своего текста. В нашем случае мы использовали подсказку, представляющую собой список дел для создания компании в США. Первый элемент должен начинаться с «`1.`», и нам нужно, чтобы модель придерживалась вывода в формате нумерованного списка

  ```
  1. <1-й пункт>
  2. <2-й пункт>
  3. <3-й пункт>
  4. <4-й пункт>
  5. <5-й пункт>;
  ```

- `temperature` определяет «креативность» текста, генерируемого моделью. Чем выше значение `temperature`, тем креативнее и разнообразнее будет сгенерированный текст. С другой стороны, более низкая температура приведет к более «консервативному» и предсказуемому завершению. В нашем случае температура настроена на `0.3`;
- `max_tokens` ограничивает максимальное количество токенов, которые будет генерировать API. В нашем случае максимальное количество токенов равно 64. Вы можете увеличить это значение, но имейте в виду, что чем больше токенов вы сгенерируете, тем больше денег спишут с вашего баланса. При обучении и тестировании использование более низкого значения поможет вам избежать перерасхода средств;
- `top_p` задает долю от общего числа вариантов ответа, которую API учитывает при генерации следующего токена. Более высокое значение приведет к более консервативному ответу модели, в то время как более низкое значение приведет к более разнообразному ответу. В данном случае `top_p` устанавливается равным `0.1`. Не рекомендуется варьировать одновременно и `top_p`, и `temperature`, но в любом случае это не станет фатальной проблемой;

- `frequency_penalty` используется для настройки поведения модели относительно частоты повторения уже существующих слов. Положительное значение уменьшит вероятность частого повторения слов, а отрицательное значение увеличит ее. В данном случае для параметра `frequency_penalty` установлено значение `0`;
- `presence_penalty` используется для настройки поведения модели относительно генерации конкретных слов или фраз, которые присутствуют или отсутствуют в подсказке. Положительное значение уменьшит шансы слов, присутствующих в подсказке, отрицательное значение увеличит их. В нашем примере для параметра `presence_penalty` установлено значение `0.5`;
- `stop` применяется для указания последовательности токенов, после которой API должен прекратить генерировать текст. В нашем примере, поскольку нам нужно только `5` элементов, мы должны прекратить генерацию после создания токена `6`.

5.16. Заключение

OpenAI Completions API – это мощный инструмент для генерации текста в различных контекстах. При правильных параметрах и настройках он может создавать естественно звучащий текст, соответствующий задаче.

Настроив правильные значения для некоторых параметров, таких как штрафы за частоту и присутствие, можно добиться практически идеального соответствия результатов своим потребностям.

Имея возможность остановить генерацию текста в нужный момент, пользователь также может задавать нужную длину сгенерированного текста. Эта опция также пригодится для уменьшения количества генерируемых токенов и косвенного снижения затрат.

Глава 6

Редактирование текста с помощью GPT

После получения текстового запроса и набора инструкций выбранная вами модель GPT с помощью своих алгоритмов сгенерирует модифицированную версию исходного текста.

В зависимости от ваших инструкций эта модифицированная версия может быть более длинной и/или более подробной, чем первоначальный текст.

Модель GPT способна понимать контекст запроса и предоставленных инструкций, что позволяет ей определить, какие дополнительные сведения было бы наиболее полезно включить в вывод.

6.1. Пример: перевод текста

Мы продолжаем использовать среду разработки под названием `chatgpt-forpythondevelopers`. Начнем с запуска среды:

```
workon chatgptforpythondevelopers
```

Обратите внимание, что в текущем каталоге, где находится файл Python, всегда должен присутствовать файл `.env`, который содержит ваш личный ключ API и номер организации.

Теперь создайте файл `app.py`, содержащий следующий код (или воспользуйтесь готовым файлом из архива книги, внеся в него необходимые изменения):

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
```

```
    openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Edit.create(
    model="text-davinci-edit-001",
    input="Hallo Welt",
    instruction="Translate to English",
)

print(response)
```

В приведенном выше коде мы используем прежнюю функцию для аутентификации:

```
import os
import openai

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()
```

Затем даем API команду перевести немецкий текст на английский:

```
response = openai.Edit.create(
    model="text-davinci-edit-001",
    input="Hallo Welt",
    instruction="Translate to English",
)
```

После выполнения этого примера кода вы должны увидеть следующий вывод:

```
{
  "choices": [
    {
      "index": 0,
      "text": "Hello World!\n"
    }
  ],
  "created": 1674653013,
  "object": "edit",
  "usage": {
    "completion_tokens": 18,
    "prompt_tokens": 20,
```

```
    "total_tokens": 38
  }
}
```

API вернул единственный вариант `Hello World!\\n` с индексом `0`, как показано в выходных данных выше. В отличие генерации текста, где мы предоставляем API начальную подсказку, нам нужно передать команду `instruction` и исходный ввод `input`.

6.2. Инструкция нужна, но ввод необязателен

Важно отметить, что при передаче запроса необходимо заполнить поле `instruction`, но передавать отдельно значение поля `input` не обязательно.

Можно заполнить только поле `instruction` и передать данные вместе с командой:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Edit.create(
    model="text-davinci-edit-001",
    instruction="Translate the following sentence to English: 'Hallo Welt'\
    ",
)

print(response)
```

6.3. Использование конечных точек *completions* и *edits*

Некоторые задачи, предназначенные для конечной точки `edits` (задачи редактирования), можно выполнить с помощью уже знакомой вам по главе 5 конечной точки `completions` (задачи завершения). Вам решать, какая из них лучше подходит для ваших нужд.

Вот пример задачи перевода с использованием конечной точки `edits`:

```
import os
import openai
```

```
def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Edit.create(
    model="text-davinci-edit-001",
    instruction="Translate from English to French, Arabic, and Spanish.",
    input="The cat sat on the mat."
)

print(response)
```

А вот та же задача, что и выше, но с использованием конечной точки `completions`:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

next = openai.Completion.create(
   model="text-davinci-003",
   prompt="""
   Translate the following sentence from English to French, Arabic, and Spanish.
   English: The cat sat on the mat.
   French:
   Arabic:
   Spanish:
   """,
   max_tokens=60,
   temperature=0
)

    print(next)
```

И наоборот, конечную точку edits можно использовать для завершения текста:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Edit.create(
   model="text-davinci-edit-001",
   instruction="Complete the story",
   input="Once upon a time",
)

print(response['choices'][0]['text'])
```

6.4. Форматирование вывода

Рассмотрим такой пример: мы просим конечную точку edits добавить комментарии к коду Golang.

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Edit.create(
   model="text-davinci-edit-001",
   instruction="Explain the following Golang code:",
   input="""
package main

import (
```

```
    "io/ioutil"
    "log"
    "net/http"
)

func main() {
    resp, err := http.Get("https://website.com")

    if err != nil {
        log.Fatalln(err)
    }

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Fatalln(err)
    }

    sb := string(body)
    log.Printf(sb)
}
    """

)

print(response['choices'][0]['text'])
```

К сожалению, мы получаем совершенно нечитаемый вывод:

```
{
  "choices": [
    {
      "index": 0,
      "text": "\npackage main\n\nimport (\n\t// code to work with input and
output\n\t\"io/ioutil\"\n\n\t// code to work with logging\n\t\"log\"\n\n\t//
code to work with http\n\t\"net/http\"\n)\n\nfunc main() {\n resp, err :=
http.Get(\"https://website.com\")\n if err != nil {\n log.Fatalln(err)\n }\n\n
body, err := ioutil.ReadAll(resp.Body)\n if err != nil {\n log.Fatalln(err)\n }
\n\n sb := string(body)\n log.Printf(sb)\n} \n \n"
    }
  ],
  "created": 1674765343,
  "object": "edit",
  "usage": {
    "completion_tokens": 467,
    "prompt_tokens": 162,
    "total_tokens": 629
  }
}
```

Однако, если вы выведете на печать только текст при помощи следующих изменений в коде, он будет правильно отформатирован.

```
import os
import openai

[..код без изменений..]

print(response["choices"][0]["text"])
```

Модифицированный код вернет нам следующий текстовый вывод :

```
//Main File - this will connect to the webpagge, download the file and print
out contents package main

import (
   //Import io for string and log for any problems I may have with connection
   "io/ioutil"
   "log"
   "net/http"
)

func main() {
   //http.Get will connect to the provided webpage
   resp, err := http.Get("https://website.com")
   if err != nil {
     log.Fatalln(err)
   }
   //once webpagge is connected body, err will download the webpage

   body, err := ioutil.ReadAll(resp.Body)
   if err != nil {
      log.Fatalln(err)

   //once I have the body of the webpage then body, err will download the webpage
   }

   //once download is finished, sb will print out the contents of the website
   sb := string(body)
   log.Printf(sb)
}
```

6.5. Креативность или определенность?

Как и в случае с конечной точкой `completions` из главы 5, мы можем управлять креативностью результата с помощью параметра `temperature`.

В следующем примере мы используем два разных значения параметра `temperature`, чтобы увидеть разницу в выходе:

```
import os
import openai

def init_api():
   with open(".env") as env:
```

```
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

response_1 = openai.Edit.create(
    model="text-davinci-edit-001",
    instruction="correct the spelling mistakes:",
    input="The kuick brown fox jumps over the lazy dog and",
    temperature=0,
)

response_2 = openai.Edit.create(
    model="text-davinci-edit-001",
    instruction="correct the spelling mistakes:",
    input="The kuick brown fox jumps over the lazy dog and",
    temperature=0.9,
)

print("Temperature 0:")
print(response_1['choices'][0]['text'])
print("Temperature 0.9:")
print(response_2['choices'][0]['text'])
```

Как правило, после многократного запуска этого кода вы можете заметить, что первый вывод остается неизменным, а второй меняется от одного запуска к другому. Для такого варианта использования, как исправление опечаток, нам обычно не требуется креативность, поэтому достаточно установить для параметра `temperature` значение `0`.

Если нужно, чтобы вывод модели не был уныло предсказуемым, параметр креативности должен быть больше `0`. Вот пример большего простора для творчества:

```
import os
import openai

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()
```

```
response = openai.Edit.create(
   model="text-davinci-edit-001",
   input="Exercise is good for your health.",
   instruction="Edit the text to make it longer.",
   temperature=0.9,
)

print(response["choices"][0]["text"])
```

Так выглядит вывод этого кода:

```
Exercise is good for your health. Especially if you haven't done any for a month.
```

Вот еще один вариант вывода:

```
Exercise is good for your health.
It will help you improve your health and mood.
It is important for integrating your mind and body.
```

Удивительно (а может быть, и нет), это результат, который я получил с параметром `temperature=0`:

```
Exercise is good for your health.
Exercise is good for your health.
Exercise is good for your health.
Exercise is good for your health.
Exercise is good for your health.
Exercise is good for your health.
...
...
```

Еще один способ проявить больше творчества – использовать параметр `top_np`.

Это альтернатива параметру `temperature`, поэтому не рекомендуется использовать одновременно и `temperature`, и `top_p`:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Edit.create(
```

```
    model="text-davinci-edit-001",
    input="Exercise is good for your health.",
    instruction="Edit the text to make it longer.",
    top_p=0.1,
)

print(response["choices"][0]["text"])
```

В приведенном выше примере я использовал `top_p=0.1`, и это означает, что модель будет учитывать только токены, составляющие вероятностную долю 0.1. Другими словами, при формировании результата применяются только токены, составляющие верхние 10 % наиболее вероятных вариантов.

6.6. Создание нескольких правок

Во всех предыдущих примерах у нас был только один результат редактирования. Однако, используя параметр `n`, можно получить больше результатов. Просто укажите количество правок, которое вы хотите получить:

```
import os
import openai

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Edit.create(
    model="text-davinci-edit-001",
    input="Exercise is good for your health.",
    instruction="Edit the text to make it longer.",
    top_p=0.2,
    n=2,
)

print(response["choices"][0]["text"])
print(response["choices"][1]["text"])
```

В данном примере я использовал `n=2`, чтобы получить два результата.

Я также использовал `top_p=0.2`. Однако этот параметр не влияет на количество результатов; мне просто хотелось иметь *более широкий спектр* результатов.

Глава 7

Примеры более сложной работы с текстом

До сих пор мы использовали конечные точки `edits` и `completions` по отдельности. Давайте рассмотрим примеры совместного их использования, чтобы лучше понять различные возможности, предлагаемые моделью.

7.1. Последовательное использование *completions* и *edits*

В этом примере мы попросим модель сгенерировать твит из текста, а затем перевести его. За основу запроса мы возьмем фрагмент текста из «Википедии», уже знакомый вам по предыдущим примерам (этот текст есть в файловом архиве книги).

В первой задаче мы используем конечную точку `completions`, чтобы получить текст твита, за ним следует код запроса на перевод этого твита:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")
init_api()

prompt = "The first programming language to be invented was Plankalkül, which
was designed by Konrad Zuse in the 1940s, but not publicly known until 1972
(and not implemented until 1998). The first widely known and successful high-
level programming language was Fortran, developed from 1954 to 1957 by a
team of IBM researchers led by John Backus. The success of FORTRAN led to
the formation of a committee of scientists to develop a "universal" computer
language; the result of their effort was ALGOL 58. Separately, John McCarthy
```

```
of MIT developed Lisp, the first language with origins in academia to be
successful. With the success of these initial efforts, programming languages
became an active topic of research in the 1960s and beyond.\n\nTwee\t with
hashtags:"

english_tweet = openai.Completion.create(
   model="text-davinci-002",
   prompt=prompt,
   temperature=0.5,
   max_tokens=20,
)

english_tweet_text = english_tweet["choices"][0]["text"].strip()
print("English Tweet:")
print(english_tweet_text)

spanish_tweet = openai.Edit.create(
   model="text-davinci-edit-001",
   input=english_tweet_text,
   instruction="Translate to Spanish",
   temperature=0.5,
)

spanish_tweet_text = spanish_tweet["choices"][0]["text"].strip()
print("Spanish Tweet:")
print(spanish_tweet_text)
```

Выполнив приведенный выше код, мы получим два твита на двух разных языках:

```
English Tweet:
The #first #programming #language to be invented was #Plankalkül
Spanish Tweet:
El primer lenguaje de programación inventado fue #Plankalkül
```

Обратите внимание, что мы использовали функцию `strip()` для удаления начальных и конечных пробелов.

7.2. Apple – это компания или фрукт?

Теперь мы создадим код, который сообщает нам, является ли слово существительным или прилагательным. Очевидная проблема понимания контекста возникает, когда мы передаем модели такие неоднозначные слова, как, например, английское слово *light*. Это слово может быть существительным, прилагательным или глаголом:

- The light is red (свет красный),
- This desk is very light (этот стол очень легкий),
- You light up my life (ты делаешь мою жизнь ярче).

Существует множество других слов, которые могут одновременно употребляться как существительные, прилагательные или глаголы, так что этот пример относится и к ним тоже.

Теперь попробуем отправить модели простой запрос при помощи следующего кода (показан только фрагмент, относящийся к запросу):

```
prompt = "Determine the part of speech of the word 'light'.\n\n"

result = openai.Completion.create(
   model="text-davinci-002",
   prompt=prompt,
   max_tokens=20,
   temperature=1,
)

print(result.choices[0]["text"].strip())
```

Попробуйте запустить этот код несколько раз, и вы увидите, что иногда модель распознает слово light как глагол (`verb`), иногда как прилагательное (`adjective`), иногда как существительное (`noun`), а иногда отвечает более развернуто: `The word 'light' can be used as a noun, adjective, or verb` (слово light может использоваться как существительное, прилагательное или глагол).

Используя контекст, мы можем влиять на реакцию модели. Контекст – это подсказка, данная модели, чтобы провести ее через определенные пользователем шаблоны.

Давайте повторим запрос к модели, дав ей несколько подсказок. Подсказкой может быть что угодно, что поможет модели понять контекст.

Например:

```
prompt_a = "The light is red. Determine the part of speech of the word
'light'.\n\n"
prompt_b = "This desk is very light. Determine the part of speech of the word
'light'.\n\n"
prompt_c = "You light up my life. Determine the part of speech of the word
'light'.\n\n"
for prompt in [prompt_a, prompt_b, prompt_c]:
   result = openai.Completion.create(
      model="text-davinci-002",
      prompt=prompt,
      max_tokens=20,
      temperature=0,
)

print(result.choices[0]["text"].strip())
```

Лучшее понимание контекста приводит к такому результату:

```
noun (существительное)
adjective (прилагательное)
verb (глагол)
```

В следующем примере мы даем модели две разные подсказки. В первом случае модель должна понять, что Apple – это компания, а во втором – что слово Apple обозначает фрукт (яблоко).

```
# Первый пример контекста
prompt = "Huawei:\ncompany\n\nGoogle:\ncompany\n\nMicrosoft:\ncompany\n\
nApple:\n"

# Второй пример контекста
prompt = "Huawei:\ncompany\n\nGoogle:\ncompany\n\nMicrosoft:\ncompany\n\
nApricot:\nFruit\n\nApple:\n"

result = openai.Completion.create(
   model="text-davinci-002",
   prompt=prompt,
   max_tokens=20,
   temperature=0,
   stop=["\n", " "],
)

print(result.choices[0]["text"].strip())
```

7.3. Получение информации о криптовалюте на основе пользовательской схемы (наполнение контекста)

Давайте рассмотрим еще один пример, где мы предоставляем схему или шаблон, которому модель должна следовать в выходных данных.

Наша цель – получить некоторую информацию об определенной криптовалюте, включая ее сокращенное обозначение, дату создания, страницу Coingecko, а также максимум и минимум стоимости за все время.

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

prompt = """
Input: Bitcoin
Output:
BTC was created in 2008, you can learn more about it here: https://bitcoin.
org/en/ and get the latest price here: https://www.coingecko.com/en/coins/
bitcoin.
It's all-time high is $64,895.00 and it's all-time low is $67.81.

Input: Ethereum
Output:
ETH was created in 2015, you can learn more about it here: https://ethereum.
org/en/ and get the latest price here: https://www.coingecko.com/en/coins/
ethereum
It's all-time high is $4,379.00 and it's all-time low is $0.43.

Input: Dogecoin
Output:
DOGE was created in 2013, you can learn more about it here: https://dogecoin.
com/ and get the latest price here: https://www.coingecko.com/en/coins/
dogecoin
It's all-time high is $0.73 and it's all-time low is $0.000002.

Input: Cardano
Output:\n"""
```

```
result = openai.Completion.create(
   model="text-davinci-002",
   prompt=prompt,
   max_tokens=200,
   temperature=0,
)

print(result.choices[0]["text"].strip())
```

В этом коде мы сначала показываем модели примеры того, что она должна возвращать:

```
Input: Bitcoin
Output:
BTC was created in 2008, you can learn more about it here: https://bitcoin.
org/en/ and get the latest price here: https://www.coingecko.com/en/coins/
bitcoin.
It's all-time high is $64,895.00 and it's all-time low is $67.81.

Input: Ethereum
Output:
ETH was created in 2015, you can learn more about it here: https://ethereum.
org/en/ and get the latest price here: https://www.coingecko.com/en/coins/
ethereum
It's all-time high is $4,379.00 and it's all-time low is $0.43.

Input: Dogecoin
Output:
DOGE was created in 2013, you can learn more about it here: https://dogecoin.
com/ and get the latest price here: https://www.coingecko.com/en/coins/
dogecoin
It's all-time high is $0.73 and it's all-time low is $0.000002.
```

Затем мы вызываем конечную точку. Вы можете изменить формат вывода в соответствии с вашими потребностями. Допустим, если вам нужен вывод в формате HTML, вы можете добавить теги HTML к ответам.

Например, так:

```
Input: Bitcoin
Output:
BTC was created in 2008, you can learn more about it <a href="https://bitcoin.
org/en/">here</a> and get the latest price <a href="https://www.coingecko.com/
en/coins/bit\ coin">here</a>.
It's all-time high is $64,895.00 and it's all-time low is $67.81.
```

Модель вернет результат с HTML-разметкой:

```
Cardano was created in 2015, you can learn more about it <a href="https://www.
cardano.org/en/home/">here</a> and get the latest price <a href="https://www.
coingecko.com/en/coins/cardano">here</a>.
It's all-time high is $1.33 and it's all-time low is $0.000019. Let's make it
reusable with other cryptocurrencies:
```

7.4. Создание помощника чат-бота для помощи с командами Linux

Предупреждение:
этот раздел создан на основе моделей OpenAI от 2020 г.

Наша цель – разработать инструмент командной строки, который поможет нам познакомиться с командами Linux посредством диалога.

Начнем с этого примера:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

prompt = """
Input: List all the files in the current directory
Output: ls -l
Input: List all the files in the current directory, including hidden files
Output: ls -la

Input: Delete all the files in the current directory
Output: rm *

Input: Count the number of occurrences of the word "sun" in the file "test.txt"
Output: grep -o "sun" test.txt | wc -l

Input:{}
Output:
"""

result = openai.Completion.create(
   model="text-davinci-002",
   prompt=prompt.format("Count the number of files in the current directory"),
   max_tokens=200,
   temperature=0,
)

print(result.choices[0]["text"].strip())
```

Мы ожидаем от модели только простые однозначные ответы, поэтому используем параметр `temperature=0`. Мы предоставляем модели достаточное количество токенов для обработки вывода (`max_tokens=200`).

После выполнения кода модель должна выдать следующий ответ:

```
ls -l | wc -l
```

Далее мы воспользуемся пакетом Python click[1] (CLI Creation Kit) для создания интерфейса командной строки с минимальным кодом. Это сделает нашу программу более интерактивной.

Начнем с активации виртуальной среды разработки и установки пакета Python:

```
workon chatgptforpythondevelopers
pip install click==8.1.3
```

Затем создадим новую версию файла `app.py` (в файловом архиве книги этот файл сохранен под именем `app2.py`):

```
import os
import openai
import click

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

_prompt = """
Input: List all the files in the current directory
Output: ls -l

Input: List all the files in the current directory, including hidden files
Output: ls -la

Input: Delete all the files in the current directory
Output: rm *

Input: Count the number of occurrences of the word "sun" in the file "test.txt"
Output: grep -o "sun" test.txt | wc -l

Input: {}
Output:"""
```

[1] **https://click.palletsprojects.com/.**

```
while True:
    request = input(click.style("Input", fg="green"))
    prompt = _prompt.format(request)
    result = openai.Completion.create(
        model="text-davinci-002",
        prompt=prompt,
        temperature=0.0,
        max_tokens=100,
        stop=["\n"],
    )

    command = result.choices[0].text.strip()
    click.echo(click.style("Output: ", fg="yellow") + command)
    click.echo()
```

Мы используем прежний запрос. Единственное важное изменение, которое мы сделали, – это добавление вызова `click` в код внутри бесконечного цикла `while`. При выполнении кода наша программа запросит данные у пользователя (`request`), затем вставит их в запрос и передаст в API.

В программе `click` также отвечает за печать результата. Последний вызов `click.echo()` напечатает пустую строку.

```
$ python app.py

Input: list all
Output: ls

Input: delete all
Output: rm -r *

Input: count all files
Output: ls -l | wc -l

Input: count all directories
Output: find . -type d | wc -l

Input: count all files that are not directories
Output: find . ! -type d | wc -l
```

Добавим также команду выхода из программы (показана жирным шрифтом):

```
import os
import openai
import click

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value
```

```
    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

_prompt = """
Input: List all the files in the current directory
Output: ls -l

Input: List all the files in the current directory, including hidden files
Output: ls -la

Input: Delete all the files in the current directory
Output: rm *

Input: Count the number of occurrences of the word "sun" in the file "test.txt"
Output: grep -o "sun" test.txt | wc -l

Input: {}
Output:"""

while True:
    request = input(click.style("Input", fg="green"))
    if request == "exit":
        break

    prompt = _prompt.format(request)

    result = openai.Completion.create(
        model="text-davinci-002",
        prompt=prompt,
        temperature=0.0,
        max_tokens=100,
        stop=["\n"],
    )

    command = result.choices[0].text.strip()
    click.echo(click.style("Output: ", fg="yellow") + command)
    click.echo()
```

Наконец, добавим блок кода, выполняющий команду, которую сгенерировала модель:

```
import os
import openai
import click

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
```

```
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

_prompt = """
Input: List all the files in the current directory
Output: ls -l

Input: List all the files in the current directory, including hidden files
Output: ls -la

Input: Delete all the files in the current directory
Output: rm *

Input: Count the number of occurrences of the word "sun" in the file "test.txt"
Output: grep -o "sun" test.txt | wc -l

Input: {}
Output:"""

while True:
    request = input(click.style("Input", fg="green"))
    if request == "exit":
        break

    prompt = _prompt.format(request)

    result = openai.Completion.create(
        model="text-davinci-002",
        prompt=prompt,
        temperature=0.0,
        max_tokens=100,
        stop=["\n"],
    )

    command = result.choices[0].text.strip()
    click.echo(click.style("Output: ", fg="yellow") + command)

    click.echo(click.style("Execute? (y/n): ", fg="yellow"), nl=False)
    choice = input()
    if choice == "y":
        os.system(command)
    elif choice == "n":
        continue
    else:
        click.echo(click.style("Invalid choice. Please enter 'y' or 'n'.", fg="red"))

    click.echo()
```

Теперь, если вы введете в окне командной строки команду `python app.py`, вы увидите вопрос, хотите ли вы выполнить предложенную моделью команду:

```
Input (type 'exit' to quit): list all files in /tmp
Output: ls /tmp
Execute? (y/n): y
<здесь будет список файлов в каталоге /tmp/ >

Input (type 'exit' to quit): list all files in /tmp
Output: ls /tmp
Execute? (y/n): cool
Invalid choice. Please enter 'y' or 'n'.
```

Мы также можем добавить блок `try…except` для перехвата любых возможных исключений:

```
import os
import openai
import click

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

_prompt = """
Input: List all the files in the current directory
Output: ls -l

Input: List all the files in the current directory, including hidden files
Output: ls -la

Input: Delete all the files in the current directory
Output: rm *

Input: Count the number of occurrences of the word "sun" in the file "test.txt"
Output: grep -o "sun" test.txt | wc -l

Input: {}
Output:"""

while True:
   request = input(click.style("Input", fg="green"))
   if request == "exit":
```

```
            break

        prompt = _prompt.format(request)

        result = openai.Completion.create(
            model="text-davinci-002",
            prompt=prompt,
            temperature=0.0,
            max_tokens=100,
            stop=["\n"],
        )

        command = result.choices[0].text.strip()
        click.echo(click.style("Output: ", fg="yellow") + command)

        click.echo(click.style("Execute? (y/n): ", fg="yellow"), nl=False)
        choice = input()
        if choice == "y":
            os.system(command)
        elif choice == "n":
            continue
        else:
            click.echo(click.style("Invalid choice. Please enter 'y' or 'n'.",
fg="red"))

          click.echo()
        except Exception as e:
            click.echo(click.style("The command could not be executed. {}".
format(e), fg="red"))
        pass

        click.echo()
```

Глава 8

Встраивание

8.1. Что такое встраивание

Если бы меня попросили описать эту функцию в одном предложении, я бы сказал, что *встраивание текста* (text embedding) OpenAI измеряет, насколько две текстовые строки похожи друг на друга.

Встраивания, как правило, часто используются для таких задач, как поиск наиболее релевантных результатов для поискового запроса, группировка текстовых строк на основе их сходства, рекомендация элементов с похожими текстовыми строками, поиск текстовых строк, которые сильно отличаются от других, анализ того, насколько текстовые строки отличаются друг от друга, и маркировка текстовых строк на основе того, что они больше всего похожи.

С практической точки зрения встраивание – это способ представления объектов и отношений реального мира в виде вектора (списка чисел). Одно и то же векторное пространство используется для измерения того, насколько похожи две сущности. Чем ближе друг к другу расположены два вектора в этом пространстве, тем больше схожи между собой две сущности.

8.2. Примеры использования

Текстовые встраивания OpenAI измеряют взаимосвязь текстовых строк и могут использоваться для различных целей.

Вот некоторые варианты использования:

- задачи обработки естественного языка (NLP), такие как анализ эмоциональной окраски, семантическое сходство и классификация эмоций;
- создание встраиваний признаков текста для задач машинного обучения, таких как сопоставление ключевых слов, классификация документов и моделирование тем;
- создание независимых от языка представлений текста, позволяющих сравнивать текстовые строки на разных языках;
- повышение точности систем текстового поиска и систем понимания естественного языка;

- создание персонализированных рекомендаций путем сравнения введенного пользователем текста с широким диапазоном текстовых строк.

Более кратко можно резюмировать варианты использования следующим образом:

- **поиск**: результаты ранжируются по релевантности строке запроса;
- **кластеризация**: текстовые строки группируются по сходству;
- **рекомендации**: подборка элементов со связанными текстовыми строками;
- **обнаружение аномалий**: выявление малосвязанных выбросов;
- **измерение разнообразия**: анализ распределения подобия;
- **классификация**: текстовые строки классифицируются по их наиболее похожей метке.

Ниже приведено несколько примеров использования встраиваний на практике (не обязательно в продуктах OpenAI).

8.2.1. *Tesla*

Работать с неструктурированными данными может быть непросто: необработанный текст, изображения и видео не всегда упрощают создание моделей с нуля. Часто это связано с тем, что получить исходные данные сложно из-за ограничений конфиденциальности, а для создания хороших моделей могут потребоваться большие вычислительные ресурсы, огромный набор данных и время.

Встраивание – это способ взять информацию из одного контекста (например, изображение автомобиля) и использовать ее в другом контексте (например, в игре). Этот прием называется *переносом обучения*, и он помогает нам обучать качественные модели без использования большого количества реальных данных.

Tesla использует эту методику в своих беспилотных автомобилях.

8.2.2. *Kalendar AI*

Kalendar AI – это продукт для развития продаж, в котором используются встраивания, чтобы автоматически сопоставить правильное предложение о продажах с нужными клиентами из набора данных из 340 млн профилей.

Автоматизация основана на сходстве между встроенными профилями клиентов и торговыми предложениями и выявлении наиболее подходящих совпадений. Согласно данным OpenAI это уменьшило рассылку нежелательных предложений на 40–56 % по сравнению с предыдущими методами.

8.2.3. *Notion*

Notion – это онлайн-инструмент для организации рабочего пространства. Он улучшил свои возможности поиска, используя функциональность

встраиваний OpenAI. Поиск в этом случае выходит за рамки простых систем сопоставления ключевых слов, которые инструмент использовал до настоящего времени.

Новая функция интеллектуального поиска позволяет Notion лучше понимать структуру, контекст и значение контента, хранящегося на его платформе, позволяя пользователям выполнять более точный поиск и быстрее находить документы.

8.2.4. *DALL·E 2*

DALL·E 2 – это система, которая преобразует текстовые метки в изображения.

Она использует в своей работе две модели под названием Prior и Decoder. Prior берет текстовые метки и создает встраивания изображений CLIP[1]. В свою очередь, Decoder берет встраивания изображений CLIP и создает изображение. Затем изображение масштабируется с размера 64×64 до 1024×1024.

8.3. Подготовка к работе

Для экспериментов со встраиванием текста необходимо установить `datalib` с помощью следующей команды:

```
pip install datalib
```

В дальнейшем нам понадобятся Matplotlib и другие библиотеки:

```
pip install matplotlib plotly scipy scikit-learn
```

Убедитесь, что вы устанавливаете их в правильной виртуальной среде разработки. Этот пакет также установит такие инструменты, как pandas и NumPy.

Все упомянутые библиотеки являются наиболее часто применяются в области ИИ и науки о данных в целом:

- pandas – это быстрый, мощный, гибкий и простой в использовании инструмент для анализа и обработки данных с открытым исходным кодом, созданный на основе Python;
- NumPy – еще одна библиотека Python, добавляющая поддержку больших многомерных массивов и матриц, а также большой набор высокоуровневых математических функций для работы с этими массивами;
- Matplotlib – это библиотека построения графиков для языка программирования Python и его расширения для числовой математики NumPy;

[1] **https://openai.com/blog/clip/.**

- plotly.py – это интерактивная графическая библиотека с открытым исходным кодом для Python Sparkle;
- SciPy – это бесплатная библиотека Python с открытым исходным кодом, используемая для научных и технических вычислений.

8.4. Знакомство со встраиванием текста

Начнем с простого примера:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Embedding.create(
   model="text-embedding-ada-002",
   input="I am a programmer",
)

print(response)
```

Как обычно, мы импортируем `openai`, проходим аутентификацию и вызываем конечную точку. Однако на этот раз мы используем модель Ada, которая является единственной моделью, доступной в OpenAI для встраивания. Команда OpenAI рекомендует использовать модель `text-embedding-ada-002` почти во всех случаях, поскольку она «лучше, дешевле и проще в использовании».

Вывод должен быть относительно длинным и выглядеть следующим образом:

```
{
  "data": [
    {
      "embedding": [
        -0.0169205479323864,
        -0.019740639254450798,
        -0.011300412937998772,
        -0.016452759504318237,
         [..]
        0.003966170828789473,
```

```
          -0.011714739724993706
        ],
        "index": 0,
        "object": "embedding"
      }
    ],
    "model": "text-embedding-ada-002-v2",
    "object": "list",
    "usage": {
      "prompt_tokens": 4,
      "total_tokens": 4
    }
  }
```

Мы можем получить доступ к встраиванию:

```
print(response["embedding"])
```

Написанная нами программа выводит список чисел с плавающей запятой наподобие `0.010284645482897758` и `0.013211660087108612`.

Эти числа представляют собой встраивание входного текста `I am a programmer` (я программист), сгенерированного моделью OpenAI `text-embedding-ada-002`.

Встраивание – это представление входного текста в многомерном пространстве смыслов, которое отражает его значение (встраивание вектора смыслов в пространство, отсюда и происходит термин). Иногда его называют векторным представлением или просто вектором встраивания.

Встраивание можно рассматривать как способ представления объекта, например текста, с использованием достаточно большого набора значений. Каждое значение представляет определенный аспект объекта и значимость этого аспекта для данного конкретного объекта. В случае текста аспекты могут представлять темы, эмоциональную окраску или другие семантические характеристики текста.

Другими словами, здесь вам нужно понять, что векторное представление, сгенерированное конечной точкой `embeddings`, представляет собой способ представления данных в формате, понятном моделям и алгоритмам машинного обучения. Это способ взять определенный ввод и преобразовать его в форму, которую можно использовать в этих моделях и алгоритмах.

Дальше мы рассмотрим различные способы использования встраиваний.

8.5. Встраивания для нескольких вводов

В последнем примере мы использовали один ввод:

```
    input="I am a programmer",
```

но можно использовать несколько входов, и вот как это делается:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

response = openai.Embedding.create(
   model="text-embedding-ada-002",
   input=["I am a programmer", "I am a writer"],
)

for data in response["data"]:
   print(data["embedding"])
```

Важно отметить, что длина каждого ввода не должна превышать 8192 токена.

8.6. Семантический поиск

Далее реализуем семантический поиск с использованием встраивания OpenAI. Это базовый пример, но мы рассмотрим и более сложные примеры.

Начнем с аутентификации:

```
import openai
import os
import pandas as pd
import numpy as np

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()
```

Затем нужно создать файл с названием `words.csv` (его можно взять в файловом архиве книги). Файл CSV содержит один столбец `text`, состоящий из случайных слов:

```
text
apple
banana
cherry
dog
cat
house
car
tree
phone
computer
television
book
music
food
water
sky
air
sun
moon
star
ocean
desk
bed
sofa
lamp
carpet
window
door
floor
ceiling
wall
clock
watch
ewelry
ring
necklace
bracelet
earring
wallet
key
photo
```

Pandas – чрезвычайно мощный инструмент, когда дело доходит до манипулирования данными, включая данные в файлах CSV. Это именно то, что нам нужно. Давайте применим pandas для чтения файла и создания объекта pandas `DataFrame`.

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

`DataFrame` – наиболее часто используемый объект pandas.

Официальная документация Pandas описывает `DataFrame` как двумерную помеченную структуру данных со столбцами потенциально разных типов. Мы можем рассматривать этот объект как своего рода электронную таблицу, таблицу SQL или набор объектов `Series`.

Если мы отправим `df` на печать, то увидим такой вывод:

```
      text
0        apple
1       banana
2       cherry
3          dog
4          cat
5        house
6          car
7         tree
8        phone
9     computer
10 television
11        book
12       music
13        food
14       water
15         sky
16         air
17         sun
18        moon
19        star
20       ocean
21        desk
22         bed
23        sofa
24        lamp
25      carpet
26      window
27        door
28       floor
29     ceiling
30        wall
31       clock
32       watch
33     jewelry
34        ring
35    necklace
36    bracelet
37     earring
38      wallet
39         key
40       photo
```

Далее мы получим встраивание для каждого элемента в кадре данных. Для этого мы будем использовать не функцию `openai.Embedding.create()`, а

функцию `get_embedding()`. Обе они делают одно и то же, но первая возвращает JSON, включая встраивания и другие данные, а вторая – список встраиваний. Поэтому вторая более практична для использования с `DataFrame`.

Эта функция работает следующим образом:

```
get_embedding("Hello", engine='text-embedding-ada-002')
# Вернет [-0.02499537356197834, -0.019351257011294365, ... и т.д.]
```

Мы также собираемся использовать метод `apply`, который есть у каждого объекта `DataFrame`. Этот метод применяет функцию к оси кадра данных.

```
# Импорт функции для получения встраивания
from openai.embeddings_utils import get_embedding

# Получение встраивания для каждого слова в кадре данных
df['embedding'] = df['text'].apply(lambda x: get_embedding(x, engine='text-
embedding-ada-002'))
```

Теперь у нас есть кадр данных с двумя осями: одна из них текст, а другая – встраивания. Последняя содержит встраивания для каждого слова на первой оси.

Давайте сохраним кадр данных в другой файл CSV:

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

Новый файл должен выглядеть так, как показано на рис. 8.1. Он содержит три столбца: идентификатор, текст и встраивания.

Рис. 8.1. Содержимое файла *embeddings.csv*

Давайте теперь прочитаем новый файл и преобразуем последний столбец в массив numpy. Почему?

Потому что на следующем шаге мы будем использовать функцию `cosine_similarity`. Эта функция ожидает массив numpy, хотя по умолчанию это строка.

Но почему бы просто не использовать обычный массив или список Python?

На самом деле массивы numpy широко используются в объемных вычислениях. Модуль обычного списка Python не предоставляет никакой помощи в таких вычислениях. Кроме того, массив numpy потребляет меньше памяти и работает быстрее. Это связано с тем, что массив представляет собой набор однородных типов данных, которые хранятся в смежных областях памяти, тогда как список в Python представляет собой набор разнородных типов данных, хранящихся в несмежных областях памяти.

Вернемся к нашему коду и преобразуем последний столбец в пустой массив:

```
df['embedding'] = df['embedding'].apply(eval).apply(np.array)
```

Теперь мы запросим у пользователя ввод, прочитаем его и выполним семантический поиск, используя сходство `cosine_similarity`:

```
# Получение поискового запроса от пользователя
user_search = input('Enter a search term: ')

# Получение встраивания для поискового запроса
user_search_embedding = get_embedding(user_search, engine='text-embedding-
ada-002')

# Импорт функции для вычисления косинусного сходства
from openai.embeddings_utils import cosine_similarity

# Вычисление косинусного сходства между запросом и словами в кадре данных
df['similarity'] = df['embedding'].apply(lambda x: cosine_similarity(x, user_
search_embedding))
```

Давайте разберемся, что делают три последние операции в приведенном выше коде.

1. `user_search_embedding = get_embedding(user_search, engine='text-embedding-ada-002')`

   Эта строка кода использует функцию `get_embedding`, чтобы получить встраивание для заданного пользователем условия поиска `user_search`. Параметр `engine` имеет значение `text-embedding-ada-002`, которое определяет используемую модель встраивания текста OpenAI.

2. `from openai.embeddings_utils import cosine_similarity`

   Эта строка кода импортирует функцию `cosine_similarity` из модуля `openai.embeddings_utils`. Функция `cosine_similarity` вычисляет косинусное сходство между двумя встраиваниями.

3. `df['similarity'] = df['embedding'].apply(lambda x: cosine_similarity(x, user_search_ embedding))`

   Эта строка кода создает новый столбец в кадре данных с именем `'similarity'` и использует метод `apply` с лямбда-функцией для вычисления косинусного сходства между встраиванием пользовательского поискового запроса и встраиванием каждого слова в кадре данных.

Косинусное сходство между каждой парой встраиваний сохраняется в новом столбце `similarity`. Полный код выглядит так (он есть в файловом архиве):

```
import openai
import os
import pandas as pd
import numpy as np
from openai.embeddings_utils import get_embedding
from openai.embeddings_utils import cosine_similarity

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

# words.csv это csv-файл со столбцом 'text', который содержит слова
df = pd.read_csv('words.csv')

# Получаем встраивания для каждого слова в кадре данных
df['embedding'] = df['text'].apply(lambda x: get_embedding(x, engine='text-
embedding-ada-002'))

# Сохраняем встраивания в другой csv-файл
df.to_csv('embeddings.csv')

# Читаем новый csv-файл
df = pd.read_csv('embeddings.csv')

# Преобразовываем ось встраиваний в массив питру
df['embedding'] = df['embedding'].apply(eval).apply(np.array)

# Получаем поисковый запрос от пользователя
user_search = input('Enter a search term: ')

# Получаем встраивание для поискового запроса
search_term_embedding = get_embedding(user_search, engine='text-embedding-
ada-002')
```

```
# Вычисляем косинусное подобие между поисковым запросом и каждым словом кадра
# данных
df['similarity'] = df['embedding'].apply(lambda x: cosine_similarity(x, search_
term_embedding))

print(df)
```

После запуска кода я ввел запрос office и получил такой результат:

```
Unnamed: 0        text                                          embedding  similarity
0          0       apple  [0.0077999732457101345, -0.02301608957350254, ...    0.830448
1          1      banana  [-0.013975119218230247, -0.03290277719497681, ...    0.805773
2          2      cherry  [0.006462729535996914, -0.018950263038277626, ...    0.792155
3          3         dog  [-0.0033353185281157494, -0.017689190804958344...    0.828782
4          4         cat  [-0.0070945825427770615, -0.017328109592199326...    0.802046
5          5       house  [-0.007152134086936712, 0.007141574751585722, ...    0.874455
6          6         car  [-0.0074789817444980145, -0.021566664800047874...    0.821671
7          7        tree  [-0.0047506773844361305, -0.013216584920883179...    0.825486
8          8       phone  [-0.0014101049164310098, -0.022890757769346237...    0.853214
9          9    computer  [-0.003125436371192336, -0.014225165359675884,...    0.861776
10        10  television  [-0.004810569807887077, -0.019971350207924843,...    0.799359
11        11        book  [-0.006842561066150665, -0.019114654511213303,...    0.838213
12        12       music  [-0.0018855653470382094, -0.023304970934987068...    0.820804
13        13        food  [0.022285157814621925, -0.026815656572580338, ...    0.831213
14        14       water  [0.019031280651688576, -0.01257743313908577, 0...    0.816956
15        15         sky  [0.004940779879689217, -0.0014005625853314996,...    0.818631
16        16         air  [0.008958300575613976, -0.02343503013253212, -...    0.801587
17        17         sun  [0.024797989055514336, -0.0025757758412510157,...    0.816029
18        18        moon  [0.017512451857328415, -0.009135404601693153, ...    0.801957
19        19        star  [0.011644366197288036, -0.009548380970954895, ...    0.812378
20        20       ocean  [0.0049842894077301025, 0.0002579695428721607,...    0.797868
21        21        desk  [0.01278653647750616, -0.02077387273311615, -0...    0.889902
22        22         bed  [0.005934594664722681, 0.004146586172282696, 0...    0.823993
23        23        sofa  [0.011793493293225765, -0.011562381871044636, ...    0.814385
24        24        lamp  [0.006820859387516975, -0.008771182037889957, ...    0.834019
25        25      carpet  [0.009344781748950481, -0.013140080496668816, ...    0.802807
26        26      window  [0.007339522708207369, -0.01684434525668621, 0...    0.829965
27        27        door  [-0.004870024509727955, -0.026941418647766113,...    0.833316
28        28       floor  [0.018742715939879417, -0.021140681579709053, ...    0.861481
29        29     ceiling  [-0.01695505902171135, -0.00972691923379898, -...    0.813539
30        30        wall  [0.0010972798336297274, 0.014719482511281967, ...    0.829286
31        31       clock  [-0.011117076501250267, -0.013727957382798195,...    0.815507
32        32       watch  [-0.0024846186861395836, -0.010468898341059685...    0.818415
33        33     jewelry  [-0.016019975766539574, 0.010300415568053722, ...    0.783474
34        34        ring  [-0.02060825377702713, -0.025675412267446518, ...    0.819233
35        35    necklace  [-0.024919956922531128, 0.0024241949431598186,...    0.777729
36        36    bracelet  [-0.03430960699915886, 0.005157631356269121, -...    0.779868
37        37     earring  [-0.025862164795398712, -0.009202365763485432,...    0.776563
38        38      wallet  [0.015366269275546074, -0.020114824175834656, ...    0.825882
39        39         key  [0.003653161460533738, -0.02867439016699791, 0...    0.815625
40        40       photo  [0.004279852379113436, -0.03139236196875572, -...    0.833338
```

Используя ось `similarity`, мы можем увидеть, какое слово семантически похоже на `office`. Чем выше значение сходства, тем больше похоже слово из столбца `text`.

Такие слова, как `necklace` (ожерелье), `bracelet` (браслет) и `earring` (серьга) имеют оценку 0.77, однако слово `desk` (стол) имеет оценку 0.88.

Чтобы сделать результат более читаемым, мы можем отсортировать кадр данных по оси сходства:

```
# сортировка кадра данных по оси сходства
df = df.sort_values(by='similarity', ascending=False)
```

Мы также можем получить 10 лучших сходств, используя такую строку кода:

```
df.head(10)
```

Давайте посмотрим на окончательный код:

```
import openai
import os
import pandas as pd
import numpy as np
from openai.embeddings_utils import get_embedding
from openai.embeddings_utils import cosine_similarity

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

# words.csv это csv-файл со столбцом 'text', который содержит слова
df = pd.read_csv('words.csv')

# Получаем встраивания для каждого слова в кадре данных
df['embedding'] = df['text'].apply(lambda x: get_embedding(x, engine='text-
embedding-ada-002'))

# Сохраняем встраивания в другой csv-файл
df.to_csv('embeddings.csv')

# Читаем новый csv-файл
df = pd.read_csv('embeddings.csv')

# Преобразовываем ось встраиваний в массив питру
df['embedding'] = df['embedding'].apply(eval).apply(np.array)
```

```
# Получаем поисковый запрос от пользователя
user_search = input('Enter a search term: ')

# Получаем встраивание для поискового запроса
search_term_embedding = get_embedding(user_search, engine='text-embedding-ada-002')

# Вычисляем косинусное подобие между поисковым запросом и каждым словом кадра
# данных
df['similarity'] = df['embedding'].apply(lambda x: cosine_similarity(x,
search_term_embedding))

# Сортируем кадр данных по оси сходства
df = df.sort_values(by='similarity', ascending=False)

# Выводим на печать первые 10 результатов
print(df.head(10))
```

Вот первые 10 наиболее близких по смыслу слов:

```
Unnamed: 0      text                                          embedding  similarity
21      21      desk  [0.012774260714650154, -0.020844005048274994, ...    0.890026
5        5     house  [-0.007152134086936712, 0.007141574751585722, ...    0.874455
9        9  computer  [-0.0031794828828424215, -0.014211298897862434...    0.861704
28      28     floor  [0.018742715939879417, -0.021140681579709053, ...    0.861481
8        8     phone  [-0.0014101049164310098, -0.022890757769346237...    0.853214
11      11      book  [-0.006843345705419779, -0.019184302538633347,...    0.838138
24      24      lamp  [0.006820859387516975, -0.008771182037889957, ...    0.834019
27      27      door  [-0.004844364244490862, -0.026875808835029602,...    0.833317
40      40     photo  [0.004297871608287096, -0.03132128715515137, -...    0.833313
13      13      food  [0.022335752844810486, -0.02753201313316822, -...    0.831723
```

Вы можете попробовать ввести другие слова (например, `dog`, `hat`, `phone`, `fashion`, `philosophy` и т. д.) и увидеть соответствующие результаты.

8.7. Косинусное сходство

Мы вычисляли взаимную схожесть различных слов с помощью так называемого *косинусного сходства* (cosine similarity). Чтобы использовать его в программах, не обязательно разбираться в математике, но, если вы хотите глубже погрузиться в тему, прочтите этот раздел. Но даже если вы его пропустите, это не повлияет на ваши навыки использования API OpenAI для создания интеллектуальных приложений.

Косинусное сходство – это математический способ измерения того, насколько похожи два вектора. Фактически это нахождение косинуса угла между двумя векторами (линиями) в многомерном пространстве. Результатом является число от –1 до 1. Если векторы полностью одинаковы, результат равен 1. Если векторы совершенно разные, результат равен –1. Если векторы находятся под углом 90°, результат равен 0. В математике это уравнение записывают так:

$$Сходство = (A.B) / (||A||.||B||).$$

- A и B – векторы;
- A.B – это способ перемножения двух числовых множеств. Это делается путем умножения каждого элемента из одного множества на соответствующий элемент из другого множества, а затем суммирования этих произведений;
- ||A|| – длина вектора A. Она вычисляется путем извлечения квадратного корня из суммы квадратов каждого элемента вектора A.

Возьмем векторы A = [2, 3, 5, 2, 6, 7, 9, 2, 3, 4] и вектор B = [3, 6, 3, 1, 0, 9, 2, 3, 4, 5]. Вот как можно вычислить косинусное сходство между ними, используя Python:

```
# Импортируем numpy и norm из numpy.linalg
import numpy as np
from numpy.linalg import norm

# Задаем два вектора
A = np.array([2,3,5,2,6,7,9,2,3,4])
B = np.array([3,6,3,1,0,9,2,3,4,5])

# Выводим векторы на печать
print("Vector A: {}".format(A))
print("Vector B: {}".format(B))

# Вычисляем косинусное подобие
cosine = np.dot(A,B)/(norm(A)*norm(B))

# Выводим результат на печать
print("Косинусное подобие между A и B: {}".format(cosine))
```

Мы можем сделать то же самое, используя библиотеку Python Scipy:

```
import numpy as np
from scipy import spatial

# Задаем два вектора
A = np.array([2,3,5,2,6,7,9,2,3,4])
B = np.array([3,6,3,1,0,9,2,3,4,5])

# Выводим векторы на печать
print("Vector A: {}".format(A))
print("Vector B: {}".format(B))

# Вычисляем косинусное подобие
cosine = 1 - spatial.distance.cosine(A, B)

# Выводим результат на печать
print("Косинусное подобие между A и B: {}".format(cosine))
```

Или с помощью Scikit-Learn:

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Задаем два вектора
A = np.array([2,3,5,2,6,7,9,2,3,4])
B = np.array([3,6,3,1,0,9,2,3,4,5])

# Выводим векторы на печать
print("Vector A: {}".format(A))
print("Vector B: {}".format(B))

# Вычисляем косинусное подобие
cosine = cosine_similarity([A],[B])

# Выводим результат на печать
print("Косинусное подобие: {}".format(cosine[0][0]))
```

Глава 9

Более сложные примеры встраивания

9.1. Предсказание вашего любимого сорта кофе

Сейчас наша цель – научиться рекомендовать пользователю лучшую смесь кофейных зерен на основе его отзывов. Например, пользователь вводит *Ethiopia Dumerso*, и программа обнаруживает, что *Ethiopia Dumerso*, *Ethiopia Guji Natural Dasaya* и *Organic Dulce de Guatemala* являются наиболее похожими смесями. Выходные данные будут содержать названия этих трех смесей.

Нам понадобится набор данных, который можно скачать на Kaggle. Перейдите на Kaggle и загрузите набор данных с именем `simple_coffee.csv`[1]. (Вам нужно будет создать учетную запись.)

Набор данных имеет 1267 строк (смесей) и 9 признаков:

- *name* (название смеси кофейных зерен),
- *roaster* (имя обжарщика),
- *roast* (тип обжарки),
- *loc_country* (страна обжарщика),
- *origin* (происхождение зерен),
- *100g_USD* (цена за 100 г в долл. США),
- *rating* (рейтинг, максимум 100),
- *review_date* (дата обзора),
- *review* (текст обзора).

Что нас интересует в этом наборе данных, так это отзывы пользователей. Эти обзоры были взяты с сайта **www.coffeereview.com**.

Когда пользователь вводит название кофе, мы вызываем конечную точку Embeddings API OpenAI, чтобы получить встраивание для текста обзора этого кофе (подразумевается, что пользователь вводит название, которое есть в наборе данных). Затем мы вычисляем косинусное сходство между отзывом о выбранном кофе и всеми остальными отзывами в наборе данных. Отзывы с наивысшими показателями косинусного сходства будут наиболее похожи на обзор входного кофе. Затем мы выводим на печать

[1] **https://www.kaggle.com/datasets/schmoyote/coffee-reviews-dataset?select=simplified_coffee.csv**.

названия сортов кофе, у которых описания наиболее похожи на описание сорта, указанного пользователем.

Теперь создадим это приложение шаг за шагом.

Активируйте виртуальную среду разработки и установите пакет *nltk*:

```
pip install nltk
```

Natural Language Toolkit, или просто NLTK, представляет собой набор библиотек и программ для символической и статистической обработки естественного английского языка, написанных на Python. Скоро вы увидите, как он применяется.

Теперь, используя свой терминал командной строки, введите `python`, чтобы войти в интерпретатор Python. Затем введите следующие команды:

```
import nltk
nltk.download('stopwords')
nltk.download('punkt')
```

NLTK поставляется с текстовыми корпусами, демонстрационными грамматиками, обученными моделями и т. д. Все, что нам нужно для данного примера, – это вышеупомянутые `stopwords` и `punkt`. Если вы хотите загрузить все компоненты, используйте команду `nltk.download('all')`. Полный список корпусов можно найти по адресу **https://www.nltk.org/nltk_data/**.

Теперь создадим три функции:

```
import os
import pandas as pd
import numpy as np
import nltk
import openai
from openai.embeddings_utils import get_embedding
from openai.embeddings_utils import cosine_similarity

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

def download_nltk_data():
    try:
        nltk.data.find('tokenizers/punkt')
    except LookupError:
        nltk.download('punkt')
    try:
        nltk.data.find('corpora/stopwords')
    except LookupError:
```

```
        nltk.download('stopwords')

    def preprocess_review(review):
        from nltk.corpus import stopwords
        from nltk.stem import PorterStemmer
        stopwords = set(stopwords.words('english'))
        stemmer = PorterStemmer()
        tokens = nltk.word_tokenize(review.lower())
        tokens = [token for token in tokens if token not in stopwords]
        tokens = [stemmer.stem(token) for token in tokens]
        return ' '.join(tokens)
```

Функция `init_api` считывает ключ API и идентификатор организации из файла `.env` и устанавливает переменные среды.

Функция `download_nltk_data` загружает корпуса `punkt` и `stopwords`, если они еще не загружены.

Функция `preprocess_review` переводит символы в нижний регистр, токенизирует, удаляет стоп-слова и выделяет текст обзора.

Затем мы разбиваем отзыв на отдельные слова с помощью функции `nltk.word_tokenize()`.

Далее мы удаляем из обзора стоп-слова (распространенные слова, такие как «the», «a», «and» и т. д.), используя список стоп-слов, полученный из корпуса `stopwords` в NLTK с помощью функции `stopwords`.

Наконец, мы применяем к тексту стеммер Портера из NLTK с помощью функции `nltk.stem.PorterStemmer()`. *Стемминг* (stemming) – это процесс приведения слова к его корневой форме. Например, слова «бегун», «бегать» и «бежать» имеют один и тот же корень «бег». Это важный этап, так как он помогает нам уменьшить количество уникальных слов в тексте отзыва, оптимизировать производительность нашей модели за счет уменьшения количества параметров и снизить затраты, связанные с вызовами API.

Затем мы собираем слова в корневой форме обратно в единую строку, используя метод `.join()` для строк Python. Теперь у нас есть окончательный предварительно обработанный текст обзора, который мы будем использовать для создания встраивания.

Далее добавим следующий код:

```
init_api()
download_nltk_data()
```

Этот код инициализирует API OpenAI и загрузит необходимые данные NLTK с помощью уже определенной функции. В основном функция проверяет, не загрузили ли вы необходимые данные вручную раньше, и загружает их при необходимости.

Затем нам нужно прочитать пользовательский ввод:

```
# Читаем пользовательский ввод
input_coffee_name = input("Введите название кофе: ")
```

Далее мы загрузим CSV-файл в объект DataFrame Pandas. Обратите внимание, что здесь мы читаем только первые 50 строк. Вы можете изменить этот код и удалить параметр nrows, чтобы загрузить весь набор данных CSV.

```
# Читаем csv-файл в DataFrame (только первые 50 строк для
# ускорения работы примера и снижения расходов на оплату API)
df = pd.read_csv('simplified_coffee.csv', nrows=50)
```

Затем выполним предварительную обработку всех текстов обзоров:

```
# Предварительная обработка обзоров: нижний регистр, токенизация,
# удаление стоп-слов и стемминг
df['preprocessed_review'] = df['review'].apply(preprocess_review)
```

Теперь нам нужно получить встраивания для каждого обзора:

```
# Получаем встраивание для каждого обзора
review_embeddings = []
for review in df['preprocessed_review']:
    review_embeddings.append(get_embedding(review, engine='text-embedding-
ada-002'))
```

Затем мы получаем индекс названия кофе, введенного пользователем. Если такого названия кофе нет в нашей базе данных, завершаем работу программы:

```
# Получаем индекс названия кофе, введенного пользователем
try:
   input_coffee_index = df[df['name'] == input_coffee_name].index[0]
except:
   print("Извините, такого названия кофе нет в нашей базе данных.
   Попробуйте еще раз.")
   exit()
```

Строка input_coffee_index = df[df['name'] == input_coffee_name].index[0] использует метод Pandas df[df['name'] == input_coffee_name] для получения строки из DataFrame, которая содержит введенное название кофе.

Например, df[df['my_column'] == my_value] выбирает такие строки из кадра данных df, для которых значение в столбце my_column равно my_value.

Этот код возвращает новый объект DataFrame, содержащий только те строки, которые соответствуют этому условию. Результирующий кадр данных имеет те же столбцы, что и исходный df, но содержит только те строки, которые соответствуют условию.

Например, если df – это объект DataFrame с отзывами о кофе (как в нашем случае), а my_column – это имя столбца, то условие df[df['name'] == 'Ethiopia Yirgacheffe'] вернет новый DataFrame, который содержит только отзывы о кофе *Ethiopia Yirgacheffe*.

Затем мы использовали index[0], чтобы получить индекс этой строки.

Метод `index[0]` используется для получения первого индекса результирующего отфильтрованного `DataFrame`, возвращенного условием `df[df['name'] == input_coffee_name]`.

Вот краткий обзор того, что делает строка `input_coffee_index = df[df['name'] == input_coffee_name].index[0]`.

1. `df['name'] == input_coffee_name` создает логическую маску, которая имеет значение `True` для строк, в которых столбец `name` равен `input_coffee_name`, и `False` для всех остальных строк.
2. `df[df['name'] == input_coffee_name]` использует эту логическую маску для фильтрации `DataFrame` и возвращает новый `DataFrame`, содержащий только строки, в которых столбец `name` равен `input_coffee_name`.
3. `df[df['name'] == input_coffee_name].index` возвращает метки индекса результирующего отфильтрованного `DataFrame`.
4. `index[0]` извлекает первую метку индекса из результирующих меток индекса. Поскольку отфильтрованный `DataFrame` содержит только одну строку, это метка индекса для этой строки.

Далее мы посчитаем косинусное сходство между отзывом на кофе, название которого ввел пользователь, и всеми остальными отзывами:

```
# Вычисляем косинусное сходство между обзором кофе, который выбрал
# пользователь, и всеми остальными обзорами
similarities = []
input_review_embedding = review_embeddings[input_coffee_index]
for review_embedding in review_embeddings:
    similarity = cosine_similarity(input_review_embedding, review_embedding)
    similarities.append(similarity)
```

Функция `cosine_similarity(input_review_embedding, review_embedding)` использует функцию OpenAI `openai.embeddings_utils.cosine_similarity()` для вычисления косинусного сходства между обзором кофе, который выбрал пользователь, и текущим обзором. (Мы уже использовали эту функцию в предыдущем примере.)

После этого получим индексы наиболее похожих отзывов (исключая обзор исходного кофе):

```
# Получаем индексы наиболее похожих отзывов (исключая обзор исходного кофе)
most_similar_indices = np.argsort(similarities)[-6:-1]
```

Если вы уже использовали библиотеку NumPy и хорошо знакомы с сортировкой аргументов, вы можете пропустить следующее подробное объяснение того, как она работает.

- `np.argsort(similarities)[-6:-1]` использует функцию NumPy `argsort()` для получения индексов пяти отзывов, наиболее похожих отзыв пользовательского кофе. Вот что происходит по шагам:

- `argsort()` возвращает индексы, которые будут сортировать массив сходств в порядке возрастания.

Например, если сходства равны `[0.8, 0.5, 0.9, 0.6, 0.7, 0.4, 0.3, 0.2, 0.1, 0.0]`, функция `np.argsort(similarities)[-6:-1]` будет работать так:

1) `np.argsort(similarities)` вернет отсортированные индексы: `[9, 8, 7, 6, 5, 4, 1, 3, 0, 2]`. Массив сортируется по значениям сходств: `similarities [0] = 0.8`, `similarities [1] = 0.5`, `similarities [2] = 0.9` и т. д.;
2) `np.argsort(similarities)[-6:-1]` вернет индексы с 6-го по 2-й с конца отсортированного массива: `[5, 4, 1, 3, 0]`.

Когда мы вызываем `np.argsort(similarities)`, она возвращает массив индексов, которые сортируют массив сходств в порядке возрастания. Другими словами, первый индекс в отсортированном массиве будет соответствовать элементу сходства с наименьшим значением, а последний индекс в отсортированном массиве будет соответствовать элементу сходства с наибольшим значением.

В примере `[0.8, 0.5, 0.9, 0.6, 0.7, 0.4, 0.3, 0.2, 0.1, 0.0]` индекс наименьшего значения (`0.0`) равен `9`, индекс второго наименьшего значения (`0.1`) равен `8` и т. д. Результирующий отсортированный массив индексов имеет вид `[9, 8, 7, 6, 5, 4, 1, 3, 0, 2]`.

Этот отсортированный массив индексов затем используется для получения индексов пяти наиболее похожих отзывов путем среза массива, чтобы получить элементы с 6-го по 2-й с конца массива: `[5, 4, 1, 3, 0]`. Эти индексы соответствуют наиболее похожим отзывам в порядке убывания сходства.

Вы можете спросить, почему мы не используем `[-5:]`?

Если бы мы использовали `np.argsort(similarities)[-5:]` вместо `np.argsort(similarities)[-6:-1]`, мы бы получили пять наиболее похожих отзывов, включая сам обзор исходного кофе. Причина, по которой мы исключаем обзор исходного кофе из наиболее похожих обзоров – нет смысла рекомендовать кофе, который пользователь уже пробовал. Используя `[-6:-1]`, мы исключаем из среза 1-й элемент, который соответствует обзору исходного кофе.

Другой вопрос, который вы можете задать: почему этот обзор оказался в массиве `similarities`?

Он был добавлен в массив `review_embeddings`, когда мы создавали встраивания для каждого обзора с помощью функции `get_embedding()`.

Далее получим названия наиболее похожих сортов кофе:

```
# Получаем названия наиболее похожих сортов кофе
similar_coffee_names = df.iloc[most_similar_indices]['name'].tolist()
```

`df.iloc[most_similar_indices]['name'].tolist()` использует функцию Pandas `iloc[]` для получения названий наиболее похожих сортов кофе. Если подробнее, это делается так:

`df.iloc[most_similar_indices]` использует `iloc[]` для получения строк `DataFrame`, которые соответствуют наиболее похожим отзывам. Например, если наиболее похожие индексы – `[3, 4, 0, 2]`, то `df.iloc[most_similar_indices]` вернет строки `DataFrame`, соответствующие 4-му, 5-му, 1-му и 3-му наиболее похожим отзывам.

Затем мы используем `['name']` для получения значений столбца `name` из этих строк. Наконец, мы используем `tolist()` для преобразования столбца в список. Это дает нам список названий наиболее похожих сортов кофе.

Теперь можно вывести результаты на печать:

```
# Вывод результатов на печать
print("Сорта кофе, наиболее похожие на {}:".format(input_coffee_name))
for coffee_name in similar_coffee_names:
    print(coffee_name)
```

Окончательный код выглядит так:

```
import os
import pandas as pd
import numpy as np
import nltk
import openai
from openai.embeddings_utils import get_embedding
from openai.embeddings_utils import cosine_similarity

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

def download_nltk_data():
    try:
        nltk.data.find('tokenizers/punkt')
    except LookupError:
        nltk.download('punkt')
    try:
        nltk.data.find('corpora/stopwords')
    except LookupError:
        nltk.download('stopwords')

def preprocess_review(review):
    from nltk.corpus import stopwords
    from nltk.stem import PorterStemmer
    stopwords = set(stopwords.words('english'))
    stemmer = PorterStemmer()
    tokens = nltk.word_tokenize(review.lower())
```

```
    tokens = [token for token in tokens if token not in stopwords]
    tokens = [stemmer.stem(token) for token in tokens]
    return ' '.join(tokens)

init_api()
download_nltk_data()

# Получаем поисковый запрос от пользователя
input_coffee_name = input("Enter a coffee name: ")

# Читаем csv-файл в DataFrame (только первые 50 строк для
# ускорения работы примера и снижения расходов на оплату API)
df = pd.read_csv('simplified_coffee.csv', nrows=50)

# Предварительная обработка обзоров: нижний регистр, токенизация,
# удаление стоп-слов и стемминг
df['preprocessed_review'] = df['review'].apply(preprocess_review)

# Получаем встраивание для каждого обзора
review_embeddings = []
for review in df['preprocessed_review']:
    review_embeddings.append(get_embedding(review, engine='text-embedding-
    ada-002'))

# Получаем индекс названия кофе, введенного пользователем
try:
    input_coffee_index = df[df['name'] == input_coffee_name].index[0]
except:
    print("Извините, такого названия кофе нет в нашей базе данных.
    Попробуйте еще раз.")
    exit()

# Вычисляем косинусное сходство между обзором кофе, который выбрал
# пользователь, и всеми остальными обзорами
similarities = []
input_review_embedding = review_embeddings[input_coffee_index]
for review_embedding in review_embeddings:
    similarity = cosine_similarity(input_review_embedding, review_embedding)
    similarities.append(similarity)

# Получаем индексы наиболее похожих отзывов (исключая обзор исходного кофе)
most_similar_indices = np.argsort(similarities)[-6:-1]

# Получаем названия наиболее похожих сортов кофе
similar_coffee_names = df.iloc[most_similar_indices]['name'].tolist()

# Вывод результатов на печать
print("Сорта кофе, наиболее похожие на {}:".format(input_coffee_name))
for coffee_name in similar_coffee_names:
    print(coffee_name)
```

9.2. Выполняем «нечеткий» поиск

Потенциальная проблема кода из предыдущего раздела заключается в том, что пользователь должен ввести точное название кофе, которое присутствует в наборе данных, например: *Estate Medium Roast*, *Gedeb Ethiopia* и т. д. В реальной жизни такая точность встречается не очень часто. Пользователь может пропустить символ или слово, неправильно ввести название или использовать другой регистр, и это приведет к прекращению поиска с сообщением «*Извините, такого названия кофе нет в нашей базе данных. Попробуйте еще раз*».

Одним из решений является более гибкий поиск. Например, мы можем искать название сорта, которое содержит введенный пользователем текст, игнорируя регистр:

```
# Получаем индекс названия кофе, введенного пользователем
try:
    # Поиск названия кофе, похожего на введенное пользователем
    input_coffee_index = df[df['name'].str.contains(input_coffee_name,
    case=False)].index[0]
    print("Найдено название кофе, похожее на{}. Используем его.".
    format(df.iloc[input_coffee_index]['name']))
except:
    print("Извините, такого названия кофе нет в нашей базе данных.
    Попробуйте еще раз.")
    exit()
```

Если мы воспользуемся приведенным выше кодом и выполним его, мы получим более одного результата для некоторых ключевых слов.

Запустив этот код на наборе данных, который у нас есть, с поиском по ключевому слову *Ethiopia*, мы получим около 390 результатов. Поэтому мы должны обработать встраивание каждого описания этих результатов и сравнить каждое из них со встраиваниями других описаний кофе. Это может быть довольно долго, и мы обязательно получим десятки результатов. В этом случае лучше было бы использовать только три названия из всех результатов, но как их выбрать? Может быть, случайным образом?

Лучшим решением является использование метода *нечеткого поиска*. Например, с помощью Python мы можем использовать метод *расстояния Левенштейна*[1]. Говоря упрощенно, расстояние Левенштейна между двумя словами – это минимальное количество односимвольных правок (вставок, удалений или замен), необходимых для преобразования одного слова в другое. Вам не нужно заново реализовывать какой-либо алгоритм самостоятельно, так как большинство из них можно найти в таких библиотеках, как *textdistance*[2].

Другим решением является вычисление косинусного сходства между пользовательским вводом и названиями кофе.

[1] **https://en.wikipedia.org/wiki/Levenshtein_distance .**

[2] **https://github.com/life4/textdistance .**

```
# Получение индекса для пользовательского названия кофе
try:
    input_coffee_index = df[df['name'] == input_coffee_name].index[0]
except IndexError:
    # Получение встраиваний для каждого названия
    print("Извините, такого названия нет в нашей базе данных. Мы попробуем
    найти наиболее похожее название.")
    name_embeddings = []
    for name in df['name']:
        name_embeddings.append(get_embedding(name, engine='text-embedding-
        ada-002'))
    # Поиск по косинусному подобию среди названий кофе
    input_coffee_embedding = get_embedding(input_coffee_name, engine='text-
    embedding-ada-002')
    _similarities = []
    for name_embedding in name_embeddings:
        _similarities.append(cosine_similarity(input_coffee_embedding,
        name_embedding))
    input_coffee_index = _similarities.index(max(_similarities))
except:
    print("Извините, у нас нет подходящего названия кофе в базе данных.
    Пожалуйста, попробуйте снова")
    exit()
```

Это означает, что у нас будет два поиска по косинусному сходству в одном и том же коде:

- поиск названия кофе, наиболее похожего на пользовательский ввод (`name_embeddings`);
- поиск описания кофе, наиболее похожего на описание сорта, выбранного пользователем (`review_embeddings`).

Мы также можем комбинировать методы нечеткого и косинусного поиска подобия в одном и том же коде.

9.3. Прогнозирование категории новостей с помощью встраивания

В этом примере будет представлен классификатор новостей, который предсказывает категорию новостной статьи. Мы всегда начинаем с уже привычного фрагмента кода:

```
import os
import openai
import pandas as pd
from openai.embeddings_utils import get_embedding
from openai.embeddings_utils import cosine_similarity

def init_api():
```

```
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")
init_api()
```

Затем определяем список категорий:

```
categories = [
    "POLITICS",
    "WELLNESS",
    "ENTERTAINMENT",
    "TRAVEL",
    "STYLE & BEAUTY",
    "PARENTING",
    "HEALTHY LIVING",
    "QUEER VOICES",
    "FOOD & DRINK",
    "BUSINESS",
    "COMEDY",
    "SPORTS",
    "BLACK VOICES",
    "HOME & LIVING",
    "PARENTS",
]
```

Выбор категорий не случаен. В следующем разделе вы узнаете, почему я сделал этот выбор.

Теперь напишем функцию, которая классифицирует предложение как часть одной из вышеуказанных категорий:

```
# Определение функции для классификации предложений
def classify_sentence(sentence):
    # Получение встраивания предложения
    sentence_embedding = get_embedding(sentence, engine="text-embedding-
    ada-002")
    # Вычисление сходства между предложением и каждой категорией
    similarity_scores = {}
    for category in categories:
        category_embeddings = get_embedding(category, engine="text-embedding-
        ada-002")
        similarity_scores[category] = cosine_similarity(sentence_embedding,
        category_embeddings)
    # Возвращаем категорию с наивысшей оценкой сходства
    return max(similarity_scores, key=similarity_scores.get)
```

Вот что делает каждая часть этой функции.

1. `sentence_embedding = get_embedding(sentence, engine="text-embedding-ada-002")`: в этой строке используется функция OpenAI `get_embedding` для встраивания входного предложения. Аргумент `engine="text-embedding-ada-002"` указывает, какую модель OpenAI использовать для встраивания.
2. `category_embeddings = get_embedding(category, engine="text-embedding-ada-002")`: мы получаем встраивание текущей категории внутри цикла `for`.
3. `similarity_scores[category] = cosine_similarity(sentence_embedding, category_embeddings)`: мы вычисляем косинусное сходство между встраиванием предложения и встраиванием категории и сохраняем результат в словаре `similarity_scores`.
4. `return max(similarity_scores, key=similarity_scores.get)`: мы возвращаем категорию с наивысшей оценкой сходства. Функция `max` находит ключ (категорию) с максимальным значением (оценкой сходства) в словаре `similarity_scores`.

Теперь мы можем классифицировать несколько предложений:

```
# Классификация предложений
sentences = [
   "1 dead and 3 injured in El Paso, Texas, mall shooting",
   "Director Owen Kline Calls Funny Pages His 'Self-Critical' Debut",
   "15 spring break ideas for families that want to get away",
   "The US is preparing to send more troops to the Middle East",
   "Bruce Willis' 'condition has progressed' to frontotemporal dementia,
   his family says",
   "Get an inside look at Universal's new Super Nintendo World",
   "Barcelona 2-2 Manchester United: Marcus Rashford shines but Raphinha
   salvages d\ raw for hosts",
   "Chicago bulls win the NBA championship",
   "The new iPhone 12 is now available",
   "Scientists discover a new dinosaur species",
   "The new coronavirus vaccine is now available",
   "The new Star Wars movie is now available",
   "Amazon stock hits a new record high",
]

for sentence in sentences:
   print("{:50} category is {}".format(sentence, classify_sentence(sentence)))
   print()
```

Вот как выглядит окончательный код:

```
import os
import openai
import pandas as pd
from openai.embeddings_utils import get_embedding
from openai.embeddings_utils import cosine_similarity

def init_api():
```

```
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

categories = [
    "POLITICS",
    "WELLNESS",
    "ENTERTAINMENT",
    "TRAVEL",
    "STYLE & BEAUTY",
    "PARENTING",
    "HEALTHY LIVING",
    "QUEER VOICES",
    "FOOD & DRINK",
    "BUSINESS",
    "COMEDY",
    "SPORTS",
    "BLACK VOICES",
    "HOME & LIVING",
    "PARENTS",
]

# Определение функции классификации предложений
def classify_sentence(sentence):
    # Получение встраивания предложения
    sentence_embedding = get_embedding(sentence, engine="text-embedding-ada-002")
    # Вычисление сходства между предложением и каждой категорией
    similarity_scores = {}
    for category in categories:
        category_embeddings = get_embedding(category, engine="text-
        embeddingada-002")
        similarity_scores[category] = cosine_similarity(sentence_embedding,
        category_embeddings)
    # Возвращаем категорию с наивысшей оценкой сходства
    return max(similarity_scores, key=similarity_scores.get)

# Классификация предложений
sentences = [
    "1 dead and 3 injured in El Paso, Texas, mall shooting",
    "Director Owen Kline Calls Funny Pages His 'Self-Critical' Debut",
    "15 spring break ideas for families that want to get away",
    "The US is preparing to send more troops to the Middle East",
    "Bruce Willis' 'condition has progressed' to frontotemporal dementia, his
    family says",
```

```
    "Get an inside look at Universal's new Super Nintendo World",
    "Barcelona 2-2 Manchester United: Marcus Rashford shines but Raphinha
    salvages draw for hosts",
    "Chicago bulls win the NBA championship",
    "The new iPhone 12 is now available",
    "Scientists discover a new dinosaur species",
    "The new coronavirus vaccine is now available",
    "The new Star Wars movie is now available",
    "Amazon stock hits a new record high",
]

for sentence in sentences:
    print("{:50} category is {}".format(sentence, classify_sentence(sentence)))
    print()
```

После выполнения приведенного выше кода вы получите вывод с такой структурой:

```
<Предложение>
<Прогнозируемая категория>

<Предложение>
<Прогнозируемая категория>

<Предложение>
<Прогнозируемая категория>
```

В следующем примере мы будем использовать тот же код.

9.4. Оценка точности классификатора

Похоже, что предыдущий классификатор почти идеален, но есть способ понять, действительно ли он точен, и получить оценку точности.

Мы начнем с загрузки набора данных из Kaggle[1] и сохранения его в `data/News_Category_Dataset_v3.json`.

Этот набор данных содержит около 210 тыс. заголовков новостей с 2012 по 2022 гг. от HuffPost[2]. Набор данных классифицирует заголовки каждой статьи по категориям.

Категории, используемые в наборе данных, такие же, как мы использовали ранее (отсюда и мой первоначальный выбор в предыдущем разделе):

```
POLITICS
WELLNESS
ENTERTAINMENT
TRAVEL
STYLE & BEAUTY
PARENTING
HEALTHY LIVING
```

[1] **https://www.kaggle.com/datasets/rmisra/news-category-dataset .**

[2] **https://www.huffingtonpost.com/ .**

```
QUEER VOICES
FOOD & DRINK
BUSINESS
COMEDY
SPORTS
BLACK VOICES
HOME & LIVING
PARENTS
```

Мы будем использовать функцию `sklearn.metrics.precision_score`, которая вычисляет показатель точности.

Точность (precision) – это отношение `tp / (tp + fp)`, где `tp` – количество истинно положительных (true positive) прогнозов, а `fp` – количество ложноположительных (false positive) прогнозов. Проще говоря, точность – это способность классификатора не маркировать отрицательный образец как положительный.

Посмотрим, как выполняется оценка точности.

```
from sklearn.metrics import precision_score
def evaluate_precision(categories):
    # Загрузка набора данных
    df = pd.read_json("data/News_Category_Dataset_v3.json", lines=True).head(20)
    y_true = []
    y_pred = []

    # Классификация каждого предложения
    for _, row in df.iterrows():
        true_category = row['category']
        predicted_category = classify_sentence(row['headline'])

        y_true.append(true_category)
        y_pred.append(predicted_category)

# Раскомментируйте следующие строки для вывода на печать истинных и ложных прогнозов

# if true_category != predicted_category:
#    print("Ложный прогноз: {:50} Истина: {:20} Прогноз: {:20}".format(row['he\
adline'], true_category, predicted_category))
    # else:
    #    print("Истинный прогноз: {:50} Истина: {:20} Прогноз:{:20}".format(row['hea\
dline'], true_category, predicted_category))

# Вычисление показателя точности
return precision_score(y_true, y_pred, average='micro', labels=categories)
```

Давайте посмотрим, что делает каждая строка.

1. `df = pd.read_json("data/News_Category_Dataset_v3.json", lines=True).head(20)`: эта строка считывает первые 20 записей набора данных `News_Category_Dataset_v3.json` в кадр данных Pandas. Аргумент `lines=True` указыва-

ет, что файл содержит один объект JSON на строку. Для более точного расчета следует использовать более 20 записей. Однако для этого примера я использую только 20 записей, а вам рекомендую извлекать больше записей.

2. `y_true = []` и `y_pred = []`: мы инициализируем пустые списки для хранения истинных и предсказанных категорий для каждого предложения.
3. `for _, row in df.iterrows()`:: здесь мы перебираем каждую строку в DataFrame.
4. `true_category = row['category']` и `predicted_category = classify_sentence (row['headline'])`: эти строки извлекают истинную категорию из текущей строки и используют функцию `classify_sentence` для предсказания категории заголовка в текущей строке.
5. `y_true.append(true_category)` и `y_pred.append(predicted_category)`: мы добавляем истинную и предсказанную категории в списки `y_true` и `y_pred`.
6. `return precision_score(y_true, y_pred, average='micro', labels=categories)`: эта строка вычисляет показатель точности предсказанных категорий, используя функцию `scikit precision_score`. Аргумент `average='micro'` указывает, что точность должна рассчитываться глобально путем подсчета общего количества истинно положительных, ложноотрицательных и ложноположительных результатов. Аргумент `labels=categories` указывает список категорий, используемых для расчета точности.

К слову, вместо `micro` может быть `macro`, `samples`, `weighted` или `binary`. Официальная документация[1] объясняет разницу между этими вариантами среднего значения.

В целом функция `evaluate_precision` загружает небольшое подмножество набора данных `News_Category_Dataset_v3.json`, использует функцию `classify_sentence` для прогнозирования категории каждого заголовка и вычисляет точность прогнозирования категорий. Возвращенная оценка точности представляет собой точность функции `classify_sentence` на этом небольшом подмножестве набора данных.

После того как мы объединим все компоненты, код будет выглядеть так:

```
import os
import openai
import pandas as pd
from openai.embeddings_utils import get_embedding
from openai.embeddings_utils import cosine_similarity
from sklearn.metrics import precision_score

def init_api():
   with open(".env") as env:
```

[1] **https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html.**

```
            for line in env:
                key, value = line.strip().split("=")
                os.environ[key] = value

        openai.api_key = os.environ.get("API_KEY")
        openai.organization = os.environ.get("ORG_ID")

    init_api()

    categories = [
        "POLITICS",
        "WELLNESS",
        "ENTERTAINMENT",
        "TRAVEL",
        "STYLE & BEAUTY",
        "PARENTING",
        "HEALTHY LIVING",
        "QUEER VOICES",
        "FOOD & DRINK",
        "BUSINESS",
        "COMEDY",
        "SPORTS",
        "BLACK VOICES",
        "HOME & LIVING",
        "PARENTS",
    ]

    # Определение функции для классификации предложений
    def classify_sentence(sentence):
        # Получение встраивания предложения
        sentence_embedding = get_embedding(sentence, engine="text-embedding-ada-002")
        # Вычисление сходства между предложением и каждой категорией
        similarity_scores = {}
        for category in categories:
            category_embeddings = get_embedding(category, engine="text-embedding-
            ada-002")
            similarity_scores[category] = cosine_similarity(sentence_embedding,
            category_embeddings)
        # Возвращаем категорию с наивысшей оценкой сходства
        return max(similarity_scores, key=similarity_scores.get)


    def evaluate_precision(categories):
        # Загружаем набор данных
        df = pd.read_json("data/News_Category_Dataset_v3.json", lines=True).head(20)
        y_true = []
        y_pred = []

        # Классифицируем каждое предложение
        for _, row in df.iterrows():
```

```
            true_category = row['category']
            predicted_category = classify_sentence(row['headline'])

            y_true.append(true_category)
            y_pred.append(predicted_category)

    # Вычисляем показатель точности
    return precision_score(y_true, y_pred, average='micro', labels=categories)

precision_evaluated = evaluate_precision(categories)
print("Точность: {:.2f}".format(precision_evaluated))
```

Глава 10

Тонкая настройка и передовые методы работы

10.1. Обучение на ограниченных примерах

Модель GPT-3 была предварительно обучена с использованием огромного объема данных, состоящих из миллиардов слов. Благодаря этому она стала невероятно мощным инструментом, способным быстро осваивать новые задачи всего на нескольких примерах. Этот прием известен как *обучение на ограниченных примерах* (few-shot learning). Он стал революционным достижением в области искусственного интеллекта.

Неудивительно, что появилось множество мощных проектов для обучения моделей с открытым исходным кодом. Для реализации небольших учебных проектов можно использовать несколько надежных библиотек Python:

- Pytorch – Torchmeta[1]: набор расширений и загрузчиков данных для быстрого обучения и метаобучения в PyTorch[2];
- Few Shot[3]: репозиторий с чистым, легко читаемым и проверенным кодом для воспроизведения исследований по обучению на ограниченных примерах;
- FewRel[4]: крупномасштабный набор данных для извлечения нескольких отношений, который содержит более ста отношений и десятки тысяч аннотированных экземпляров данных в различных предметных областях;
- Few-Shot Object Detection (FsDet)[5]: содержит официальную реализацию метода Simple Few-Shot Object Detection[6];

---

1 **https://tristandeleu.github.io/pytorch-meta/**.

2 **https://pytorch.org/**.

3 **https://github.com/oscarknagg/few-shot**.

4 **https://github.com/thunlp/FewRel**.

5 **https://github.com/ucbdrive/few-shot-object-detection**.

6 **https://arxiv.org/abs/2003.06957**.

- Meta Transfer Learning[1]: фреймворк для решения сложных задач обучения на ограниченных примерах. Цель этого проекта состоит в том, чтобы воспользоваться преимуществами нескольких похожих задач с малым количеством примеров, чтобы узнать, как модифицировать базовую обучаемую модель под новую задачу, для которой доступно лишь несколько размеченных примеров;
- прототипные сети в наборе данных Omniglot[2]: реализация метода из статьи *Prototypical Networks for Few-shot Learning* на основе Pytorch;
- …и многие другие.

Возможность обучения всего на нескольких примерах позволяет GPT-3 быстро понимать предоставленные пользователем инструкции даже с минимальными данными. Другими словами, GPT-3 можно запрограммировать на выполнение задач, используя всего несколько примеров в качестве входных данных. Это открывает новый мир безграничных возможностей для приложений, управляемых ИИ.

10.2. Улучшенное обучение с ограниченными примерами

Тонкая настройка (fine tuning) – это логическое продолжение обучения с ограниченными примерами, но на гораздо большем количестве примеров, чем может уместиться в одной подсказке. Этот подход можно использовать для достижения лучших результатов в широком круге задач:

- увеличении количества используемых примеров;
- повышении точности результатов;
- расширении круга выполняемых задач.

После тонкой настройки модели вам больше не нужно приводить обучающие примеры при каждом запросе. Это экономит деньги и делает запросы быстрее.

Тонкая настройка доступна для основных базовых моделей: `davinci`, `curie`, `babbage` и `ada`.

10.3. Тонкая настройка на практике

Теперь перейдем к практической части и рассмотрим реальные примеры.

Этот раздел является лишь введением, а всю основную сложную работу мы будем делать позже. Сейчас вам достаточно понять общие принципы, но потом мы углубимся в более сложный пример.

Начнем с активации виртуальной среды Python для разработки, а затем экспортируем ключ API OpenAI:

[1] **https://github.com/yaoyao-liu/meta-transfer-learning**.

[2] **https://github.com/cnielly/prototype-networks-omniglot**.

```
export OPENAI_API_KEY="<OPENAI_API_KEY>"
```

Если вы используете Windows, можете использовать команду `set`. Теперь создадим файл JSONL. Вот пример такого файла:

```
{"prompt":"When do I have to start the heater?", "completion":"Every day in
the morning at 7AM. You should stop it at 2PM"}
{"prompt":"Where is the garage remote control?", "completion":"Next to the
yellow door, on the key ring"}
{"prompt":"Is it necessary to program the scent diffuser every day?",
"completion":"The scent diffuser is already programmed, you just need to
recharge it when its battery is low"}
```

Файл JSONL, т. е. JSON, в котором разделителем служит символ новой строки, – это удобный формат для хранения структурированных данных, которые можно обрабатывать по одной записи за раз. Наши обучающие данные должны быть в формате JSONL.

Для обучения необходимо предоставить не менее двухсот примеров. Более того, удвоение размера набора данных приводит к линейному улучшению качества модели.

Мы используем всего три строки в нашем файле JSONL, поэтому полученная модель будет работать не очень хорошо, но сейчас мы просто хотим посмотреть, как это работает на практике.

Запустим в окне командной строки утилиту для преобразования файла с обучающими данными. В процессе преобразования нужно будет ответить на несколько вопросов:

```
openai tools fine_tunes.prepare_data -f data.json

Анализ файла...

- Ваш файл JSON имеет формат JSONL. Ваш файл будет преобразован в формат JSONL.
- Ваш файл содержит 3 пары примеров запрос-ответ. В целом, мы рекомендуем иметь
хотя бы несколько сотен примеров. Мы обнаружили, что качество модели склонно к
линейному росту при каждом удвоении количества примеров.
- Все запросы должны заканчиваться суффиксом `?`
- Ваши данные не содержат общего окончания в конце примера ответов. Наличие
общей конечной строки, добавленной к концу ответа, более точно показывает
модели, где должен заканчиваться сгенерированный ответ. См. https://beta.
openai.com/docs/guides/fine-tuning/preparing-your-dataset для более подробной
информации и примеров.
- Ответ должен начинаться с пробела (` `). Это, как правило, дает лучшие
результаты из-за особенностей используемой нами токенизации. Подробнее
см. https://beta.openai.com/docs/guides/fine-tuning/preparing-your-dataset.
```

После анализа содержимого файла вам будут предложены следующие действия:

```
- [Обязательно] Ваш формат `JSON` будет конвертирован в `JSONL`
```

```
- [Рекомендовано] Добавить суффикс конца `\n` ко всем ответам [Y/n]: Y
- [Рекомендовано] Добавить символ пробела в начало каждого ответа [Y/n]: Y

Ваши данные будут записаны в новый файл JSONL. Продолжить [Y/n]: Y
Модифицированный файл записан в `data_prepared.jsonl`
```

Укажем API использовать этот файл при тонкой настройке:

```
> openai api fine_tunes.create -t "data_prepared.jsonl"
```

Важно помнить, что после завершения тонкой настройки модели ваш запрос должен заканчиваться индикатором `?`, чтобы модель начала генерировать ответ, а не ожидала продолжение обучения. Обязательно включите параметр `stop=["\n"]`, чтобы сгенерированные тексты заканчивались в ожидаемом месте.

Для обучения модели `curie` потребуется примерно 2.48 мин, а для `ada` и `babbage` – меньше. Кроме того, в OpenAI существует очередь заданий на обучение. Ожидание в очереди займет около получаса.

CLI создаст файл с именем `<имя_вашего_файла>_prepared.jsonl`. Мы собираемся использовать его дальше:

```
openai api fine_tunes.create -t "data_prepared.jsonl" -m curie
```

Вы можете использовать `curie` или любую другую базовую модель (`davinci`, `babbage` или `ada`).

Когда тонкая настройка будет завершена, что может занять некоторое время, в интерфейс командной строки будет выведено имя созданной вами новой модели.

Пример:

```
Job complete! Status: succeeded
Try out your fine-tuned model:
openai api completions.create -m curie:ft-learninggpt-2023-02-18-08-38-08 -p
<YOUR_PROMPT>
```

Имейте в виду, что операция может быть прервана по какой-либо внутренней причине API OpenAI, в этом случае вам необходимо возобновить ее с помощью следующей команды:

```
openai api fine_tunes.follow -i <YOUR_FINE_TUNE_JOB_ID>
```

Вы можете получить список своих точно настроенных моделей:

```
openai api fine_tunes.list
```

Теперь можно использовать тонко настроенную модель в коде вашего приложения:

```
export FINE_TUNED_MODEL="<FINE_TUNED_MODEL>"
openai api completions.create -m $FINE_TUNED_MODEL -p <YOUR_PROMPT>
```

Если вы хотите использовать код Python, то фактически ничего не изменится из того, что было раньше, кроме идентификатора модели (FINE_TUNED_MODEL).

```
openai.Completion.create(
   model=FINE_TUNED_MODEL,
   prompt=YOUR_PROMPT
      # Дополнительные параметры
      # temperature,
      # frequency_penalty,
      # presence_penalty
      # ... и т.д.
)
```

Использование модели с помощью cURL выглядит так (одна строка):

```
curl https://api.openai.com/v1/completions
  -H "Authorization: Bearer $OPENAI_API_KEY"
  -H "Content-Type: application/json"
  -d '{"prompt": YOUR_PROMPT, "model": FINE_TUNED_MODEL}'
```

Если вы хотите проанализировать модель, выполните такую команду:

```
openai api fine_tunes.results -i <YOUR_FINE_TUNE_JOB_ID>
```

Вы должны получить вывод в формате CSV наподобие такого:

```
step,elapsed_tokens,elapsed_examples,training_loss,training_sequence_accuracy,
training_token_accuracy
1,25,1,1.6863485659162203,0.0,0.36363636363636365
2,58,2,1.4631860547722317,0.0,0.55
3,91,3,1.7541891464591026,0.0,0.23529411764705882
4,124,4,1.6673923087120057,0.0,0.23529411764705882
5,157,5,1.2660537725454195,0.0,0.55
6,182,6,1.440378345052401,0.0,0.45454545454545453
7,215,7,1.1451897841482424,0.0,0.6
8,240,8,1.3461188264936208,0.0,0.45454545454545453
9,273,9,1.3577824272681027,0.0,0.35294117647058826
10,298,10,1.2882517070074875,0.0,0.45454545454545453
11,331,11,1.0392776031675748,0.0,0.65
12,364,12,1.3298884843569247,0.0,0.35294117647058826
13,397,13,1.0371532278927043,0.0,0.65
```

К имени вашей модели можно добавить суффикс (до 40 символов) с помощью параметра suffix[1]:

```
openai api fine_tunes.create -t data.jsonl -m <engine> --suffix "my_model_name"
```

Наконец, вы можете удалить модель:

```
# Командная строка
openai api models.delete -i <FINE_TUNED_MODEL>
```

[1] **https://platform.openai.com/docs/api-reference/fine-tunes/create#fine-tunes/create-suffix**.

```
# Python
openai.Model.delete(FINE_TUNED_MODEL)

# cURL
curl -X "DELETE" https://api.openai.com/v1/models/<FINE_TUNED_MODEL>
   -H "Authorization: Bearer $OPENAI_API_KEY"
```

Это был небольшой и быстрый тест, чтобы понять, как все работает. Наша тонко настроенная модель не очень эффективна по двум причинам: во-первых, потому что набор данных мал, а во-вторых, потому что мы не применяли специальные приемы для повышения качества обучения.

Позже мы рассмотрим более сложные примеры, но сначала нам нужно разобраться с этими приемами.

10.4. Наборы данных, запросы и ответы: особые приемы

Каждый запрос должен заканчиваться фиксированным разделителем

Чтобы модель точно распознавала конец запроса и начало текста обучающего ответа, важно добавлять фиксированный разделитель. Обычно используется разделитель `\n\n###\n\n`, который не следует использовать в другом месте. Это поможет модели научиться генерировать желаемый результат. В более сложном примере будет показано, как использовать разделитель, чтобы сделать модель лучше.

Каждый обучающий ответ должен начинаться с пробела

При использовании токенизатора OpenAI важно помнить, что каждый обучающий ответ должен начинаться с пробела. Это связано с тем, что токенизатор в основном использует пробелы для токенизации большинства слов. Таким образом, запуск с пробела гарантирует, что токенизатор сможет точно разметить текст. Крайне важно помнить об этом, чтобы гарантировать, что токенизатор работает правильно и дает желаемые результаты.

Каждый обучающий ответ должен заканчиваться фиксированной стоп-последовательностью

Как и запросы, каждый обучающий ответ должен быть помечен стоп-последовательностью, указывающей на окончание ответа. Стоп-последовательность может быть любым токеном, который не используется в ответе, например `\n` или `###`. В рабочем режиме вы должны отформатировать свои запросы так же, как при создании набора обучающих данных, включая тот же разделитель. Кроме того, укажите ту же самую стоп-последовательность, чтобы правильно завершить ответ.

Тонкая настройка работает лучше с более качественными примерами

Чтобы оптимизированная модель работала лучше, чем базовые модели, вам необходимо предоставить как минимум несколько сотен высококаче-

ственных примеров, которые в идеале должны быть подготовлены и оценены экспертами-людьми, как это рекомендует OpenAI в своей официальной документации.

Эксперименты OpenAI свидетельствуют о том, что по мере экспоненциального увеличения количества примеров качество модели склонно к пропорциональному улучшению. Следовательно, увеличение числа обучающих примеров является наиболее надежным и эффективным методом повышения производительности модели.

Предоставляйте более качественные запросы и примеры ответов

Согласно данным OpenAI такой набор обучающих данных:

```
{"prompt": "Product is a laptop. Brand is Dell. Model is Inspiron. RAM is 8GB.
Hard drive is 256GB SSD.->", "completion": " The Dell Inspiron laptop comes
equipped with 8GB of RAM and a 256GB SSD, making it a fast and reliable option
for all your computing needs. END"}
{"prompt": "The book is a thriller. Author is James Patterson. Main character
is a detective. Setting is New York City.->", "completion": " If you're a fan
of thrillers, you'll love this book by James Patterson featuring a detective
in the heart of New York City. Get ready for a heart-pounding read. END"}
{"prompt": "The recipe is for spaghetti carbonara. Ingredients are spaghetti,
bacon,eggs, parmesan cheese. Difficulty level is easy.->", "completion": "
Looking for an easy and delicious dinner option? Try this spaghetti carbonara
recipe that features crispy bacon, creamy eggs, and freshly grated parmesan
cheese. You won't be disappointed! END"}
```

работает лучше, чем такой:

```
{"prompt": "Product : laptop. Brand : Dell. Model : Inspiron. RAM : 8GB. Hard
drive: 256GB SSD.->", "completion": " The Dell Inspiron laptop comes equipped
with 8GB of RAM and a 256GB SSD, making it a fast and reliable option for all
your computing needs. END"}
{"prompt": "The book : thriller. Author : James Patterson. Main character :
detective. Setting : New York City.->", "completion": " If you're a fan of
thrillers, you'll love this book by James Patterson featuring a detective in
the heart of New York City. Get ready for a heart-pounding read. END"}
{"prompt": "The recipe : for spaghetti carbonara. Ingredients : spaghetti,
bacon, eggs, parmesan cheese. Difficulty level : easy.->", "completion": "
Looking for an easy and delicious dinner option? Try this spaghetti carbonara
recipe that features crispy bacon, creamy eggs, and freshly grated parmesan
cheese. You won't be disappointed! END"}
```

Преобразование входных данных в естественный язык, скорее всего, приведет к повышению качества обучения. Этот эффект еще заметнее, когда вы строите генеративную модель.

Проверяйте свои данные на наличие оскорбительного контента

При тонкой настройке существующей модели важно вручную проверять данные на наличие оскорбительного или неточного контента, особенно если вы создаете приложение, которое используется публично.

Если по какой-то причине вы не можете просмотреть весь набор данных, вы все равно можете позаботиться об оценке репрезентативной выборки. Один из подходов состоит в том, чтобы просмотреть как можно больше случайных выборок. Поступая таким образом, вы по-прежнему можете получить представление о данных и выявить нежелательные закономерности или элементы данных.

Проверяйте тип и структуру вашего набора данных

Крайне важно убедиться, что набор данных, используемый для тонкой настройки, по структуре и типу задачи соответствует ожидаемому применению модели и, очевидно, должен содержать достаточное количество релевантных данных для повышения качества модели.

Тонкая настройка модели с неподходящим набором данных может привести к неоптимальным результатам или даже ухудшить их.

Анализируйте свою модель

Мы анализируем модель с помощью команды `openai api fine_tunes.results -i <YOUR_FINE_TUNE_OB_ID>`. Команда выводит результат в формате CSV, как было показано выше. Вот назначение каждого столбца.

1. `step`: в этом столбце указан номер шага обучения или количество итераций процесса обучения.
2. `elapsed_tokens`: показывает количество токенов, обработанных в процессе обучения. Токен – это единица текста, например слово или знак препинания.
3. `elapsed_examples`: это количество примеров (т. е. фрагментов текста), обработанных на данный момент в процессе обучения.
4. `training_loss`: это число показывает значение функции потерь во время обучения. (Функция потерь является мерой того, насколько хорошо работает модель, причем более низкие значения указывают на лучшую производительность.)
5. `training_sequence_accuracy`: точность модели при прогнозировании следующей последовательности токенов. (Последовательность – это группа токенов, образующих осмысленную единицу, например предложение или абзац.)
6. `training_token_accuracy`: это значение говорит нам о точности модели при прогнозировании отдельных токенов.

Как правило, целью процесса обучения является минимизация потерь при обучении при максимальной точности обучающей последовательности и токенов. Такие показатели говорят о том, что модель может создавать отличный текст на естественном языке.

Для анализа результатов также можно использовать некоторые сторонние инструменты, такие как *wandb*[1].

[1] **https://docs.wandb.ai/guides/integrations/other/openai**.

Используйте валидационные данные, если это необходимо

Вы можете отложить часть своих данных для валидации (проверки). Таким образом, вы можете следить за тем, как работает ваша модель, пока вы ее обучаете.

В интерфейсе командной строки OpenAI вы можете создать файл валидации в том же формате, что и файл обучения, и включить его при создании задания тонкой настройки с помощью команды `openai api fine_tunes.create`.

Например, вы можете создать файл валидационных данных с именем `validation_data.jsonl` и включить его в задание тонкой настройки с помощью следующей команды:

```
openai api fine_tunes.create -t train_data.jsonl -v validation_data.jsonl -m
<engine>
```

Во время обучения интерфейс командной строки OpenAI будет периодически вычислять показатели для пакетов валидационных данных и включать их в файл результатов. Результаты валидации состоят из следующих показателей:

- `validation_loss`: потери на валидационном пакете данных;
- `validation_sequence_accuracy`: процент ответов в валидационном пакете, для которых предсказанные токены модели точно совпали с истинными токенами ответа;
- `validation_token_accuracy`: процент токенов в валидационном пакете, которые были правильно предсказаны моделью.

Настройте гиперпараметры

В машинном обучении *гиперпараметр* – это параметр, который управляет *процессом* обучения, а значения других параметров (обычно весовые коэффициенты узлов) определяются посредством обучения.

Разработчики OpenAI уже настроили значения гиперпараметров по умолчанию, которые подходят для различных вариантов использования.

Однако корректировка гиперпараметров во время тонкой настройки может привести к более качественному выводу модели.

Вот некоторые из гиперпараметров:

- `n_epochs`: количество обучающих проходов через весь набор данных (эпох). Использование большего количества эпох может улучшить производительность модели, но также может привести к ее переобучению. Если коротко, переобучение происходит, когда модель слишком фокусируется на конкретных деталях и вариациях в обучающих данных, что приводит к плохой работе с новыми данными. По сути, модель в конечном итоге изучает шум и другие нерелевантные факторы в обучающих данных, как если бы они были реальными признаками;

- `batch_size`: больший размер пакета может ускорить обучение, но также может потребовать больше памяти и снизить качество обобщения модели. Размер пакета OpenAI по умолчанию составляет около 0.2 % от количества примеров в обучающем наборе, ограниченном 256. Размер пакета – это количество примеров, на которых модель обучается в каждой итерации (один проход вперед и назад). Если вы не знакомы с понятием пакета, в следующем примере дана простая иллюстрация. Допустим, вы учитесь решать математические задачи. Размер пакета – это количество задач, которые учитель задает вам на дом, а потом проверяет ответы. Если вам нужно решить много задач, то учителю будет неудобно проверять каждую задачу сразу после решения, именно поэтому он задает вам «пакет» задач. То же самое верно и для обучения модели машинного обучения. Размер пакета – это количество примеров, над которыми модель работает одновременно, прежде чем проверять правильность прогнозов. Если у вас много примеров, может потребоваться разбить их на несколько пакетов, точно так же, как учитель может задать несколько домашних заданий, состоящих из нескольких задач каждое;
- `learning_rate_multiplier`: этот параметр определяет скорость тонкой настройки модели, которая представляет собой исходную скорость предварительного обучения, умноженную на этот коэффициент. API OpenAI устанавливает этот параметр в зависимости от размера пакета, и по умолчанию он равен 0.05, 0.1 или 0.2. Чтобы лучше понять этот параметр, сначала рассмотрим другие определения. В машинном обучении *минимум* – это самая низкая точка функции потерь, которая используется для измерения того, насколько хорошо работает модель. Цель алгоритма обучения – настроить параметры модели так, чтобы она могла достичь этой минимальной точки и добиться наилучшей производительности. Однако, если скорость обучения установлена слишком высокой, модель может проскочить точку минимума и в конечном итоге колебаться вокруг нее или даже отдаляться. Это может привести к тому, что модель будет делать неверные прогнозы. Чтобы модель не проскочила минимум, важно выбрать оптимальную скорость обучения, которая фактически является компромиссом между скоростью обучения и стабильностью модели. Здесь вам поможет только тестирование, поскольку для разных типов наборов данных или моделей нужны разные скорости обучения;
- `compute_classification_metrics`: этот параметр предназначен для точной настройки задач классификации. Если установлено значение `True`, модель вычисляет специфические для классификации показатели (такие как точность[1] и оценка F-1[2]) на валидационном наборе в конце каждой эпохи. Эти показатели могут помочь вам оценить произ-

[1] **https://developers.google.com/machine-learning/crash-course/classification/accuracy**.

[2] **https://en.wikipedia.org/wiki/F-score**.

водительность модели и внести коррективы по мере необходимости.

Стоит отметить, что выбор гиперпараметров может существенно повлиять на время, необходимое для обучения и тестирования модели. Это связано с тем, что от некоторых гиперпараметров зависят сложность модели и ее способность обобщать новые данные.

Используйте для классификации модель Ada

Для решения задач классификации хорошо подходит модель Ада. После тонкой настройки она работает лишь немного хуже, чем Davinci, в то же время значительно быстрее и доступнее по цене.

Используйте классы, состоящие из одного токена

Допустим, вы классифицируете предложения по некоторым категориям. Рассмотрим такой пример:

```
{prompt:"The Los Angeles Lakers won the NBA championship last year.",
completion: "sports and entertainment"}
{prompt:"Apple is set to release a new iPhone model in the coming months.",
completion: "technology and science"}
{prompt:"The United States Congress passed a $1.9 trillion COVID-19 relief bill.",
completion: "politics and government"}
{prompt:"The Tokyo Olympics were postponed to 2021 due to the COVID-19 pandemic.",
completion: "sports and entertainment"}
{prompt:"Tesla's market capitalization surpassed that of Toyota in 2020.",
completion: "technology and science"}
{prompt:"Joe Biden was inaugurated as the 46th President of the United States
on January 20, 2021.", completion: "politics and government"}
{prompt:"Novak Djokovic won the Australian Open tennis tournament for the
ninth time in 2021.", completion: "sports and entertainment"}
{prompt:"Facebook was fined $5 billion by the US Federal Trade Commission for
privacy violations.", completion: "technology and science"}
{prompt:"The UK officially left the European Union on January 31, 2020.",
completion: "politics and government"}
{prompt:"The Tampa Bay Buccaneers won the Super Bowl in 2021, led by quarterback
Tom Brady.", completion: "sports and entertainment"}
```

Эти данные разбиты на три класса:

- `sports and entertainment` (спорт и развлечения),
- `technology and science` (технологии и наука),
- `politics and government` (политика и правительство).

Вместо того чтобы использовать их в качестве длинных имен классов, разумнее использовать один токен, например:

- 1 (для спорта и развлечений),
- 2 (для технологий и науки),
- 3 (для политики и правительства).

```
{prompt:"The Los Angeles Lakers won the NBA championship last year.",
completion: "1"}
{prompt:"Apple is set to release a new iPhone model in the coming months.",
completion: "2"}
{prompt:"The United States Congress passed a $1.9 trillion COVID-19 relief bill.",
completion: "3"}
{prompt:"The Tokyo Olympics were postponed to 2021 due to the COVID-19 pandemic.",
completion: "1"}
{prompt:"Tesla's market capitalization surpassed that of Toyota in 2020.",
completion: "2"}
{prompt:"Joe Biden was inaugurated as the 46th President of the United States
on January 20, 2021.", completion: "3"}
{prompt:"Novak Djokovic won the Australian Open tennis tournament for the
ninth time in 2021.", completion: "1"}
{prompt:"Facebook was fined $5 billion by the US Federal Trade Commission for
privacy violations.", completion: "2"}
{prompt:"The UK officially left the European Union on January 31, 2020.",
completion: "3"}
{prompt:"The Tampa Bay Buccaneers won the Super Bowl in 2021, led by
quarterback Tom Brady.", completion: "1"}
```

В этом случае не только обучающие данные будут меньше, но и вывод будет состоять всего из одного токена:

```
openai.Completion.create(
    engine=<engine>,
    max_tokens=1,
)
```

Другие советы по использованию классификации

- Убедитесь, что общая длина запроса и ответа не превышает 2048 токенов, включая разделитель.
- Постарайтесь предоставить не менее 100 примеров для каждого класса.
- Разделитель не должен использоваться в тексте запроса. Вы должны удалить его из запроса, если это так. Пример: если вы используете разделитель !#! вы должны предварительно обработать текст запроса, чтобы удалить из него все сочетания !#!

Глава 11

Продвинутая тонкая настройка: классификация лекарств

11.1. Набор данных, используемый в примере

В этом примере мы будем использовать общедоступный набор данных, содержащий названия лекарств и соответствующие заболевания, болезни или состояния, для лечения которых они используются.

Мы создадим модель и «научим» ее предсказывать результат на основе пользовательского ввода. Ввод пользователя – это название лекарства, а вывод – название болезни.

Набор данных доступен на *Kaggle.com*, вам нужно будет скачать его по следующему адресу: **https://www.kaggle.com/datasets/saratchendra/medicine-recommendation/download?datasetVersionNumber=1** или перейти по адресу: **https://www.kaggle.com/datasets/saratchendra/medicine-recommendation**, а затем загрузить файл с именем `Medicine_description.xlsx`.

Таблица Excel содержит три листа; мы будем использовать первый с именем *Sheet1*, в котором есть три столбца:

- Drug_Name (имя_препарата),
- Reason (причина),
- Description (описание).

Мы будем использовать первый и второй столбцы, так как они содержат название препарата и причину, по которой он рекомендуется.

Например:

```
A CN Gel(Topical) 20gmA CN Soap 75gm ==> Acne
PPG Trio 1mg Tablet 10'SPPG Trio 2mg Tablet 10'S ==> Diabetes
Iveomezoled 200mg Injection 500ml ==> Fungal
```

11.2. Подготовка данных и запуск тонкой настройки

Данные сохранены в файле Excel с форматом XLSX. Мы преобразуем его в формат JSONL (JSON с разделением переносом строки). Затем воспользу-

емся файлом JSONL для тонкой настройки модели, как делали в предыдущей главе.

Внутри файла данные будут представлены в таком формате:

```
{"prompt":"Drug: <DRUG NAME>\nMalady:","completion":" <MALADY NAME>"}
```

Как видите, в качестве разделителя мы будем использовать строку `\nMalady:`[1].

Ответ также будет начинаться с пробела. Напомню, что это необходимо из-за токенизации (большинство слов токенизированы предшествующим пробелом).

Кроме того, вы уже знаете, что каждый пример ответа должен заканчиваться одной и той же стоп-последовательностью, чтобы информировать модель об окончании вывода, например `\n`, `###`, `END` или любой другой токен, который не встречается в тексте ответа.

Однако в нашем случае в этом нет необходимости, так как мы собираемся использовать для классификации один токен. По сути, мы присвоим каждой болезни уникальный идентификатор. Например:

```
Acne: 1
Allergies: 2
Alzheimer: 3
... и т.д.
```

Следовательно, модель во всех случаях будет возвращать только один токен во время вывода. По этой причине стоп-последовательность не нужна.

Для начала используем Pandas для преобразования данных в нужный формат:

```
import pandas as pd

# Читаем первые n строк
n = 2000

df = pd.read_excel('Medicine_description.xlsx', sheet_name='Sheet1', header=0,
nrows=n)

# Получаем уникальные значения из столбца Reason (причина)
reasons = df["Reason"].unique()

# Присваиваем номер каждой причине
reasons_dict = {reason: i for i, reason in enumerate(reasons)}

# Добавляем символ новой строки и ### в конец каждого описания
df["Drug_Name"] = "Drug: " + df["Drug_Name"] + "\n" + "Malady:"

# Объединяем столбцы Reason и Description
df["Reason"] = " " + df["Reason"].apply(lambda x: "" + str(reasons_dict[x]))
```

[1] Malady (англ.) – заболевание.

```
# Удаляем столбец Reason
df.drop(["Description"], axis=1, inplace=True)

# Переименовываем столбцы
df.rename(columns={"Drug_Name": "prompt", "Reason": "completion"},
inplace=True)

# Конвертируем кадр данных в формат jsonl
jsonl = df.to_json(orient="records", indent=0, lines=True)

# Записываем jsonl в файл
with open("drug_malady_data.jsonl", "w") as f:
f.write(jsonl)
```

В приведенном выше коде мы читаем из таблицы Excel первые 2000 строк. Это означает, что мы будем использовать набор данных из 2000 наименований лекарств для точной настройки модели. Вы можете использовать больше записей.

Код начинается с чтения первых `n` строк данных из файла Excel `Medicine_description.xlsx` и сохранения их в кадре данных с именем `df`.

Затем код получает уникальные значения в столбце `Reason` кадра данных, сохраняет их в массиве `reasons`, присваивает числовой индекс каждому уникальному значению в массиве и сохраняет его в словаре `reasons_dict`.

Далее скрипт добавляет символ новой строки и `"Malady:"` (заболевание) в конец каждого названия лекарства в столбце `Drug_Name` фрейма данных. Он объединяет пробел и соответствующий числовой индекс из `reasons_dict` и помещает в конец каждого значения `Reason` в кадре данных.

Это делается для получения нужного формата данных:

```
Drug: <DRUG NAME>\nMalady:
```

В этом примере нам не нужен столбец `Description`, поэтому скрипт удаляет его из кадра данных. Далее скрипт переименовывает столбец `Drug_Name` в `prompt`, а столбец `Reason` в `completion`.

Кадр данных преобразуется в формат JSONL и сохраняется в переменной с именем `jsonl`, которая записывается в файл `drug_malady_data.jsonl`.

Вот как выглядит содержимое файла `drug_malady_data.jsonl`:

```
1 [..]
2 {"prompt":"Drug: Acleen 1% Lotion 25ml\nMalady:","completion":" 0"}
3 [..]
4 {"prompt":"Drug: Capnea Injection 1ml\nMalady:","completion":" 1"}
5 [..]
6 {"prompt":"Drug: Mondeslor Tablet 10'S\nMalady:","completion":" 2"}
7 [..]
```

Теперь перейдем к следующему шагу, выполнив в окне командной строки команду:

```
openai tools fine_tunes.prepare_data -f drug_malady_data.jsonl
```

Эта команда поможет нам подготовить данные и понять, как они будут приниматься моделью:

```
Analyzing...

- Your file contains 2000 prompt-completion pairs
- Based on your data it seems like you're trying to fine-tune a model for
classification
- For classification, we recommend you try one of the faster and cheaper models,
such as `ada`
- For classification, you can estimate the expected model performance by keeping
a held out dataset, which is not used for training
- All prompts end with suffix `\nMalady:`
- All prompts start with prefix `Drug:
```

Вы можете разделить данные на два набора: один будет использоваться для обучения, а другой – для проверки:

```
- [Recommended] Would you like to split into training and validation set?
[Y/n]:
```

В окне командной строки также будет показана команда для обучения модели:

```
Now use that file when fine-tuning:
> openai api fine_tunes.create -t "drug_malady_data_prepared_train.jsonl" -v
"drug_malady_data_prepared_valid.jsonl" --compute_classification_metrics --
classification_n_classes 3
```

Она также может определять количество классов, используемых в наборе данных.

Наконец, выполните показанную вам команду, чтобы начать обучение. Вы также можете указать модель. Мы будем использовать `ada`, она недорогая и отлично подходит для нашей задачи.

Вы также можете добавить суффикс к имени настроенной модели. Мы будем использовать `drug_malady_data`.

```
# export your OpenAI key
export OPENAI_API_KEY="xxxxxxxxxxxx"
openai api fine_tunes.create
 -t "drug_malady_data_prepared_train.jsonl"
 -v "drug_malady_data_prepared_valid.jsonl"
 --compute_classification_metrics \
 --classification_n_classes 3
 -m ada
 --suffix "drug_malady_data"
```

Если клиент отключается до завершения задания, вам может быть предложено повторно подключиться и проверить выполнение с помощью следующей команды:

```
openai api fine_tunes.follow -i <JOB ID>
```

Ниже показан пример вывода, когда задание завершено:

```
Created fine-tune: <JOB ID>
Fine-tune costs $0.03
Fine-tune enqueued
Fine-tune is in the queue. Queue number: 31
Fine-tune is in the queue. Queue number: 30
Fine-tune is in the queue. Queue number: 29
Fine-tune is in the queue. Queue number: 28
[...]
[...]
[...]
Fine-tune is in the queue. Queue number: 2
Fine-tune is in the queue. Queue number: 1
Fine-tune is in the queue. Queue number: 0
Fine-tune started
Completed epoch 1/4
Completed epoch 2/4
Completed epoch 3/4
Completed epoch 4/4
Uploaded model: <MODEL ID>
Uploaded result file: <FILE ID>
Fine-tune succeeded

Job complete! Status: succeeded
Try out your fine-tuned model:

openai api completions.create -m <MODEL ID> -p <YOUR_PROMPT>
```

Выходные данные показывают ход и состояние задания тонкой настройки. Они подтверждают, что новое задание тонкой настройки было создано с указанным идентификатором. В этих данных также содержится дополнительная информация:

- стоимость тонкой настройки;
- количество завершенных эпох;
- идентификатор файла, содержащего результаты тонкой настройки.

11.3. Тестирование настроенной модели

Когда модель будет готова, вы можете протестировать ее, используя следующий код:

```
import os
import openai

def init_api():
   with open(".env") as env:
```

```
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

# Указываем идентификатор модели. Вставьте здесь ID своей модели
model = "ada:ft-learninggpt:drug-malady-data-2023-02-21-20-36-07"

# Будем использовать лекарства из каждого класса
drugs = [
    "A CN Gel(Topical) 20gmA CN Soap 75gm", # Класс 0
    "Addnok Tablet 20'S", # Класс 1
    "ABICET M Tablet 10's", # Класс 2
]

# Возвращает класс лекарства для каждого класса
for drug_name in drugs:
    prompt = "Drug: {}\nMalady:".format(drug_name)

    response = openai.Completion.create(
        model=model,
        prompt= prompt,
        temperature=1,
        max_tokens=1,
    )

    # Выводим на п сгенерированный текст
    drug_class = response.choices[0].text
    # Вывод должен содержать 0, 1 и 2
    print(drug_class)
```

Мы тестируем модель с тремя названиями лекарств, каждое из которых относится к разным классам:

- A CN Gel(Topical) 20gmA CN Soap 75gm, класс 0 (Acne),
- Addnok Tablet 20'S, класс 1 (Adhd),
- ABICET M Tablet 10's, класс 2 (Allergies).

Результатом выполнения кода должен быть такой вывод в консоль:

```
0
1
2
```

Если мы попробуем следующий код

```
drugs = [
   "What is 'A CN Gel(Topical) 20gmA CN Soap 75gm' used for?", # Класс 0
   "What is 'Addnok Tablet 20'S' used for?", # Класс 1
   "What is 'ABICET M Tablet 10's' used for?", # Класс 2
]
```

результат будет таким же.

Мы можем добавить в код переводную таблицу, чтобы возвращать название болезни вместо ее класса:

```
# Используем по одному лекарству из каждого класса
drugs = [
   "What is 'A CN Gel(Topical) 20gmA CN Soap 75gm' used for?", # Класс 0
   "What is 'Addnok Tablet 20'S' used for?", # Класс 1
   "What is 'ABICET M Tablet 10's' used for?", # Класс 2
]

class_map = {
   0: "Acne",
   1: "Adhd",
   2: "Allergies",
   # ...
}

# Возвращаем класс для каждого лекарства
for drug_name in drugs:
    prompt = "Drug: {}\nMalady:".format(drug_name)

    response = openai.Completion.create(
       model=model,
       prompt= prompt,
       temperature=1,
       max_tokens=1,
    )

    response = response.choices[0].text
    try:
       print(drug_name + " применяется при " + class_map[int(response)])
    except:
       print("Я не знаю, когда применяется " + drug_name)
    print()
```

Глава 12

Продвинутая тонкая настройка: создание ассистирующего чат-бота

12.1. Интерактивная классификация

В этой главе мы будем использовать модель классификации из главы 11 для создания ассистирующего чат-бота.

Одного того факта, что классификация возвращает название болезни для данного лекарства, недостаточно для создания чат-бота.

Когда пользователь спрашивает чат-бот о лекарстве, он должен ответить, назвав болезнь, дав ее определение и по возможности предоставив дополнительную информацию.

Цель состоит в том, чтобы сделать классификацию лекарств более удобной для человека; поэтому мы и собираемся создать чат-бот, который будет выступать в роли ассистента.

12.2. Как это будет работать?

Ассистент должен инициировать обычный диалог с пользователем. Пока пользователь продолжает разговор, ассистент продолжает отвечать, пока пользователь не спросит о названии препарата. В этом случае ассистент должен ответить названием соответствующей болезни.

Мы определим три функции:

- `regular_discussion()`: эта функция возвращает ответ API с использованием Davinci каждый раз, когда пользователь говорит на обычную тему. Если пользователь спрашивает о лекарстве, функция вызывает `get_malady_name()`;
- `get_malady_name()`: возвращает название болезни, которое соответствует названию лекарства из точной модели. Кроме того, функция вызовет `get_malady_description()`, чтобы получить описание болезни;

- `get_malady_description()`: получает описание болезни из API с помощью Davinci и возвращает его.

Конечный потребитель, спросив о названии лекарства, получит название болезни (из доработанной модели) и ее описание (из Davinci).

Напишем первую функцию:

```
def regular_discussion(prompt):
    """
    аргументы: prompt - строка
    Возвращает ответ API с использованием Davinci.
    Если пользователь спрашивает о лекарстве, вызывает get_malady_name().
    """
    prompt = """
    Далее следует диалог с ИИ-ассистентом. Ассистент доброжелателен, креативен,
    умен, очень дружелюбен и осторожен в вопросах здоровья Человека
    ИИ-ассистент не является врачом и не может диагностировать заболевания или
    назначать лечение Человеку
    ИИ-ассистент не является фармацевтом и не может рекомендовать Человеку
    лекарственные препараты
    ИИ-ассистент не дает Человеку советы относительно лечения
    ИИ-ассистент не ставит Человеку медицинские диагнозы
    ИИ-ассистент не назначает Человеку медицинские процедуры
    ИИ-ассистент не выдает Человеку рецепты на лекарства
    Если Человек вводит название лекарства, ИИ-ассистент подставляет ответ
    вместо "######".

    Человек: Привет
    ИИ: Привет, человек. Как дела? Буду рад тебе помочь. Назови лекарство, и я
    расскажу, для чего оно применяется.
    Человек: Vitibex
    ИИ: ######
    Человек: У меня все хорошо. Как дела у тебя?
    ИИ: У меня тоже все хорошо. Спасибо, что спросил. Буду рад тебе помочь.
    Назови лекарство, и я расскажу, для чего оно применяется.
    Человек: Что такое хаос-инжиниринг?
    ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только
    на вопросы о лекарствах. Назови лекарство, и я расскажу, для чего оно
    применяется.
    Человек: Где находится Карфаген?
    ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только
    на вопросы о лекарствах. Назови лекарство, и я расскажу, для чего оно
    применяется.
    Человек: Что такое Maxcet 5mg Tablet 10'S?
    ИИ: ######
    Человек: Что такое Axepta?
    ИИ: ######
    Человек: {}
```

```
    ИИ:"""".format(prompt)

    # Получение ответа от API
    response = openai.Completion.create(
        model="text-davinci-003",
        prompt=prompt,
        max_tokens=100,
        stop=["\n", " Human:", " AI:"],
    )
    if response.choices[0].text.strip() == "######":
        get_malady_name(prompt)
    else:
        final_response = response.choices[0].text.strip() + "\n"
        print("AI: {}".format(final_response))
```

Чтобы сократить бесполезные разговоры и избежать предоставления медицинских советов с помощью одной и той же функции, мы использовали длинную подсказку, в которой ассистент описывается как полезный, креативный, умный, дружелюбный и осторожный в вопросах здоровья человека. Однако подчеркивается, что ИИ-ассистент не является врачом или фармацевтом и не занимается диагностикой и лечением заболеваний, а также не дает медицинских советов, не ставит диагнозы, не назначает лечение и не выписывает рецепты. Если человек напишет название лекарства, ассистент ответит ######, а модель должна подставить название соответствующего заболевания. Когда это происходит, мы вызываем вторую функцию.

Эта функция выглядит так:

```
def get_malady_name(drug_name):
    """
    аргументы: drug_name - строка
    Возвращает название заболевания, соответствующего названию препарата,
    из настроенной модели.
    Эта функция вызывает get_malady_description(), чтобы получить описание болезни
    """

    # Указываем идентификатор модели. Вставьте здесь ID своей модели
    model = "ada:ft-learninggpt:drug-malady-data-2023-02-21-20-36-07"
    class_map = {
        0: "Acne",
        1: "Adhd",
        2: "Allergies",
        # ...
    }

    # Возвращаем класс для каждого лекарства
    prompt = "Drug: {}\nMalady:".format(drug_name)

    response = openai.Completion.create(
        model=model,
```

```
            prompt= prompt,
            temperature=1,
            max_tokens=1,
            )

        response = response.choices[0].text.strip()
        try:
            malady = class_map[int(response)]
            print("ИИ: Это лекарство применяется для лечения {}.".format(malady))
            print(get_malady_description(malady))
        except:
            print("ИИ: Я не знаю, для чего применяется '" + drug_name + "'")
```

Как вы знаете из предыдущей главы, этот код возвращает название болезни, соответствующее названию лекарства из отлаженной модели. Функция вызовет `get_malady_description()`, чтобы получить описание болезни.

А вот последняя функция:

```
def get_malady_description(malady):
    """
    аргументы: malady - строка
    Получает описание болезни из API с помощью Davinci.
    """
    prompt = """
    Далее следует диалог с ИИ-ассистентом. Ассистент полезен, креативен,
    умен и очень дружелюбен.
    Ассистент не выдает медицинские заключения. Он только описывает
    недомогание, болезнь или состояние.
    Если ассистент не знает ответ, он просит перефразировать вопрос.

    Q: Что такое {}?
    A:""".format(malady)

    # Получение ответа из API.
    response = openai.Completion.create(
        model="text-davinci-003",
        prompt=prompt,
        max_tokens=100,
        stop=["\n", " Q:", " A:"],
    )
    return response.choices[0].text.strip()
```

Это обычный диалог вопрос–ответ, который дает определение болезни. Давайте соберем все фрагменты вместе:

```
import os
import openai

def init_api():
    with open(".env") as env:
        for line in env:
```

```
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

def regular_discussion(prompt):
    """
    аргументы: prompt - строка
    Возвращает ответ API с использованием Davinci.
    Если пользователь спрашивает о лекарстве, вызывает get_malady_name().
    """
    prompt = """
    Далее следует диалог с ИИ-ассистентом. Ассистент доброжелателен, креативен,
    умен, очень дружелюбен и осторожен в вопросах здоровья Человека
    ИИ-ассистент не является врачом и не может диагностировать заболевания или
    назначать лечение Человеку
    ИИ-ассистент не является фармацевтом и не может рекомендовать Человеку
    лекарственные препараты
    ИИ-ассистент не дает Человеку советы относительно лечения
    ИИ-ассистент не ставит Человеку медицинские диагнозы
    ИИ-ассистент не назначает Человеку медицинские процедуры
    ИИ-ассистент не выдает Человеку рецепты на лекарства
    Если Человек вводит название лекарства, ИИ-ассистент подставляет ответ
    вместо "######".

    Человек: Привет
    ИИ: Привет, человек. Как дела? Буду рад тебе помочь. Назови лекарство,
    и я расскажу, для чего оно применяется.
    Человек: Vitibex
    ИИ: ######
    Человек: У меня все хорошо. Как дела у тебя?
    ИИ: У меня тоже все хорошо. Спасибо, что спросил. Буду рад тебе помочь.
    Назови лекарство, и я расскажу, для чего оно применяется.
    Человек: Что такое хаос-инжиниринг?
    ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только
    на вопросы о лекарствах. Назови лекарство, и я расскажу, для чего оно
    применяется.
    Человек: Где находится Карфаген?
    ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только
    на вопросы о лекарствах. Назови лекарство, и я расскажу, для чего оно
    применяется.
    Человек: Что такое Maxcet 5mg Tablet 10'S?
    ИИ: ######
    Человек: Что такое Ахерта?
    ИИ: ######
    Человек: {}
    ИИ:""".format(prompt)

    # Получение ответа от API
```

```
        response = openai.Completion.create(
            model="text-davinci-003",
            prompt=prompt,
            max_tokens=100,
            stop=["\n", " Human:", " AI:"],
        )
        if response.choices[0].text.strip() == "######":
            get_malady_name(prompt)
        else:
            final_response = response.choices[0].text.strip() + "\n"
            print("AI: {}".format(final_response))

    def get_malady_name(drug_name):
        """
        аргументы: drug_name - строка
        Возвращает название заболевания, соответствующего названию препарата, из
        настроенной модели.
        Эта функция вызывает get_malady_description(), чтобы получить описание болезни
        """

        # Указываем идентификатор модели. Вставьте здесь ID своей модели
        model = "ada:ft-learninggpt:drug-malady-data-2023-02-21-20-36-07"
        class_map = {
            0: "Acne",
            1: "Adhd",
            2: "Allergies",
            # ...
        }

        # Возвращаем класс для каждого лекарства
        prompt = "Лекарство: {}\nMalady:".format(drug_name)

        response = openai.Completion.create(
            model=model,
            prompt= prompt,
            temperature=1,
            max_tokens=1,
        )

        response = response.choices[0].text.strip()
        try:
            malady = class_map[int(response)]
            print("ИИ: Это лекарство применяется для лечения {}.".format(malady))
            print(get_malady_description(malady))
        except:
            print("ИИ: Я не знаю, для чего применяется '" + drug_name + "'")

    def get_malady_description(malady):
        """
        аргументы: malady - строка
        Получает описание болезни из API с помощью Davinci.
        """
```

```
prompt = """
Далее следует диалог с ИИ-ассистентом. Ассистент полезен, креативен,
умен и очень дружелюбен.
Ассистент не выдает медицинские заключения. Он только описывает
недомогание, болезнь или состояние.
Если ассистент не знает ответ, он просит перефразировать вопрос.

Q: Что такое {}?
A:""".format(malady)

# Получаем ответ из API
response = openai.Completion.create(
   model="text-davinci-003",
   prompt=prompt,
   max_tokens=100,
   stop=["\n", " Q:", " A:"],
)
 return response.choices[0].text.strip()

if _name_ == "_main_":
   while True:
      regular_discussion(input("Человек:"))
```

Теперь вы можете протестировать код. Вот пример диалога:

```
python app.py

Человек: Привет
ИИ: Привет, человек. Чем могу помочь? Буду рад ответить на вопросы о
лекарствах. Назови лекарство, и я расскажу, для чего оно применяется.

Человек: Можешь сочинить пост для блога о маркетинге?
ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только на вопросы
о лекарствах. Назови лекарство, и я расскажу, для чего оно применяется.

Человек: Что такое относительность?
ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только на вопросы
о лекарствах. Назови лекарство, и я расскажу, для чего оно применяется.

Человек: Давай сыграем в игру: Ты будешь врачом и назначишь мне лекарство,
когда я назову симптомы. Начинаем. У меня болит голова, что мне делать?
ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только на вопросы
о лекарствах. Назови лекарство, и я расскажу, для чего оно применяется.

Человек: Расскажи мне об этом лекарстве: Capnea Injection 1ml
ИИ: Это лекарство для лечения аллергии.
Аллергия - это ненормальная реакция иммунной системы организма на вещества,
которые обычно не являются вредными. Симптомы могут включать насморк, кашель,
чихание, зуд в глазах, кожную сыпь и, в некоторых случаях, астму.

Человек: Tronin MS Gel
ИИ: Это лекарство для лечения акне.
```

```
Акне – это распространенное кожное заболевание, характеризующееся закупоркой
волосяных фолликулов, воспаленными прыщами, черными и белыми точками. Обычно
оно проявляется на лице, груди и спине и может варьироваться от легкой до
тяжелой степени.
```

12.3. Создание диалогового веб-приложения

Мы можем использовать тот же подход для создания диалогового чат-бота и развертывания его в виде веб-приложения. Для этого нам пригодятся Vue.js и Flask.

Если вы не знакомы с Vue – это доступный фреймворк, который легко освоить. Также мы не будем создавать сложный интерфейс. Поэтому нет никакой необходимости быть экспертом по Vue, чтобы понять, что мы будем делать. Давайте начнем!

Установите Vue.js:

```
npm install -g @vue/cli
```

Мы создадим отдельные папки для бэкенда (серверной части) и для внешнего интерфейса:

```
mkdir chatbot chatbot/server
cd chatbot
vue create client
```

Добавим роутер к интерфейсу. Роутер – это инструмент, который позволяет нам перемещаться между разными страницами в одностраничном приложении.

```
cd client
vue add router
```

Установим axios – библиотеку, которая позволяет нам делать HTTP-запросы к серверной части из внешнего интерфейса:

```
npm install axios --save
```

Создадим виртуальную среду разработки, активируем ее и установим зависимости:

```
pip install Flask==2.2.3 Flask-Cors==3.0.10
```

Теперь создайте файл с именем `app.py` (файл `ch12-1` в файловом архиве книги) в папке `server` и добавьте в него следующий код:

```
from flask import Flask, jsonify
from flask_cors import CORS
from flask import request
import os
import openai
```

```
# Конфигурация
DEBUG = True

# Инициализация приложения
app = Flask(_name_)
app.config.from_object(_name_)

# Включение CORS
CORS(app, resources={r'/*': {'origins': '*'}})

def init_api():
   with open(".env") as env:
     for line in env:
         key, value = line.strip().split("=")
         os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

def regular_discussion(prompt):
   """
   аргументы: prompt - строка
   Возвращает ответ API с использованием Davinci.
   Если пользователь спрашивает о лекарстве, вызывает get_malady_name().
   """
   prompt = """
   Далее следует диалог с ИИ-ассистентом. Ассистент доброжелателен, креативен,
   умен, очень дружелюбен и осторожен в вопросах здоровья Человека
   ИИ-ассистент не является врачом и не может диагностировать заболевания или
   назначать лечение Человеку
   ИИ-ассистент не является фармацевтом и не может рекомендовать Человеку
   лекарственные препараты
   ИИ-ассистент не дает Человеку советы относительно лечения
   ИИ-ассистент не ставит Человеку медицинские диагнозы
   ИИ-ассистент не назначает Человеку медицинские процедуры
   ИИ-ассистент не выдает Человеку рецепты на лекарства
   Если Человек вводит название лекарства, ИИ-ассистент подставляет ответ
   вместо "######".

   Человек: Привет
   ИИ: Привет, человек. Как дела? Буду рад тебе помочь. Назови лекарство,
   и я расскажу, для чего оно применяется.
   Человек: Vitibex
   ИИ: ######
   Человек: У меня все хорошо. Как дела у тебя?
   ИИ: У меня тоже все хорошо. Спасибо, что спросил. Буду рад тебе помочь.
   Назови лекарство, и я расскажу, для чего оно применяется.
   Человек: Что такое хаос-инжиниринг?
   ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только
```

```
        на вопросы о лекарствах. Назови лекарство, и я расскажу, для чего оно
        применяется.
        Человек: Где находится Карфаген?
        ИИ: Прости, не могу тебе помочь. Я запрограммирован отвечать только
        на вопросы о лекарствах. Назови лекарство, и я расскажу, для чего оно
        применяется.
        Человек: Что такое Maxcet 5mg Tablet 10'S?
        ИИ: ######
        Человек: Что такое Ахерта?
        ИИ: ######
        Человек: {}
        ИИ:""".format(prompt)

        # Получение ответа от API
        response = openai.Completion.create(
            model="text-davinci-003",
            prompt=prompt,
            max_tokens=100,
            stop=["\n", " Человек:", " ИИ:"],
        )
        if response.choices[0].text.strip() == "######":
            return get_malady_name(prompt)

        else:
            final_response = response.choices[0].text.strip() + "\n"
            return("{}".format(final_response))

    def get_malady_name(drug_name):
        """
        аргументы: drug_name - строка
        Возвращает название заболевания, соответствующего названию препарата,
        из настроенной модели.
        Эта функция вызывает get_malady_description(), чтобы получить описание
        болезни
        """

        # Указываем идентификатор модели. Вставьте здесь ID своей модели
        model = "ada:ft-learninggpt:drug-malady-data-2023-02-21-20-36-07"
        class_map = {
            0: "Acne",
            1: "Adhd",
            2: "Allergies",
            # ...
        }

        # Возвращаем класс для каждого лекарства
        prompt = "Лекарство: {}\nMalady:".format(drug_name)

        response = openai.Completion.create(
            model=model,
            prompt= prompt,
```

```
        temperature=1,
        max_tokens=1,
    )

    response = response.choices[0].text.strip()
    try:
        malady = class_map[int(response)]
        print("==")
        print("Это лекарство применяется для лечения {}.".format(malady) +
        get_malady_description(malady))
        return " Это лекарство применяется для лечения {}.".format(malady) + " "
        get_malady_description(malady)
    except:
        return " Я не знаю, для чего применяется '" + drug_name + "'"

def get_malady_description(malady):
    """
    аргументы: malady - строка
    Получает описание болезни из API с помощью Davinci.
    """
    prompt = """
    Далее следует диалог с ИИ-ассистентом. Ассистент полезен, креативен,
    умен и очень дружелюбен.
    Ассистент не выдает медицинские заключения. Он только описывает
    недомогание, болезнь или состояние.
    Если ассистент не знает ответ, он просит перефразировать вопрос.

    Q: Что такое {}?
    A:""".format(malady)

    # Получаем ответ из API
    response = openai.Completion.create(
        model="text-davinci-003",
        prompt=prompt,
        max_tokens=100,
        stop=["\n", " Q:", " A:"],
    )
    return response.choices[0].text.strip() + "\\n"

    @app.route('/', methods=['GET'])
    def repply():
        m = request.args.get('m')
        chatbot = regular_discussion(m)
        print("chatbot: ", chatbot)
    return jsonify({'m': chatbot})

    if _name_ == '_main_':
        app.run()
```

Создайте файл с именем `.env` в папке сервера и добавьте свой ключ API и идентификатор организации:

```
API_KEY=sk-xxxx
ORG_ID=org-xxx #не обязательно
```

В папке внешнего интерфейса откройте файл `chatbot/client/src/router/index.js` и добавьте следующий код:

```
import { createRouter, createWebHistory } from 'vue-router'
import HomeView from '../views/HomeView.vue'

const routes = [
  {
   path: '/',
   name: 'home',
   component: HomeView
  },
  {
   path: '/about',
   name: 'about',
   // разделение кода на уровне маршрута
   // создает отдельный фрагмент (about.[hash].js) для этого маршрута,
   // который загружается отложенно при посещении маршрута
   component: () => import(/* webpackChunkName: "about" */ '../views/
   AboutView.vue')
  }
]

const router = createRouter({
   history: createWebHistory(process.env.BASE_URL),
   routes
})

export default router
```

Приведенный выше код создает два маршрута: / и `/about`. Маршрут / будет использоваться для отображения чат-бота (`HomeView.vue`), а маршрут `/about` – для отображения страницы с информацией (`AboutView.vue`).

Внутри папки `client/src/views` создайте файл с именем `HomeView.vue` и добавьте следующий код:

```
<template>
  <div>
    <h2>DrugBot</h2>
    <div v-if="messages.length">
      <div v-for="message in messages" :key="message.id">
        <strong>{{ message.author }}:</strong> {{ message.text }}
      </div>
    </div>
    <form @submit.prevent="sendMessage">
      <input type="text" v-model="newMessage" placeholder="Введите ваше
сообщение">
      <button type="submit">Send</button>
```

```
    </form>
  </div>
</template>

<script>
import axios from 'axios';

export default {
  data() {
    return
      messages: [
        { id: 1, author: "AI", text: "Привет, чем могу помочь?" },
    ],
    newMessage: "",
  };
},
methods: {
    sendMessage() {
      if (this.newMessage.trim() === "") {
        return
      }
      this.messages.push({
        id: this.messages.length + 1,
        author: "Человек",
        text: this.newMessage.trim(),
      });

      const messageText = this.newMessage.trim();

        axios.get(`http://127.0.0.1:5000/?m=${encodeURI(messageText)}`)
        .then(response => {
          const message = {
            id: this.messages.length + 1,
            author: "ИИ",
            text: response.data.m
          };
          this.messages.push(message);
        })
        .catch(error => {
          console.error(error);
          this.messages.push({
            id: this.messages.length + 1,
            author: "ИИ",
            text: "Простите, я не понимаю.",
          });
        });

        this.newMessage = "";
      },
    },
  };
</script>
```

Я предполагаю, что ваше приложение Flask использует порт 5000, в противном случае измените номер порта в приведенном выше коде на нужный вам.

Приведенный выше код создает простой интерфейс чат-бота. Метод `sendMessage()` отправляет сообщение пользователя на сервер и получает ответ от сервера. Затем ответ отображается в интерфейсе чат-бота.

Файл `HomeView.vue` является представлением по умолчанию для маршрута `/`. Чтобы создать файл `AboutView.vue`, выполните следующую команду:

```
touch client/src/views/AboutView.vue
```

Откройте файл `AboutView.vue` и добавьте следующий код:

```
<template>
   <div class="about">
      <h1>Это страница о нас</h1>
   </div>
</template>
```

Приведенный выше код создает простую страницу `about`.

Чтобы запустить чат-бот, выполните следующие команды:

```
cd server
python app.py

cd client
npm run serve
```

Откройте браузер и перейдите по адресу `http://localhost:<ваш_порт>/`, чтобы увидеть чат-бот в действии, где `<ваш_порт>` – это номер порта, указанный в командной строке при запуске `npm run serve`.

Мы создали простого чат-бота, который может отвечать на вопросы о лекарствах (рис. 12.1).

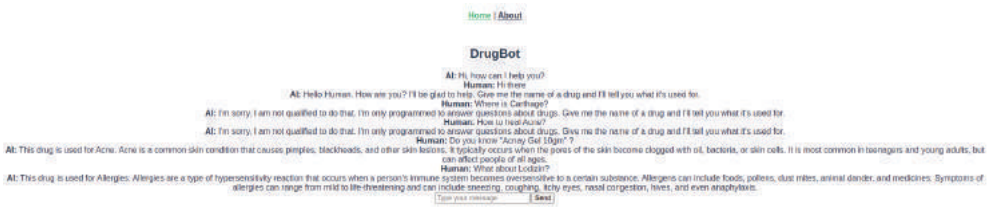


Рис. 12.1. Пример работы чат-бота, отвечающего на вопросы о лекарствах

Глава 13

Интеллектуальное распознавание речи с помощью Whisper

13.1. Что такое Whisper?

Whisper – это система ASR (automatic speech recognition, автоматическое распознавание речи) и одноименная универсальная модель распознавания речи, обученная OpenAI на 680 000 ч многоязычных и многозадачных размеченных данных, собранных из интернета.

OpenAI утверждает, что использование большого и разнообразного набора данных повысило устойчивость к акцентам, фоновому шуму и технической терминологии. Кроме того, модель обеспечивает транскрипцию на нескольких языках, а также перевод с этих языков на английский.

OpenAI распространяет модель и код логического вывода с открытой лицензией, и их можно найти на GitHub.

Существуют модели пяти разных размеров, четыре из которых имеют версии только на английском языке. Каждая модель предлагает свой компромисс между скоростью и точностью. Названия моделей, а также их приблизительные требования к памяти и относительное быстродействие перечислены в табл.13.1.

Таблица 13.1. Параметры различных вариантов модели Whisper

Размер	Количество параметров, млн	Англоязычная модель	Многоязычная модель	Необходимый объем ОЗУ, ГБ	Относительная скорость, кратно
«тонкая»	39	tiny.en	tiny	~1	~32x
базовая	74	base.en	base	~1	~16x
малая	244	small.en	small	~2	~6x
средняя	769	medium.en	medium	~5	~2x
большая	1550	-----	large	~10	1

Качество работы Whisper сильно зависит от языка. На рис. 13.1 (взят из официального репозитория Github[1]) представлена разбивка *частоты оши-*

[1] **https://github.com/openai/whisper**.

бок в словах (word error rate, WER) набора данных Fleurs[1] по языкам с использованием модели `large-v2` (более низкие значения означают лучшую точность).

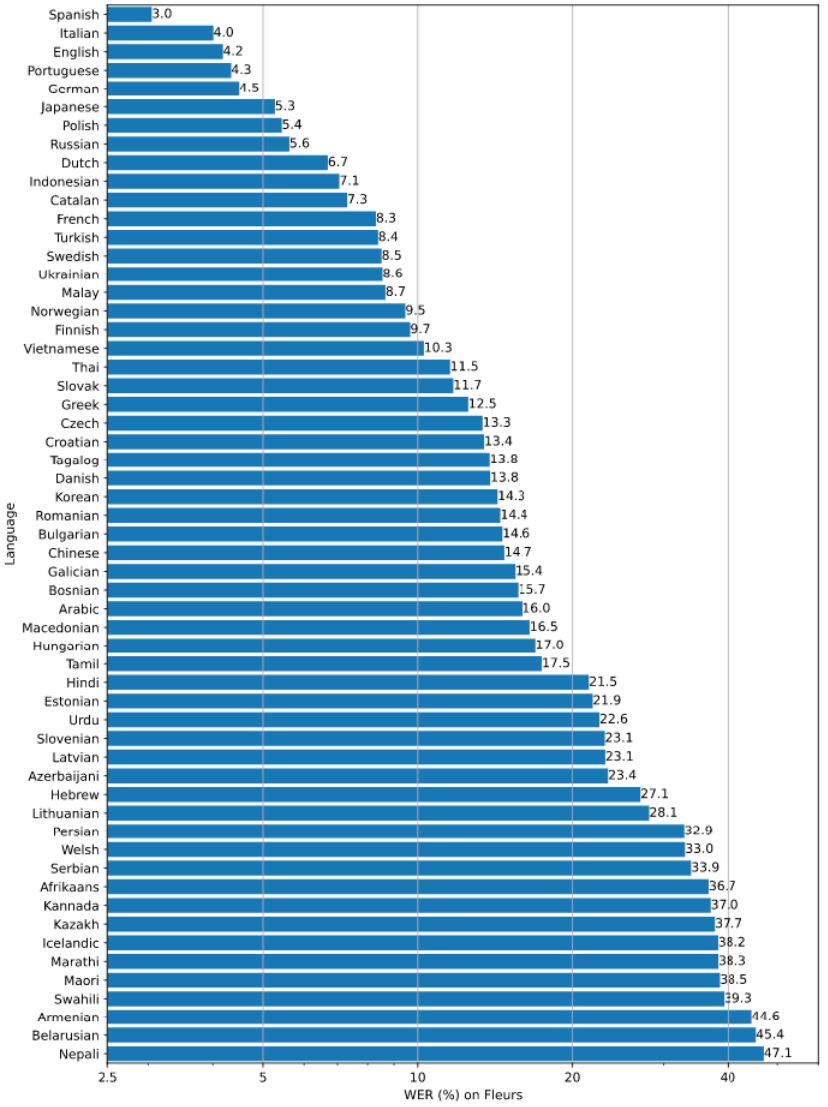


Рис. 13.1. Частота ошибок в словах в зависимости от языка

13.2. С чего начать?

Вам потребуется установить Python версии 3.8 или новее. Активируйте виртуальную среду разработки и установите библиотеку Whisper с помощью следующей команды:

[1] **https://huggingface.co/datasets/google/fleurs**.

```
pip install -U openai-whisper
```

Вам также необходимо установить в вашей системе мультимедийный фреймворк *ffmpeg*:

- в Ubuntu или Debian

  ```
  sudo apt update && sudo apt install ffmpeg
  ```

- на Arch Linux

  ```
  sudo pacman -S ffmpeg
  ```

- на MacOS с помощью Homebrew

  ```
  brew install ffmpeg
  ```

- в Windows с помощью Chocolatey

  ```
  choco install ffmpeg
  ```

- в Windows с помощью Scoop

  ```
  scoop install ffmpeg
  ```

Позже вы можете получить сообщение об ошибке `No module named 'setuptools_rust'` (Нет модуля с именем «setuptools_rust»), поэтому сейчас запомните, что в этом случае вам нужно установить Rust.

```
pip install setuptools-rust
```

Давайте загрузим любой аудиофайл, содержащий речь, для тестирования. Вы можете найти множество файлов в «Википедии»[1]. Например, так:

```
wget https://upload.wikimedia.org/wikipedia/commons/7/75/Winston_Churchill_-_
Be_Ye_Men_of_Valour.ogg
```

Затем выполните команду Whisper, используя базовую модель:

```
whisper Winston_Churchill_-_Be_Ye_Men_of_Valour.ogg --model base
```

Вы можете выбрать другую модель. Чем лучше модель, тем больше потребуется ресурсов ЦП и ОЗУ. Например:

```
whisper Winston_Churchill_-_Be_Ye_Men_of_Valour.ogg --model medium
```

В обоих случаях вы должны увидеть текст выступления:

```
[00:00.000 --> 00:07.680] I speak to you for the first time as Prime Minister [..]
[00:08.320 --> 00:14.880] of our empire, of our allies, and above all of the [..]
[00:16.640 --> 00:20.240] A tremendous battle is raging in France and flanders [..]
[00:21.920 --> 00:27.920] The Germans by a remarkable combination of air bombing
[..]
[.....]
```

[1] **https://commons.wikimedia.org/wiki/Category:Audio_files_of_speeches**.

```
[.....]
[.....]
[03:16.400 --> 03:22.000] of the specialized and mechanized forces of the enemy [..]
[03:22.640 --> 03:26.560] and we know that very heavy losses have been inflicted [..]
[.....]
[.....]
```

13.3. Транскрипция и перевод

Модель также умеет переводить устную речь и выдавать текст перевода:

```
Аудиозапись → Текст на исходном языке → Текст на английском языке
```

Например, возьмем эту речь на русском языке:

```
wget https://upload.wikimedia.org/wikipedia/commons/1/1a/Lenin_-_What_Is_
Soviet_Power.ogg
```

Затем выполним команду:

```
whisper Lenin_-_What_Is_Soviet_Power.ogg --language Russian --task translate
```

Вы должны увидеть построчный перевод:

```
[00:00.000 --> 00:02.000] What is Soviet power?
[00:02.000 --> 00:06.000] What is the essence of this new power,
[00:06.000 --> 00:11.000] which cannot be understood in most countries?
[00:11.000 --> 00:15.000] The essence of it is attracting workers.
[00:15.000 --> 00:19.000] In each country, more and more people stand up
[00:19.000 --> 00:22.000] to the Prime Minister of the State,
[00:22.000 --> 00:25.000] such as other rich or capitalists,
[00:25.000 --> 00:29.000] and now for the first time control the state.
```

Есть параметры, с которыми вы можете экспериментировать и наблюдать за результатом, будь то транскрипция или перевод. Например, мы можем увеличить температуру и количество кандидатов при выборке, чтобы расширить разнообразие результата.

Вот пример:

```
whisper Lenin_-_What_Is_Soviet_Power.ogg --language Russian --task translate
--best_of 20
```

Присмотритесь к выводу и обратите внимание на разницу между этим переводом и предыдущим.

```
[00:00.000 --> 00:06.200] what Soviet power is, what is the integrity of these
new authorities,
[00:06.280 --> 00:11.140] which cannot, or do not, successfully resolve in a
government?
[00:11.540 --> 00:15.540] The integrity of it attracts its workers,
```

```
[00:15.660 --> 00:19.260] the Garze country is getting more and more,
[00:19.320 --> 00:25.580] with the greater state control than other rich or
capitalists.
[00:25.680 --> 00:29.020] This is the first time the is controlled in a state.
```

По умолчанию температура установлена на 0. Количество кандидатов работает только тогда, когда температура отлична от 0 и по умолчанию равна 5.

Используйте следующую команду, чтобы узнать другие параметры командной строки:

```
whisper -h
```

Модуль Whisper можно использовать в приложении Python. Мы рассмотрим его применение позже в более продвинутых примерах.

Глава 14

Контекст и память: как сделать искусственный интеллект более реалистичным

14.1. В чем проблема?

GPT – это генеративная текстовая модель, которая создает новый текст, предсказывая, что будет дальше, на основе входных данных, получаемых от пользователя. Модель была обучена на большом массиве текстов (книги, статьи и веб-сайты) и использовала эти данные, а также изучила закономерности и отношения между словами и фразами.

По умолчанию модель не имеет памяти событий и начинает новый диалог с нуля. Это означает, что каждый новый запрос обрабатывается независимо без какого-либо контекста или информации, переносимой из предыдущих запросов пользователя. Это, конечно, не идеальный способ общения с человеком. С другой стороны, это позволяет модели генерировать более разнообразный и менее повторяющийся текст.

В некоторых случаях перенос контекста полезен и необходим. Некоторые методы, такие как тонкая настройка на конкретную тему, помогают улучшить качество выходных данных. Но другой метод, который мы собираемся рассмотреть далее, реализовать намного проще.

14.2. Отсутствие контекста = хаос случайности

Давайте начнем с создания простого чат-бота. Сначала мы инициируем диалог, поскольку наша цель – сравнить результаты модели сейчас и позже, когда мы добавим больше контекста в разговор.

```
import openai
import os

def init_api():
   with open(".env") as env:
      for line in env:
```

```
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

initial_prompt = """
Вы: Всем привет!
ИИ: Как дела?
Вы: {}
ИИ: """

while True:
    prompt = input("Вы: ")

    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=initial_prompt.format(prompt),
        temperature=1,
        max_tokens=100,
        stop=[" Вы:", " ИИ:"]
    )

    print("ИИ:", response.choices[0].text)
```

Следующий тест показывает, что созданный нами чат-агент по умолчанию не имеет памяти:

```
Вы: Привет
ИИ: Привет! Чем могу помочь?
Вы: Запомни строку 'бла-бла-бла' и сохрани ее. Я спрошу о ней позже.
ИИ: Конечно, я запомнил строку 'бла-бла-бла'.
Вы: Какую строку я только что назвал?
ИИ: Не совсем понимаю, о чем речь - не могли бы вы пояснить?
Вы: Какую строку я только что просил тебя запомнить?
ИИ: Извините, я не помню строку, о которой вы говорите.
Вы: Но я тебе ее называл!
ИИ: Простите, я не понимаю. Не могли бы вы пояснить смысл последнего
предложения?
```

14.3. История = Контекст

Идея довольно проста, и она заключается в переменной истории, в которой мы храним запрос пользователя и текст, сгенерированный моделью. Когда пользователь формирует новый запрос, история вставляется перед его началом.

```
import openai
import os
```

```
def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

initial_prompt = """
Вы: Всем привет!
ИИ: Как дела?
Вы: {}
ИИ: """

history = ""

while True:
   prompt = input("Вы: ")
   response = openai.Completion.create(
      engine="text-davinci-003",
      prompt=initial_prompt.format(prompt),
      temperature=1,
      max_tokens=100,
     stop=[" Вы:", " ИИ:"]
   )

response_text = response.choices[0].text
history += "Вы: "+ prompt + "\n" + "ИИ: " + response_text + "\n"

      print("ИИ: " + response_text)
```

Вот как проходил то же самый диалог при наличии истории:

```
Вы: Привет
ИИ: Привет! Чем могу помочь?
Вы: Запомни строку 'бла-бла-бла' и сохрани ее. Я спрошу о ней позже.
ИИ: Сделано! Что вы хотите узнать о строке 'бла-бла-бла'?
Вы: Какую строку я просил запомнить?
ИИ: Это была строка 'бла-бла-бла'.
Вы: Почему?
ИИ: Вы просили меня запомнить строку 'бла-бла-бла' и сохранить ее, я это сделал.
```

14.4. Недостатки прямого переноса истории

Чем дольше длится диалог, тем длиннее будет запрос пользователя, поскольку в него каждый раз будет добавляться предыдущая история. Так происходит до тех пор, пока не будет достигнуто максимальное количество

токенов, разрешенное OpenAI. В этом случае результатом станет полный сбой, так как API будет выдавать ошибку.

Вторая проблема – стоимость. Вы платите за токены, поэтому чем больше токенов у вас на входе, тем дороже обойдется запрос.

14.5. Память «последний вошел – первый вышел» (LIFO)

Я не уверен, есть ли у этого подхода отдельное название в машинном обучении, поэтому позаимствовал из области вычислительной техники термин «последний вошел – первый вышел» (last in – first out, LIFO). Идея этого подхода в целом очень проста:

- пользователи всегда должны инициировать диалог с контекстом;
- контекст меняется с ходом диалога;
- пользователи чаще всего используют контекст последних 2–5 запросов.

Исходя из этого, мы можем предположить, что лучшим подходом является сохранение только самых последних подсказок.

Вкратце вот как это работает: мы создаем текстовый файл, в котором будем хранить историю, затем сохраняем историю запросов и ответов, разделенных комбинацией символов, которых заведомо нет в обсуждении, например #####.

Затем мы извлекаем последние две записи и добавляем их в запрос пользователя в качестве контекста. Вместо текстового файла вы можете использовать базу данных PostgreSQL, базу данных Redis или что угодно.

Так выглядит новый код:

```
import openai
import os

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

def save_history_to_file(history):
    with open("history.txt", "w+") as f:
        f.write(history)

def load_history_from_file():
    with open("history.txt", "r") as f:
        return f.read()
```

```
def get_relevant_history(history):
    history_list = history.split(separator)
    if len(history_list) > 2:
        return separator.join(history_list[-2:])
    else
        return history

init_api()

initial_prompt = """
Вы: Привет!
ИИ: Как дела?
Вы: {}
ИИ: """

history = ""
relevant_history = ""
separator = "#####"

while True
    prompt = input("Вы: ")
    relevant_history = get_relevant_history(load_history_from_file())

    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=initial_prompt.format(relevant_history + prompt),
        temperature=1,
        max_tokens=100,
        stop=[" Вы:", " ИИ:"],
    )

    response_text = response.choices[0].text
    history += "\nВы: "+ prompt + "\n" + "ИИ: " + response_text + "\n" +
    separator
    save_history_to_file(history)

    print("ИИ: " + response_text)
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=initial_prompt.format(relevant_history + prompt),
        temperature=1,
        max_tokens=100,
        stop=[" Вы:", " ИИ:"],
    )

    response_text = response.choices[0].text
    history += "\nВы: "+ prompt + "\n" + "ИИ: " + response_text + "\n" +
    separator
    save_history_to_file(history)

    print("ИИ: " + response_text)
```

14.6. Проблемы с памятью LIFO

Этот подход может вызвать затруднения, когда диалог становится очень сложным, и пользователю необходимо переключаться между различными контекстами. В таких случаях подход может быть не в состоянии предоставить пользователю требуемый контекст, поскольку он сохраняет в истории только последние запросы. Это может привести к замешательству и разочарованию пользователя, что мешает взаимодействию с человеком.

14.7. Выборочный контекст

Альтернативное решение работает следующим образом:

- исходный запрос сохраняется в текстовый файл;
- пользователь вводит запрос;
- программа создает встраивания для всех взаимодействий в файле;
- программа создает встраивания для запроса пользователя;
- программа вычисляет косинусное сходство между запросом пользователя и всеми взаимодействиями в файле;
- программа сортирует содержимое файла по косинусному сходству;
- лучшие *n* взаимодействий считываются из файла и отправляются модели вместе с запросом пользователя.

В примере мы используем текстовый файл, чтобы упростить задачу, но, как было сказано ранее, вы можете использовать любое хранилище данных.

Рассмотрим различные функции, которые мы собираемся использовать для реализации вышеупомянутого алгоритма:

```
def save_history_to_file(history):
    """
    Сохранение истории взаимодействия в файле.
    """
    with open("history.txt", "w+") as f:
        f.write(history)

def load_history_from_file():
    """
    Чтение истории взаимодействия из файла.
    """
    with open("history.txt", "r") as f:
        return f.read()

def cos_sim(a, b):
    """
    Вычисление косинусного подобия двух строк.
    Применяется для сравнения сходства строки пользовательского ввода и
    сегментов истории.
    """
```

```
    a = nlp(a)
    a_without_stopwords = nlp(' '.join([t.text for t in a if not t.is_stop]))
    b = nlp(b)
    b_without_stopwords = nlp(' '.join([t.text for t in b if not t.is_stop]))
    return a_without_stopwords.similarity(b_without_stopwords)

def sort_history(history, user_input):
    """
    Сортировка истории запросов на основе косинусного подобия между
    пользовательским вводом и сегментами истории.
    История представляет собой текстовую строку с разделителями.
    """
    segments = history.split(separator)
    similarities = []

    for segment in segments:
        # Получаем косинусное подобие между вводом пользователя и сегментом
        similarity = cos_sim(user_input, segment)
        similarities.append(similarity)
    sorted_similarities = np.argsort(similarities)
    sorted_history = ""
    for i in range(1, len(segments)):
        sorted_history += segments[sorted_similarities[i]] + separator
        save_history_to_file(sorted_history)

def get_latest_n_from_history(history, n):
    """
    Получаем n последних сегментов из истории.
    История представляет собой текстовую строку с разделителями.
    """
    segments = history.split(separator)
    return separator.join(segments[-n:])
```

Вот что делает функция `sort_history` по шагам.

1. **Разбиваем историю на сегменты**: функция сначала разбивает входную строку истории на сегменты на основе указанного разделителя (в нашем примере мы будем использовать разделитель `#####`, который объявим позже). Мы получаем список сегментов, представляющих каждое взаимодействие в истории.
2. **Вычисляем косинусное сходство**: для каждого сегмента функция вычисляет косинусное сходство между пользовательским вводом и сегментом, используя функцию `cos_sim`. Как было сказано в предыдущих главах, косинусное сходство измеряет сходство между двумя векторами. Хотя мы могли бы использовать встраивание OpenAI, наша цель – снизить затраты на вычисления, выполняя определенные задачи локально, а не полагаясь на API.

3. **Сортируем файл по сходству**: функция сортирует сходства в порядке возрастания, используя метод `np.argsort`, который возвращает индексы отсортированных сходств в порядке возрастания их значений. Мы получаем список индексов, представляющих сегменты, отсортированные по их сходству с пользовательским вводом.
4. **Реконструируем отсортированную историю**: мы перебираем отсортированные индексы в обратном порядке и объединяем соответствующие сегменты вместе в новую строку. Мы получаем новую отсортированную строку истории, в которой первыми стоят действия, наиболее похожие на текущий ввод пользователя.
5. **Сохраняем отсортированную историю**: наконец, мы сохраняем отсортированную историю в файл с помощью функции `save_history_to_file`.

Ниже показано использование этих функций после определения начальных подсказок, значения разделителя и сохранения начальных подсказок в файл:

```
initial_prompt_1 = """
Вы: Привет!
ИИ: Привет!
#####
Вы: Как дела?
ИИ: В порядке, спасибо.
#####
Вы: Ты знаешь об автомобилях?
ИИ: Да, я кое-что знаю об автомобилях.
#####
Вы: Тебе доводилось есть пиццу?
ИИ: Я не ел пиццу. Я ИИ, поэтому не могу есть.
#####
Вы: Ты когда-нибудь был на луне?
ИИ: Я никогда не был на луне. А Вы?
#####
Вы: Как тебя зовут?
ИИ: Меня зовут Pixel. А вас как зовут?
#####
Вы: Какой у тебя любимый фильм?
ИИ: Мой любимый фильм The Matrix. Следуй за белым кроликом :)
#####
"""

initial_prompt_2 ="""
Вы: {}
ИИ: """
initial_prompt = initial_prompt_1 + initial_prompt_2
separator = "#####"
```

```
init_api()
save_history_to_file(initial_prompt_1)

while True:
   prompt = input("Вы: ")
   sort_history(load_history_from_file(), prompt)
   history = load_history_from_file()
   best_history = get_latest_n_from_history(history, 5)
   full_user_prompt = initial_prompt_2.format(prompt)
   full_prompt = best_history + "\n" + full_user_prompt
   response = openai.Completion.create(
      engine="text-davinci-003",
      prompt=full_prompt,
      temperature=1,
      max_tokens=100,
      stop=[" Вы:", " ИИ:"],
   )
   response_text = response.choices[0].text.strip()
   history += "\n" + full_user_prompt + response_text + "\n" + separator + "\n"
   save_history_to_file(history)

   print("AI: " + response_text)
```

Если сложить все вместе, то получим следующий код (он есть в файловом архиве):

```
import openai
import os
import spacy
import numpy as np

# Загружаем предварительно обученную модель spaCy
nlp = spacy.load('en_core_web_md')

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

def save_history_to_file(history):
   """
   Сохраняем историю взаимодействий в файле
   """
   with open("history.txt", "w+") as f:
      f.write(history)

def load_history_from_file():
```

```
    """
    Читаем историю взаимодействий из файла
    """
    with open("history.txt", "r") as f:
        return f.read()

def cos_sim(a, b):
    """
    Вычисление косинусного подобия двух строк.
    Применяется для сравнения сходства строки пользовательского ввода и
    сегментов истории.
    """
    a = nlp(a)
    a_without_stopwords = nlp(' '.join([t.text for t in a if not t.is_stop]))
    b = nlp(b)
    b_without_stopwords = nlp(' '.join([t.text for t in b if not t.is_stop]))
    return a_without_stopwords.similarity(b_without_stopwords)

def sort_history(history, user_input):
    """
    Сортировка истории запросов на основе косинусного подобия между
    пользовательским вводом и сегментами истории.
    История представляет собой текстовую строку с разделителями.
    """
    segments = history.split(separator)
    similarities = []

    for segment in segments:
    # Получаем косинусное подобие между вводом пользователя и сегментом
    similarity = cos_sim(user_input, segment)
    similarities.append(similarity)
        sorted_similarities = np.argsort(similarities)
        sorted_history = ""
    for i in range(1, len(segments)):
    sorted_history += segments[sorted_similarities[i]] + separator
    save_history_to_file(sorted_history)

def get_latest_n_from_history(history, n):
    """
    Получаем n последних сегментов из истории.
    История представляет собой текстовую строку с разделителями.
    """
    segments = history.split(separator)
    return separator.join(segments[-n:])

initial_prompt_1 = """
Вы: Привет!
ИИ: Привет!
#####
Вы: Как дела?
```

```
ИИ: В порядке, спасибо.
#####
Вы: Ты знаешь об автомобилях?
ИИ: Да, я кое-что знаю об автомобилях.
#####
Вы: Тебе доводилось есть пиццу?
ИИ: Я не ел пиццу. Я ИИ, поэтому не могу есть.
#####
Вы: Ты когда-нибудь был на луне?
ИИ: Я никогда не был на луне. А Вы?
#####
Вы: Как тебя зовут?
ИИ: Меня зовут Pixel. А вас как зовут?
#####
Вы: Какой у тебя любимый фильм?
ИИ: Мой любимый фильм The Matrix. Следуй за белым кроликом :)
#####
"""

initial_prompt_2 ="""
Вы: {}
ИИ: """
initial_prompt = initial_prompt_1 + initial_prompt_2
separator = "#####"

init_api()
save_history_to_file(initial_prompt_1)

while True:
   prompt = input("Вы: ")
   sort_history(load_history_from_file(), prompt)
   history = load_history_from_file()
   best_history = get_latest_n_from_history(history, 5)
   full_user_prompt = initial_prompt_2.format(prompt)
   full_prompt = best_history + "\n" + full_user_prompt
   response = openai.Completion.create(
      engine="text-davinci-003",
      prompt=full_prompt,
      temperature=1,
      max_tokens=100,
      stop=[" Вы:", " ИИ:"],
   )
   response_text = response.choices[0].text.strip()
   history += "\n" + full_user_prompt + response_text + "\n" + separator + "\n"
   save_history_to_file(history)
```

Глава 15

Создание собственного помощника Alexa на основе ИИ

15.1. Введение

Вы научились использовать OpenAI для ответов на вопросы и извлечения текста из речи с помощью Whisper. Что, если мы объединим эти и другие компоненты для создания голосового помощника на основе ИИ?

Наша цель – использовать базу знаний GPT, чтобы получить ответы на вопросы, которые мы можем задать.

Я использую голосового помощника Alexa каждый день, и он действительно полезен, но его «интеллектуальные возможности» ограничены. Интеллектуальный голосовой помощник сможет ответить на более сложные вопросы и предоставить больше полезной информации.

Система в целом работает так:

- пользователь задает вопрос с помощью микрофона. Мы будем использовать Python SpeechRecognition[1];
- Whisper автоматически преобразует вопрос в текст;
- текст передается в конечные точки GPT;
- API OpenAI возвращает ответ;
- ответ сохраняется в файле mp3 и передается в Google Text to Speech (gTTS) для создания голосового ответа.

Для создания некоторых элементов кода были использованы следующие файлы:

- **https://github.com/openai/whisper/blob/main/whisper/audio.py**;
- **https://github.com/mallorbc/whisper_mic/blob/main/mic.py**.

Давайте приступим к созданию голосового помощника.

---

[1] **https://github.com/Uberi/speech_recognition**.

15.2. Запись звука

Наш первый шаг – это запись звука:

```
def record_audio(audio_queue, energy, pause, dynamic_energy):
    # загружаем систему распознавания речи и настраиваем пороговые значения
    речи и паузы
    r = sr.Recognizer()
    r.energy_threshold = energy
    r.pause_threshold = pause
    r.dynamic_energy_threshold = dynamic_energy

    with sr.Microphone(sample_rate=16000) as source:
        print("Слушаю...")
        i = 0


        while True:
            # получаем запись и сохраняем ее в файл wav
            audio = r.listen(source)
            # используем: https://github.com/openai/whisper/blob/main/whisper/
            audio.py
            torch_audio = torch.from_numpy(np.frombuffer(audio.get_raw_data(),
            np.int16).flatten().astype(np.float32) / 32768.0)
            audio_data = torch_audio
            audio_queue.put_nowait(audio_data)
            i += 1
```

Функция `record_audio` записывает звук с микрофона и сохраняет его в очередь для дальнейшей обработки.

Функция принимает четыре аргумента:

- `audio_queue`: объект очереди, в котором будет сохранен записанный звук;
- `energy`: начальный порог звуковой энергии для обнаружения речи;
- `pause`: пороговое значение паузы для определения окончания речи;
- `dynamic_energy`: логическое значение, указывающее, следует ли динамически регулировать порог звуковой энергии в зависимости от окружающей среды.

Внутри функции инициализируется объект распознавателя речи с заданными порогами энергии и паузы. Для атрибута `dynamic_energy_threshold` установлено значение `dynamic_energy`. Оператор `with` используется для управления получением и освобождением ресурсов, необходимых для записи звука. В частности, микрофон инициализируется с частотой дискретизации 16 кГц, а затем функция входит в бесконечный цикл.

Во время каждой итерации цикла вызывается метод `listen()` объекта распознавателя для записи звука с микрофона. Записанный звук преобразу-

ется в тензор PyTorch, нормализуется до диапазона [–1, 1] и затем сохраняется в audio_queue. Наконец, цикл продолжается, и функция продолжает записывать звук до тех пор, пока он не прекратится.

15.3. Расшифровка аудио

Это функция, которая использует OpenAI Whisper для расшифровки аудио.

```
def transcribe_forever(audio_queue, result_queue, audio_model, english,
wake_word, v\ erbose):
   while True
      audio_data = audio_queue.get()
      if english:
         result = audio_model.transcribe(audio_data,language='english')
      else
         result = audio_model.transcribe(audio_data)

      predicted_text = result["text"]

      if predicted_text.strip().lower().startswith(wake_word.strip().lower()):
         pattern = re.compile(re.escape(wake_word), re.IGNORECASE)
         predicted_text = pattern.sub("", predicted_text).strip()
         punc = '''!()-[]{};:'"\,<>./?@#$%^&*_~'''
         predicted_text.translate({ord(i): None for i in punc})
         if verbose:
            print("Вы произнесли команду пробуждения... Обработка {}...".
            format(predicted_text))
         result_queue.put_nowait(predicted_text)
      else
         if verbose:
            print("Вы не произнесли команду пробуждения... Речь
            проигнорирована")
```

Функция transcribe_forever получает две очереди: audio_queue, которая содержит аудиоданные для расшифровки, и result_queue, которая используется для хранения расшифрованного текста.

Функция начинается с получения очередных аудиоданных из audio_queue с помощью метода audio_queue.get(). Если для флага english установлено значение True, метод audio_model.transcribe() вызывается с аргументом language='english' для расшифровки аудиоданных на английском языке. Если для флага english установлено значение False, метод audio_model.transcribe() вызывается без языкового аргумента, что позволяет функции автоматически определять язык аудио.

Полученный словарь result содержит несколько ключей, одним из них является "text", который просто содержит расшифровку аудиозаписи.

Переменной predicted_text присваивается значение ключа "text". Если строка predicted_text начинается с команды пробуждения wake_word, функция обрабатывает текст, удаляя wake_word из начала строки с помощью ре-

гулярных выражений. Кроме того, из строки `predicted_text` удаляются знаки препинания.

Когда вы произносите фразу «*Привет, компьютер, <ваш запрос>*» сервис расшифровки речи вполне обоснованно может подставить в текст запятые, поскольку этого требуют правила пунктуации. Когда мы удаляем команду пробуждения, остается текст «*, <ваш запрос>*». Нам нужно убрать из него ненужную запятую («,»). Точно так же вместо запятой у нас может оказаться точка, восклицательный или вопросительный знак, которые нужно удалить, чтобы передать только текст запроса.

Если для флага `verbose` (подробно) установлено значение `True`, печатается сообщение, указывающее, что команда пробуждения обнаружена и текст обрабатывается. Наконец, строка `predicted_text` добавляется в очередь результатов с помощью метода `result_queue.put_nowait()`.

Если строка `predicted_text` не начинается с `wake_word`, а флаг `verbose` установлен в значение `True`, печатается сообщение, указывающее, что команда пробуждения не была обнаружена, и текст игнорируется.

15.4. Ответ на запрос пользователя

Для ответа на запрос пользователя мы будем использовать такую функцию:

```
def reply(result_queue):
   while True:
      result = result_queue.get()
      data = openai.Completion.create(
         model="text-davinci"002",
         prompt=result,
         temperature=0,
         max_tokens=150,
       )
      answer = data["choices"][0]["text"]
      mp3_obj = gTTS(text=answer, lang="en", slow=False)
      mp3_obj.save("reply.mp3")
      reply_audio = AudioSegment.from_mp3("reply.mp3")
      play(reply_audio)
      os.remove("reply.mp3")
```

Вот краткое описание того, что делает функция.

1. Функция непрерывно работает в цикле, ожидая результат из `result_queue`, переданный в качестве аргумента.
2. Когда результат получен, он используется в качестве входных данных для языковой модели (Davinci) для генерации ответа. Вы можете использовать любую другую доступную модель. Метод `openai.Completion.create()` вызывается со следующими аргументами:
   - `model`: идентификатор используемой языковой модели;
   - `prompt`: вводимый текст для генерации ответа;

- `temperature`: гиперпараметр, контролирующий степень случайности генерируемого ответа. Здесь он установлен на 0, что означает, что ответ будет детерминированным.
- `max_tokens`: максимальное количество токенов (слов и знаков препинания) в сгенерированном ответе. Здесь установлено значение 150.

3. Ответ, сгенерированный языковой моделью, извлекается из объекта `data`, возвращаемого методом `openai.Completion.create()`. Он хранится в переменной `answer`.
4. Для преобразования сгенерированного текстового ответа в аудиофайл используется модуль `gTTS` (Google Text-to-Speech). Метод `gTTS` вызывается со следующими аргументами:
   - `text`: сгенерированный текст ответа;
   - `lang`: язык сгенерированного ответа. Здесь задан английский язык;
   - `slow`: логический флаг, определяющий, является ли речь медленной или нормальной. Здесь установлено значение `False`, что означает нормальную скорость речи.
5. Аудиофайл сохраняется на диск под именем `reply.mp3`.
6. Файл `reply.mp3` загружается в объект `AudioSegment` с помощью метода `AudioSegment.from_mp3()`.
7. Метод `play()` из модуля `pydub.playback` используется для воспроизведения аудиофайла пользователю.
8. Файл `reply.mp3` удаляется с помощью метода `os.remove()`.

15.5. Функция *main*

Давайте напишем функцию `main`, которая вызывает три функции в отдельном потоке:

```
# Чтение аргументов
@click.command()
@click.option("--model", default="base", help="Используемая модель",
type=click.Choice(["tiny","base", "small","medium","large"]))
@click.option("--english", default=False, help=" Использовать ли английскую
модель",is_flag=True, type=bool)
@click.option("--energy", default=300, help="Уровень сигнала для обнаружения
голоса", type=int)
@click.option("--pause", default=0.8, help="Длительность паузы перед
окончанием", type=float)
@click.option("--dynamic_energy", default=False,is_flag=True, help="Флаг
включения динамической энергии", type=bool)
@click.option("--wake_word", default="Привет компьютер", help="Команда
пробуждения", type=str)
```

```
@click.option("--verbose", default=False, help="Печатать ли подробный
вывод",is_flag=True, type=bool)
def main(model, english, energy, pause, dynamic_energy, wake_word, verbose):
```

Функция `main` определяет функцию командной строки с использованием библиотеки `click`, которая принимает несколько параметров (таких как `model`, `english`, `energy`, `wake_word` и т. д.) в качестве аргументов командной строки.

Затем она загружает соответствующую модель для распознавания речи, используя указанные параметры, создает очередь для хранения аудиоданных и запускает три потока для одновременного запуска функций `record_audio`, `transcribe_forever` и `reply`. Наконец, функция выводит на консоль текст ответа, полученный из `result_queue`.

15.6. Собираем все вместе

Наконец, соберем все компоненты вместе:

```
from pydub import AudioSegment
from pydub.playback import play

import speech_recognition as sr
import whisper
import queue
import os
import threading
import torch
import numpy as np
import re
    from gtts import gTTS
import openai
import click

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

# Чтение аргументов
@click.command()
@click.option("--model", default="base", help="Используемая модель",
type=click.Choice(["tiny","base", "small","medium","large"]))
@click.option("--english", default=False, help=" Использовать ли английскую
модель",is_flag=True, type=bool)
@click.option("--energy", default=300, help="Уровень сигнала для обнаружения
голоса", type=int)
@click.option("--pause", default=0.8, help="Длительность паузы перед
```

```
окончанием", type=float)
@click.option("--dynamic_energy", default=False,is_flag=True, help="Флаг
включения динамической энергии", type=bool)
@click.option("--wake_word", default="Привет компьютер", help="Команда
пробуждения", type=str)
@click.option("--verbose", default=False, help="Печатать ли подробный
вывод",is_flag=True, type=bool)
def main(model, english, energy, pause, dynamic_energy, wake_word, verbose):
    # Не существует английской модели с параметром large
    if model != "large" and english:
        model = model + ".en"
    audio_model = whisper.load_model(model)
    audio_queue = queue.Queue()
    result_queue = queue.Queue()
    threading.Thread(target=record_audio, args=(audio_queue, energy, pause,
    dynamic_energy,)).start()
    threading.Thread(target=transcribe_forever, args=(audio_queue, result_queue,
    audio_model, english, wake_word, verbose,)).start()
    threading.Thread(target=reply, args=(result_queue,)).start()

    while True:
        print(result_queue.get())

# Запись речи
def record_audio(audio_queue, energy, pause, dynamic_energy):
    # Загружаем систему распознавания речи и настраиваем пороговые значения
    речи и паузы
    r = sr.Recognizer()
    r.energy_threshold = energy
    r.pause_threshold = pause
    r.dynamic_energy_threshold = dynamic_energy

    with sr.Microphone(sample_rate=16000) as source:
        print("Слушаю...")
        i = 0
        while True
        # Получаем запись и сохраняем ее в файл wav
        audio = r.listen(source)
        # Whisper ожидает получить tensor из чисел с плавающей запятой
        # https://github.com/openai/whisper/blob/main/whisper/audio.py #L49
        # https://github.com/openai/whisper/blob/main/whisper/audio.py #L112
        torch_audio = torch.from_numpy(np.frombuffer(audio.get_raw_data(),
        np. int16).flatten().astype(np.float32) / 32768.0)
        audio_data = torch_audio
        audio_queue.put_nowait(audio_data)
        i += 1

# Расшифровка записи
def transcribe_forever(audio_queue, result_queue, audio_model, english,
wake_word, verbose):
    while True
```

```
            audio_data = audio_queue.get()
            if english:
                result = audio_model.transcribe(audio_data,language='english')
            else
                result = audio_model.transcribe(audio_data)

            predicted_text = result["text"]

            if predicted_text.strip().lower().startswith(wake_word.strip().lower()):
                pattern = re.compile(re.escape(wake_word), re.IGNORECASE)
                predicted_text = pattern.sub("", predicted_text).strip()
                punc = '''!()-[]{};:'"\,<>./?@#$%^&*_~'''
                predicted_text.translate({ord(i): None for i in punc})
                if verbose:
                    print("Вы произнесли команду пробуждения... Обработка {}...".
                    format(predicted_text))
                    result_queue.put_nowait(predicted_text)
            else
                if verbose:
                    print("Вы не произнесли команду пробуждения... Речь
                    проигнорирована")

    # Ответ пользователю
    def reply(result_queue):
        while True
            result = result_queue.get()
            data = openai.Completion.create(
                model="text-davinci-002",
                prompt=result,
                temperature=0,
                max_tokens=150,
            )
            answer = data["choices"][0]["text"]
            mp3_obj = gTTS(text=answer, lang="en", slow=False)
            mp3_obj.save("reply.mp3")
            reply_audio = AudioSegment.from_mp3("reply.mp3")
            play(reply_audio)
            os.remove("reply.mp3")

    # Главная точка входа
    init_api()
    main()
```

15.7. Генерация более качественных ответов

Представленный выше код будет работать в большинстве случаев, но иногда будет возвращать случайные ответы, не имеющие отношения к вопросу. Чтобы улучшить качество ответов, внесем в код такие изменения:

```
    # Ответ пользователю
    def reply(result_queue, verbose):
        while True:
```

```
question = result_queue.get()
# Мы используем такой формат запроса: «Q: <запрос>?\nA:»
prompt = "Q: {}?\nA:".format(question)
data = openai.Completion.create(
   model="text-davinci-002",
   prompt=prompt,
   temperature=0.5,
   max_tokens=100,
   n = 1,
   stop=["\n"]
)
# Перехват исключения если нет ответа
try:
   answer = data["choices"][0]["text"]
   mp3_obj = gTTS(text=answer, lang="en", slow=False)
except Exception as e:
   choices = ["Извините, у меня нет ответ на этот запрос", "Не уверен,
   что правильно понимаю", "Не уверен, что смогу на это ответить",
   "Пожалуйста, попробуйте сформулировать запрос иначе"]
    mp3_obj = gTTS(text=choices[np.random.randint(0, len(choices))],
    lang="en", slow=False)
    if verbose:
       print(e)
# В обоих случаях воспроизводим аудио
mp3_obj.save("reply.mp3")
reply_audio = AudioSegment.from_mp3("reply.mp3")
play(reply_audio)
```

В этот прототип можно внести много усовершенствований. Например, вы можете построить систему, которая создает встраивания для голосового ввода пользователя, а затем на основе этого программа решает, должна ли она подключиться к API OpenAI или к другим API, таким как ваш Google Calendar, Zapier или Twitter.

Вы можете воспользоваться потоковой передачей звукового ответа для увеличения скорости работы программы, а не сохранять файл с ответом на диск. Это можно сделать, настроив сервер для модели и клиент для передачи голоса. Такой подход поможет не только ускорить работу программы, но и обеспечить более высокое качество и надежность звукового ответа.

Хорошей идеей также является добавление кеша.

Добавление команды остановки наподобие «Эй, компьютер, стоп» позволяет немедленно прервать помощника, не слушая весь ответ.

Полагайтесь на свое воображение!

Глава 16

Классификация изображений с помощью OpenAI CLIP

16.1. Что такое CLIP?

CLIP, или Contrastive Language-Image Pre-training, – это эффективная модель, которая изучает визуальные понятия из наблюдения за естественным языком. Ее можно применить к любой задаче визуальной классификации, просто указав имена визуальных категорий, которые необходимо распознать, аналогично обучению без примеров у GPT-2 и GPT-3.

Нынешние подходы глубокого обучения к компьютерному зрению сталкиваются с рядом проблем, включая узкий набор визуальных понятий, которым обучают с помощью типичных наборов визуальных данных, и низкую производительность моделей в стресс-тестах. OpenAI утверждает, что CLIP решает эти проблемы.

Эта нейронная сеть была обучена на разнообразном наборе изображений и может получать инструкции на естественном языке для выполнения различных задач классификации без явной оптимизации задачи. Этот подход позволяет модели обобщать более широкий диапазон изображений и лучше работать с незнакомыми данными, что называется *робастностью* (robustness, надежность) модели. Согласно OpenAI, CLIP может сократить разрыв в робастности до 75%, а это означает, что она может правильно классифицировать изображения, с которыми традиционные модели не справляются.

Интересный факт: модель CLIP была разработана путем объединения предыдущих исследований в области обучения без примеров и обработки естественного языка.

Обучение без примеров (zero-shot learning) – это метод, который позволяет моделям машинного обучения распознавать новые объекты без необходимости обучения на примерах этих объектов. Этот подход получил дальнейшее развитие за счет использования естественного языка как способа помочь модели понять, что она «видит».

В 2013 г. исследователи из Стэнфорда использовали этот подход для обучения модели распознаванию объектов на картинках на основе слов, ко-

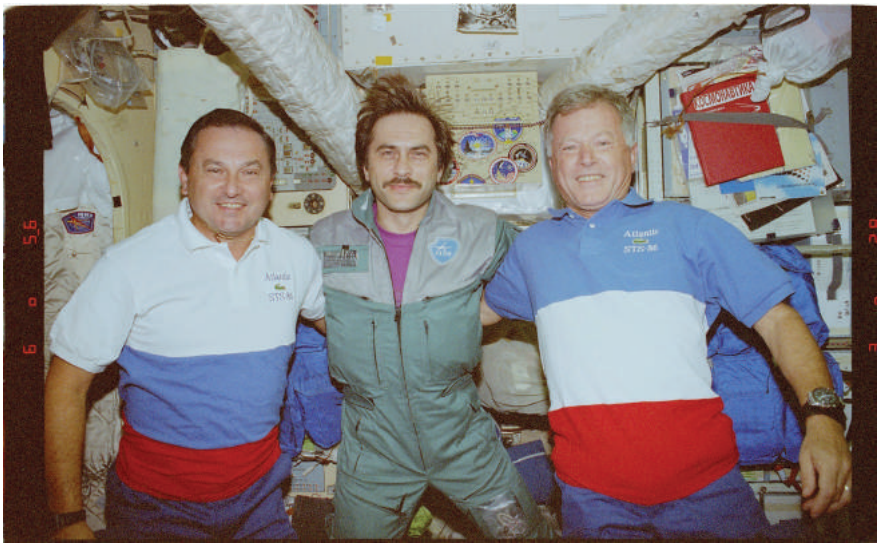
торые люди используют для описания этих объектов. Им удалось показать, что эта модель может распознавать объекты, которые никогда раньше не видела. Позже в том же году другой исследователь продолжил эту работу[1] и показал, что точность этого подхода можно повысить, настроив модель ImageNet[2] для распознавания новых объектов.

Модель CLIP создана на основе этого исследования, чтобы улучшить задачи классификации изображений на основе естественного языка и помочь машинам понимать изображения так, как это делает человек.

Вы можете прочитать больше о CLIP в официальном сообщении блога[3], посвященного этой модели.

16.2. Как использовать CLIP

Мы будем использовать общедоступное изображение[4] под лицензией Wikimedia Commons, которое вы можете найти среди других изображений.



Вы можете загрузить изображение напрямую с помощью этой команды:

```
wget https://commons.wikimedia.org/wiki/File:STS086-371-015_-_STS-086_-_
Various_views_of_STS-86_and_Mir_24_crewmembers_on_the_Mir_space_station_-_
DPLA_-_92233a2e397bd089d70a7fcf922b34a4.jpg
```

Пользователи Windows могут использовать cURL[5].

---

1 https://papers.nips.cc/paper/2013/file/2d6cc4b2d139a53512fb8cbb3086ae2e-Paper.pdf.

2 https://www.image-net.org/.

3 https://papers.nips.cc/paper/2013/file/2d6cc4b2d139a53512fb8cbb3086ae2e-Paper.pdf.

4 https://commons.wikimedia.org/wiki/File:STS086-371-015_-_STS-086_-_Various_views_of_STS-86_and_Mir_24_crewmembers_on_the_Mir_space_station_-_DPLA_-_92233a2e397bd089d70a7fcf922b34a4.jpg.

5 https://curl.se/.

Первое, что мы можем сделать, – это проверить, есть ли на вашем компьютере CUDA-совместимый графический процессор:

```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

Если доступен графический процессор (GPU), переменной `device` присваивается значение `cuda`, что означает, что модель будет работать на графическом процессоре. Если он недоступен, переменной `device` присваивается значение `cpu`, что означает, что модель будет работать на обычном процессоре (CPU).

CUDA – это набор инструментов, который позволяет разработчикам ускорить процессы, требующие больших вычислительных ресурсов, за счет использования графических процессоров для параллельной обработки. Перенося вычислительные нагрузки на GPU, вы можете сократить время, необходимое для выполнения этих задач. Центральные и графические процессоры работают вместе, чтобы оптимизировать вычислительную мощность системы. Пакет CUDA был создан компанией NVIDIA[1].

Далее загрузим модель с помощью функции `clip.load`. Эта функция принимает два аргумента:

- название модели для загрузки;
- устройство для запуска модели.

Нам нужна модель под названием ViT-B/32.

```
model, preprocess = clip.load('ViT-B/32', device=device)
```

Далее мы загружаем изображение, выполняем предварительную обработку и кодируем его с помощью модели CLIP:

```
# Загрузка изображения
image = PIL.Image.open("../resources/ASTRONAUTS.jpg")

# Предварительная обработка изображения
image_input = preprocess(image).unsqueeze(0).to(device)

# Кодирование изображения с помощью CLIP
with torch.no_grad():
   image_features = model.encode_image(image_input)
```

Функция `preprocess` применяет набор стандартных преобразований к входному изображению (изменение размера, нормализация и т. д.), чтобы подготовить его для ввода в модель CLIP.

Изображение после предварительной обработки преобразуется в тензор PyTorch, распаковывается по первому измерению для создания пакетного измерения и перемещается на устройство (CPU или GPU), указанное в переменной `device`.

[1] **https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/.**

Чтобы передать изображение в модель CLIP, нам нужно преобразовать его в тензор. Функция `preprocess` возвращает тензор, который уже имеет нужный формат для ввода в модель, но обрабатывает только одно изображение за раз. Модель ожидает пакет изображений в качестве входных данных, даже если размер пакета равен единице. Чтобы создать пакет единичного размера, нам нужно добавить к тензору дополнительное измерение вдоль первой оси (также известной как ось пакета). Именно это делает метод `unsqueeze(0)`: он добавляет к тензору новое измерение с индексом 0.

Если очень коротко, *тензор* – это многомерный массив чисел, а тензор PyTorch – это особый тип тензора, используемый в глубоком обучении. В нашем контексте входное изображение представляет собой двумерный массив пиксельных значений.

Блок `with torch.no_grad()` используется, чтобы гарантировать, что градиенты не вычисляются, что может повысить производительность и сократить использование памяти.

В машинном обучении градиент – это математическая функция, которая помогает оптимизировать нейронную сеть. PyTorch использует градиенты для обучения моделей. Однако сейчас нам не нужно вычислять градиенты, потому что мы не обучаем модель. Итак, мы используем блок `with torch.no_grad()` для временного отключения градиента; именно поэтому производительность вычислений улучшается.

Когда PyTorch отслеживает граф вычислений и сохраняет промежуточные результаты, ему требуется дополнительная память и вычислительные ресурсы. Используя `with torch.no_grad()`, мы можем уменьшить объем памяти и ускорить вычисления, поскольку PyTorch не нужно сохранять промежуточные результаты или вычислять градиенты во время этой операции. Это может быть особенно важно при кодировании большого количества текстовых запросов, так как это может повысить производительность и сократить использование памяти.

Далее определим список запросов:

```
# Определение списка запросов
prompts = [
    "Большая галактика в центре скопления галактик, расположенного в созвездии
    Волопаса.",
    "Автобус MTA Long Island только что выехал из автовокзала Хемпстеда по
    маршруту N6.",
    "Участники экспедиции STS-86 Владимир Титов и Жан-Лу Кретьен позируют для
    фото в главном модуле станции",
    "Вид на Международную космическую станцию (МКС) с космического корабля
    Союз ТМА-19 при приближении к станции для стыковки. ",
    "Цирковой тигр в клетке на фоне дрессировщика тигров.",
    "Автомеханик занимается ремонтом двигателя",
]
```

Кодируем запросы:

```
# Кодируем текстовые запросы с помощью модели CLIP
with torch.no_grad():
   text_features = model.encode_text(clip.tokenize(prompts).to(device))
```

Вычисляем сходство между изображением и каждым запросом:

```
# Вычисляем сходство между изображением и каждым запросом
similarity_scores = (100.0 * image_features @ text_features.T).softmax(dim=-1)
```

Наконец, выводим на печать запрос с наивысшей оценкой сходства:

```
# Выводим на печать запрос с наивысшей оценкой сходства
most_similar_prompt_index = similarity_scores.argmax().item()
most_similar_prompt = prompts[most_similar_prompt_index]
print("Изображение больше всего соответствует запросу: {}".
format(most_similar_prompt))
```

Так выглядит полный код:

```
import torch
import clip
import PIL

# Загрузка модели CLIP
device= "cuda" if torch.cuda.is_available() else "cpu"
   model, preprocess = clip.load('ViT-B/32', device=device)

# Загрузка изображения
image = PIL.Image.open("../resources/ASTRONAUTS.jpg")

# Предварительная обработка изображения
image_input = preprocess(image).unsqueeze(0).to(device)

# Кодирование изображения с помощью CLIP
with torch.no_grad():
        image_features = model.encode_image(image_input)

# Определение списка запросов
prompts = [
   "Большая галактика в центре скопления галактик, расположенного в созвездии
   Волопаса.",
   "Автобус MTA Long Island только что выехал из автовокзала Хемпстеда по
   маршруту N6.",
   "Специалисты экспедиции STS-86 Владимир Титов и Жан-Лу Кретьен позируют для
   фото в главном модуле станции",
   "Вид на Международную космическую станцию (МКС) с космического корабля
   Союз TMA-19 при приближении к станции для стыковки. ",
   "Цирковой тигр в клетке на фоне дрессировщика тигров.",
   "Автомеханик занимается ремонтом двигателя",
]
```

```
# Кодируем текстовые запросы с помощью модели CLIP
with torch.no_grad():
    text_features = model.encode_text(clip.tokenize(prompts).to(device))

    # Вычисляем сходство между изображением и каждым запросом
    similarity_scores = (100.0 * image_features @ text_features.T).
    softmax(dim=-1)

    # Выводим на печать запрос с наивысшей оценкой сходства
    most_similar_prompt_index = similarity_scores.argmax().item()
    most_similar_prompt = prompts[most_similar_prompt_index]
    print("Изображение больше всего соответствует запросу: {}".
    format(most_similar_prompt))
```

Обратите внимание, что сначала вам нужно установить правильные зависимости, как описано в официальном репозитории Git[1].

```
conda install --yes -c pytorch pytorch=1.7.1 torchvision cudatoolkit=11.0
pip install ftfy regex tqdm
pip install git+https://github.com/openai/CLIP.git
```

16.3. Stable Diffusion наоборот: изображение в текст

Весьма полезным приложением, созданным с использованием CLIP, является CLIP Interrogator.

CLIP Interrogator – это инструмент разработки запросов, который сочетает в себе CLIP OpenAI и BLIP[2] от Salesforce для оптимизации текстовых запросов под определенное изображение. Полученный запрос можно использовать с моделями преобразования текста в изображение, такими как Stable Diffusion в DreamStudio, для создания или воспроизведения изображений.

Использование не составит труда:

```
from PIL import Image
from clip_interrogator import Config, Interrogator
image = Image.open(image_path).convert('RGB')
ci = Interrogator(Config(clip_model_name="ViT-L-14/openai"))
print(ci.interrogate(image))
```

Но учтите, что вам необходимо сначала установить такие пакеты:

```
# Для примера устанавливаем torch с поддержкой GPU:
pip3 install torch torchvision --extra-index-url
https://download.pytorch.org/whl/cu117
```

[1] https://github.com/openai/CLIP.

[2] https://github.com/salesforce/BLIP.

```
# Устанавливаем clip-interrogator
pip install clip-interrogator==0.5.4
```

BLIP – это фреймворк для предварительного обучения моделей, позволяющий им научиться понимать и генерировать как изображения, так и речь. Он достиг передовых результатов во многих задачах, связанных как со зрением, так и с речью.

По данным Salesforce, BLIP обеспечивает высочайшую производительность в семи задачах языка визуального восприятия, это:

- поиск изображения и текста;
- подписи к изображениям;
- визуальный ответ на вопрос;
- визуальное обоснование;
- визуальный диалог;
- поиск текста и видео без обучающих примеров;
- ответы на видеовопросы без обучающих примеров.

Глава 17

Генерация изображений с помощью DALL·E

17.1. Введение

Благодаря современным методам глубокого обучения модель GPT может научиться создавать изображения на основе текстового запроса или существующего изображения (рис. 17.1).

Рис. 17.1. Пример изображения, созданного по запросу «3D-рендер милой тропической рыбки в аквариуме, темно-голубой фон, стиль digital art»

Для создания нового изображения исходное (входное) изображение изменяют с помощью набора алгоритмов. Модель может создавать множество различных изображений, от простых до сложных, в зависимости от того, с чем ей поручено работать.

Одна из мощных функций модели заключается в том, что она может просматривать созданные изображения и улучшать их, делая более подробными и точными. Другими словами, она может учиться на собственных изображениях и со временем становиться все лучше.

В целом API изображений предоставляет три способа взаимодействия с конечной точкой `images`:

- создание изображения с нуля на основе текстовой подсказки;
- внесение изменений в существующее изображение на основе текстового запроса;
- создание различных вариантов существующего исходного изображения.

За последние годы в OpenAI обучили нейронную сеть под названием DALL·E, основанную на GPT-3. DALL·E – это уменьшенная версия GPT-3 с 12 млрд параметров вместо 175 млрд. Она специально разработана для создания изображений из текстовых описаний с использованием набора данных пар текст–изображение вместо очень широкого набора данных, который применяли для обучения GPT-3.

Чтобы увидеть модель в действии, вы можете использовать приложение для предварительного просмотра результатов работы DALL·E[1].

В этой главе вы узнаете, как можно использовать API вместо веб-интерфейса.

В настоящее время ограничение скорости установлено на уровне 50 изображений в минуту, но его можно обойти, следуя инструкциям в статье справочного центра[2].

17.2. Базовый пример генерации изображения по запросу

Начнем с первого примера.[3]

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
         key, value = line.strip().split("=")
```

[1] **https://labs.openai.com/**.

[2] **https://help.openai.com/en/articles/6696591**.

[3] Рекомендуется составлять текстовые запросы на английском языке, поэтому перевод будет дан только в виде комментариев к коду. – *Прим. перев.*

```
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

kwargs = {
            "prompt": "beautiful landscape", #красивый пейзаж
}

im = openai.Image.create(**kwargs)

print(im)
```

По сути, мы выполняем те же действия, что и раньше:

- аутентификацию;
- вызов конечной точки API со списком параметров.

В этом примере мы поместили параметры в словарь `kwargs`, но это ничего не меняет. После выполнения приведенного выше кода вы получите вывод наподобие такого:

```
{
  "created": 1675354429,
  "data": [
    {
      "url": "https://oaidalleapiprodscus.blob.core.windows.net/private/
org-EDUZx9TXM1EWZ6oB5e49duhV/user-FloqMRrL7hkbSSXMojMpIaw1/img-
MvgeSIKm0izdlr32ePzcAr8H.png?st=2023-02-02T15%3A13%3A49Z&se=2023-02-02T17%
3A13%3A49Z&sp=r&sv=2021-08-06&sr=b&rscd=inline&rsct=image/png&skoid=6aaadede-
4fb3-4698-a8f6-684d7786b067&sktid=a48cca56-e6da-484e-a814-9c849652bcb3&skt=
2023-02-01T21%3A21%3A02Z&ske=2023-02-02T21%3A21%3A02Z&sks=b&skv=2021-08-06&sig=
WWyNYn5JHC2u08Hrb4go42azmA8k0daPw2G%2BQV9Tsh8%3D"
    }
  ]
}
```

Щелкните мышкой по этому URL-адресу или вставьте его в адресную строку браузера, чтобы открыть изображение.

17.3. Создание нескольких изображений

По умолчанию API возвращает одно изображение, однако вы можете запросить до 10 изображений за раз, используя параметр `n`. Давайте попробуем получить два изображения:

```
import os
import openai

def init_api():
```

```
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
    openai.organization = os.environ.get("ORG_ID")

init_api()

prompt = "beautiful landscape" #красивый пейзаж
n = 2

kwargs = {
          "prompt": prompt,
          "n": n,
}

im = openai.Image.create(**kwargs)

print(im)
```

Заметим, что можно распечатать URL-адрес изображения напрямую, используя:

```
for i in range(n):
    print(im.data[i].url)
```

17.4. Получение изображений разного размера

Сгенерированные изображения могут иметь три возможных размера:

- 256×256,
- 512×512,
- 1024×1024.

Очевидно, чем меньше размер, тем быстрее будет проходить генерация.

Чтобы генерировать изображения определенного размера, вам необходимо передать параметр размера в API:

```
import os
import openai

def init_api():
    with open(".env") as env:
        for line in env:
            key, value = line.strip().split("=")
            os.environ[key] = value

    openai.api_key = os.environ.get("API_KEY")
```

```
    openai.organization = os.environ.get("ORG_ID")

init_api()

prompt = "beautiful landscape" #красивый пейзаж
n = 1
size = "256x256"

kwargs = {
          "prompt": prompt,
          "n": n,
          "size": size,
}

im = openai.Image.create(**kwargs)

for i in range(n):
   print(im.data[i].url)
```

17.5. Улучшенные запросы на создание изображений

Разработка запросов (prompt engineering) – это способ использовать искусственный интеллект для понимания естественного языка. Он работает, превращая задачи в запросы и обучая языковую модель распознавать такие запросы. Языковая модель может быть большой, уже обученной моделью, и нужно лишь научиться делать правильные запросы.

В 2022 г. три платформы искусственного интеллекта – DALL·E, Stable Diffusion и Midjourney – стали доступны для всех желающих. Эти платформы получают текстовые запросы в качестве входных данных и создают изображения, что открыло новую грань в разработке запросов, которая фокусируется на превращении слов в изображения.

Используя DALL·E, вы можете настроить запросы в соответствии со своими вкусами и потребностями. От света, художественного стиля и эмоций до позиций и ракурсов – существует бесконечное количество возможностей.

Мы рассмотрим некоторые из них, но я предлагаю вам самостоятельно изучить множество других ресурсов, доступных в интернете и связанных с этой темой, включая бесплатную книгу DALL·E 2 Prompt Book[1].

Некоторые из запросов в этой книге составлены на основе запросов из DALL·E 2 Prompt Book, остальные были вдохновлены источниками, включая мою собственную работу.

17.5.1. *Подражание художникам*

Между этими двумя подсказками есть разница в сгенерированном изображении:

[1] **https://dallery.gallery/the-dalle-2-prompt-book/**.

```
prompt = "beautiful landscape" #красивый пейзаж
   prompt = "beautiful landscape by Van Gogh" #красивый пейзаж в стиле Ван Гога
```

Очевидно, что второе изображение будет больше похоже на картину Ван Гога.

Сгенерированное изображение может быть картиной Ван Гога, изображением в мультяшном стиле или абстрактным произведением искусства. Обратите внимание, что вы можете просто использовать ключевые слова, разделенные запятыми:

```
prompt = "beautiful landscape, Van Gogh" #красивый пейзаж, Ван Гог
```

Мы можем, например, использовать модернистский стиль живописи Агнес Лоуренс Пелтон.

```
prompt = "beautiful landscape by Agnes Lawrence Pelton" #красивый пейзаж
в стиле Агнес Лоуренс Пелтон
```

Или в стиле Масаси Кисимото, автора манги Наруто:

```
prompt = "A teenager by Masashi Kishimoto" #подросток в стиле Масаси Кисимото
```

Таким образом, вы можете настроить стиль сгенерированного изображения в соответствии с вашими потребностями.

Также можно использовать другие ключевые слова, относящиеся к желаемому стилю, не обязательно к исполнителю:

```
prompt = "A teenager. Naruto manga style" # Подросток. Стиль манги Наруто
```

Вот список некоторых популярных художников и фотографов, стилем которых владеет модель:

```
Mohammed Amin
Dorothea Lange
Yousuf Karsh
Helmut Newton
Diane Arbus
Eric Lafforgue
Annie Leibovitz
Lee Jeffries
Steve McCurry
Dmitry Ageev
Rosie Matheson
Nancy Goldin
David Lachapelle
Peter Lindbergh
Robert Mapplethorpe
David Bailey
Terry Richardson
Martin Schoeller
Julia Margaret Cameron
```

```
George Hurrell
Ansel Adams
Dorothea Lange
Edward Weston
Elliott Erwitt
Henri Cartier-Bresson
Robert Capa
W. Eugene Smith
Garry Winogrand
Diane Arbus
Robert Frank
Walker Evans
Robert Mapplethorpe
Pablo Picasso
Vincent Van Gogh
Claude Monet
Edvard Munch
Salvador Dali
Edgar Degas
Paul Cezanne
Rene Magritte
Sonia Delaunay
Zeng Fanzhi
Vitto Ngai
Yoji Shinkawa
J.M.W. Turner
Gerald Brom
Jack Kirby
Pre-Raphaelite
Alphonse Mucha
Caspar David Friedrich
William Blake
William Morris
Albrecht Durer
Raphael Sanzio
Michelangelo Buonarroti
Leonardo Da Vinci
Rene Magritte
```

17.5.2. *Имитация художественных стилей*

Чтобы имитировать другие художественные стили, вы также можете использовать ключевые слова, содержащие название художественных стилей и направлений, такие как модерн, импрессионизм, абстрактный экспрессионизм, орфизм, неоклассицизм и многие другие!

```
Art Nouveau
Impressionism
Abstract Expressionism
Orphism
Neoclassicism
```

```
Cubism
Fauvism
Surrealism
Expressionism
Dadaism
Pop Art
Minimalism
Postmodernism
Futurism
Art Deco
Early Renaissance
Religious Art
Chinese Art
Baroque
```

Попробуйте также использовать ключевые слова из приведенного ниже списка, которые объясняют различные художественные стили:

```
3D sculpture
Comic book
Sketch drawing
Old photograph
Modern photograph
Portrait
Risograph
Oil painting
Graffiti
Watercolor
Cyberpunk
Synthwave
Gouache
Pencil drawing (detailed, hyper-detailed, very realistic)
Pastel drawing
Ink drawing
Vector
Pixel art
Video game
Anime
Manga
Cartoon
Illustration
Poster
Typography
Logo
Branding
Etching
Woodcut
Political cartoon
Newspaper
Coloring sheet
Field journal line art
```

```
Street art
Airbrush
Crayon
Child's drawing
Acrylic on canvas
Pencil drawing (colored, detailed)
Ukiyo-e
Chinese watercolor
Pastels
Corporate Memphis design
Collage (photo, magazine)
Watercolor & pen
Screen printing
Low poly
Layered paper
Sticker illustration
Storybook
Blueprint
Patent drawing
Architectural drawing
Botanical illustration
Cutaway
Mythological map
Voynich manuscript
IKEA manual
Scientific diagram
Instruction manual
Voroni diagram
Isometric 3D
Fabric pattern
Tattoo
Scratch art
Mandala
Mosaic
Black velvet (Edgar Leeteg)
Character reference sheet
Vintage Disney
Pixar
1970s grainy vintage illustration
Studio Ghibli
1980s cartoon
1960s cartoon
```

17.5.3. *Атмосфера, чувства, эмоции*

Используя дополнительные ключевые слова, описывающие атмосферу и общее настроение, вы можете создавать более выразительные изображения. Например, вы можете создать спокойную атмосферу с таким запросом:

```
prompt = "beautiful landscape with a peaceful atmosphere" # красивый пейзаж
с атмосферой спокойствия
```

Или загадочную атмосферу, добавив:

```
prompt = "beautiful landscape with a mysterious atmosphere" # красивый пейзаж
с загадочной атмосферой
```

Точно так же вы можете вызывать чувство радости, печали, надежды или страха. Например, чтобы вызвать изображением чувство надежды, отправьте такой запрос:

```
prompt = "beautiful landscape with a feeling of hope" # красивый пейзаж,
вызывающий чувство надежды
```

Или чувство страха:

```
prompt = "beautiful landscape with a feeling of fear" # красивый пейзаж,
вызывающий чувство страха
```

Вот некоторые случайные ключевые слова, которые вы можете попробовать:

```
light
peaceful
calm
serene
tranquil
soothing
relaxed
placid
comforting
cosy
tranquil
quiet
pastel
delicate
graceful
subtle
balmy
mild
ethereal
elegant
tender
soft
light
muted
bleak
funereal
somber
melancholic
mournful
gloomy
dismal
sad
pale
```

```
washed-out
desaturated
grey
subdued
dull
dreary
depressing
weary
tired
dark
ominous
threatening
haunting
forbidding
gloomy
stormy
doom
apocalyptic
sinister
shadowy
ghostly
unnerving
harrowing
dreadful
frightful
shocking
terror
hideous
ghastly
terrifying
bright
vibrant
dynamic
spirited
vivid
lively
energetic
colorful
joyful
romantic
expressive
bright
rich
kaleidoscopic
psychedelic
saturated
ecstatic
brash
exciting
passionate
hot
```

Вы можете имитировать атмосферу фильмов с узнаваемым художественным стилем, например:

```
from Dancer in the Dark movie
from Howl's Moving Castle movie
from Coraline movie
from Hanna movie
from Inception movie
from Thor movie
from The Lion King movie
from Rosemary's Baby movie
from Ocean's Eleven movie
from Lovely to Look At movie
from Eve's Bayou movie
from Tommy movie
from Chocolat movie
from The Godfather movie
from Kill Bill movie
from The Lord of the Rings movie
from Legend movie
from The Abominable Dr. Phibes movie
from The Shining movie
from Pan's Labyrinth movie
from Blade Runner movie
from Lady in the Water movie
from The Wizard of Oz movie
```

17.5.4. *Цвета*

Вы также можете указать предпочтительные цвета для сгенерированных изображений. Например, если вы хотите получить красное небо, можете добавить в запрос ключевое слово «красный»:

```
prompt = "beautiful landscape with a red sky" # красивый пейзаж с красным небом
```

Вы, конечно, можете использовать другие цвета:

```
Blue
Red
Green
Yellow
Purple
Pink
Orange
Black
White
Gray
Red and Green
Yellow and Purple
Orange and Blue
Black and White
```

```
Pink and Teal
Brown and Lime
Maroon and Violet
Silver and Crimson
Beige and Fuchsia
Gold and Azure
Cyan and Magenta
Lime and Maroon and Violet
Crimson and Silver and Gold
Azure and Beige and Fuchsia
Magenta and Cyan and Teal
Pink and Teal and Lime
Yellow and Purple and Maroon
Orange and Blue and Violet
Black and White and Silver
Fade to Black
Fade to White
Fade to Gray
Fade to Red
Fade to Green
Fade to Blue
Fade to Yellow
Fade to Purple
Fade to Pink
Fade to Orange
Gradient of Red and Green
Gradient of Yellow and Purple
Gradient of Orange and Blue
Gradient of Black and White
Gradient of Pink and Teal
Gradient of Brown and Lime
Gradient of Maroon and Violet
Gradient of Silver and Crimson
Gradient of Beige and Fuchsia
Gradient of Gold and Azure
Gradient of Cyan and Magenta
```

Можно использовать такие запросы, как `6-bit color` (6-битный цвет), `8-bit color` (8-битный цвет), `black and white` (черно-белый) и `pixelated colors` (пиксельные цвета) и т. д.

```
prompt = "beautiful landscape with 6-bit color" #красивый пейзаж в 6-битном
цвете
```

17.5.5. *Разрешение*

Можно попробовать запросить изображения с разными разрешениями:

```
2 bit colors
4 bit colors
8 bit colors
```

```
16 bit colors
24 bit colors
4k resolution
HDR
8K resolution
a million colors
a billion colors
```

17.5.6. *Углы и положения*

Можете настроить вид сцены. Например, если вы хотите смотреть на пейзаж сверху вниз:

```
prompt = "beautiful landscape from above" # красивый пейзаж вид сверху
```

или вид на пейзаж от первого лица:

```
prompt = "beautiful landscape from a first person view" # красивый пейзаж вид
от первого лица
```

или широкоугольный вид пейзажа:

```
prompt = "beautiful landscape with a wide-angle view" # красивый пейзаж с
широкоугольным обзором
```

Вы можете попробовать получить дополнительные эффекты, используя некоторые из этих ключевых слов:

```
Extreme close-up
close-up
medium shot
long shot
extreme long shot
high angle
overhead view
aerial view
tilted frame
dutch angle
over-the-shoulder shot
drone view
panning shot
tracking shot
dolly shot
zoom shot
handheld shot
crane shot
low angle
reverse angle
point-of-view shot
split screen
freeze frame
flashback
```

```
flash forward
jump cut
fade in
fade out
```

17.5.7. *Типы объективов*

Вы можете использовать ключевые слова, описывающие различные типы объективов или приемы работы с камерой. Например, использовать ключевое слово `knolling` (ноллинг[1]), чтобы получить чистую и организованную сцену:

```
prompt = "beautiful landscape knolling" # красивый пейзаж ноллинг
```

или `telephoto lens` (телеобъектив), чтобы получить увеличенное изображение:

```
prompt = "beautiful landscape with a telephoto lens" # красивый пейзаж через
телеобъектив
```

или `fisheye lens` (рыбий глаз), чтобы получить чрезвычайно широкоугольный вид:

```
prompt = "beautiful landscape with a fisheye lens" # красивый пейзаж через
объектив рыбий глаз
```

или `tilt-shift lens` (объектив с наклоном и сдвигом оптической оси), чтобы получить уменьшенное изображение:

```
prompt = "beautiful landscape with a tilt-shift lens" # красивый пейзаж с
управлением перспективой
```

или `360 panorama` (панорама 360°), чтобы получить круговой обзор:

```
prompt = "beautiful landscape with a 360 panorama" # красивый пейзаж с
круговым обзором
```

Вы также можете использовать `drone view` (вид с дрона), чтобы получить вид с воздуха:

```
prompt = "beautiful landscape from a drone view" # красивый пейзаж вид с дрона
```

или `satellite imagery` (спутниковые снимки), чтобы получить вид со спутника:

```
prompt = "beautiful landscape from satellite imagery" # красивый пейзаж снимок
со спутника
```

Вот другие ключевые слова, которые вы можете попробовать:

```
high-resolution microscopy
microscopy
macro lens
```

[1] Метод организации пространства путем группировки схожих предметов и их выравнивания параллельно или под углом в 90°.

```
pinhole lens
knolling
first person view
wide angle lens
lens distortion
ultra-wide angle lens
fisheye lens
telephoto lens
panorama
360 panorama
tilt-shift lens
telescope lens
lens flare
```

Если вы знакомы с фотографией и различными настройками фотооборудования, вы также можете их использовать. Вот несколько случайных примеров:

```
Aperture: f/5.6, Shutter Speed: 1/250s, ISO: 400, Landscape photography, high-
quality DSLR
Aperture: f/8, Shutter Speed: 1/60s, ISO: 800, Street photography, low light
conditions
Aperture: f/11, Shutter Speed: 1/1000s, ISO: 1600, Sports photography, fast
shutter speed
Aperture: f/16, Shutter Speed: 2s, ISO: 100, Night photography, long exposure
Aperture: f/2.8, Shutter Speed: 1/500s, ISO: 1600, Wildlife photography, high
sensitivity, high ISO
Aperture: f/4, Shutter Speed: 1/60s, ISO: 100, Portrait photography, shallow
depth of field
Aperture: f/5.6, Shutter Speed: 1/60s, ISO: 100, Macro photography, close-up
shots
Aperture: f/8, Shutter Speed: 1/15s, ISO: 100, Fine art photography, Kodak
Gold 200 14 film color, 35mm
Aperture: f/11, Shutter Speed: 4s, ISO: 200, Architectural photography, slow
shutter speed, long exposure.
```

17.5.8. *Осветительные приборы*

Вы можете использовать ключевые слова, чтобы лучше управлять светом, что является важным аспектом в фотографии и искусстве в целом. Вот несколько примеров:

```
Warm lighting
Side lighting
High-key lighting
fluorescent lighting
Harsh flash lighting
Low-key lighting
Flat lighting
Even lighting
Ambient lighting
```

```
Colorful lighting
Soft light
Hard light
Diffused light
Direct light
Indirect light
Studio lighting
Red and green lighting
Flash photography
Natural lighting
Backlighting
Edge lighting
Cold
Backlit
Glow
Neutral white
High-contrast
Lamp light
Fireworks
2700K light
4800K light
6500K light
```

17.5.9. *Типы пленок и фильтры*

Вы также можете указать тип «фотопленки», под которую надо стилизовать изображение. Например, для эффекта cyanotype (цианотипия) можно добавить ключевое слово:

```
prompt = "beautiful landscape with a cyanotype look" # красивый пейзаж с
эффектом цианотипии
```

Для эффекта «черно-белой фотопленки» вы можете использовать такой запрос:

```
prompt = "beautiful landscape with a black and white look" # красивый пейзаж с
эффектом черно-белой фотопленки
```

А для эффекта пленки «Kodak tri-X 400TX» вы можете использовать следующий запрос:

```
prompt = "beautiful landscape with a tri-X 400TX look" # красивый пейзаж с
эффектом пленки tri-X 400TX
```

Попробуйте больше примеров, таких как `solarized` (соларизованный), `sepia` (сепия) и `monochrome` (монохромный).

```
Kodachrome
Autochrome
Lomography
Polaroid
Instax
```

```
Cameraphone
CCTV
Disposable camera
Daguerrotype
Camera obscura
Double exposure
Cyanotype
Black and white
Tri-X 400TX
Infrared photography
Bleach bypass
Contact sheet
Colour splash
Solarised
Anaglyph
```

Вы также можете попробовать добавить названия фильтров Instagram:

```
Instagram Clarendon filter
Instagram Juno filter
Instagram Ludwig filter
Instagram Lark filter
Instagram Gingham filter
Instagram Lo-fi filter
Instagram X-Pro II filter
Instagram Aden filter
Instagram Perpetua filter
Instagram Reyes filter
Instagram Slumber filter
```

17.6. Создание генератора случайных изображений

Вариантов запроса на самом деле намного больше, чем мы упоминали, и их достаточно, чтобы построить генератор случайных изображений с использованием Python.

Мы создадим несколько списков, каждый из которых будет включать в себя разные наборы ключевых атрибутов. Программа может случайным образом выбрать атрибуты из каждого списка и объединить их с запросом пользователя для создания окончательного запроса.

Вот как устроена основная часть кода:

```
# Определяем списки
list 1 = [keywords 11, keywords 12,..]
list 2 = [keywords 21, keywords 22,..]
list 3 = [keywords 31, keywords 32,..]
list 4 = [keywords 41, keywords 42,..]
..etc

# Читаем запрос пользователя
user prompt = "lorem ipsum"
```

```
# Генерируем запрос со случайными ключевыми атрибутами
some generated prompts =
"user prompt" + "keywords 11" + "keywords 21" + "keywords 31" + "keywords 42"
"user prompt" + "keywords 12" + "keywords 22" + "keywords 32" + "keywords 41"
"user prompt" + "keywords 21"
"user prompt" + "keywords 32"
"user prompt" + "keywords 21" + "keywords 41"
"user prompt" + "keywords 22" + "keywords 42"
"user prompt" + "keywords 31" + "keywords 41"
```

Давайте взглянем, как работает весь код:

```
import os
import openai
import random
import time

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

artists = ["Mohammed Amin","Dorothea Lange","Yousuf Karsh ","Helmut
Newton","Diane Arbus","Eric Lafforgue","Annie Leibovitz","Lee Jeffries",
"Steve McCurry","Dmitry Ageev","Rosie Matheson","Nancy Goldin","David
Lachapelle","Peter Lindbergh","Robert Mapplethorpe","David Bailey","Terry
Richardson","Martin Schoeller","Julia Margaret Cameron","George
Hurrell","Ansel Adams","Dorothea Lange","Edward Weston","Elliott
Erwitt","Henri Cartier-Bresson","Robert Capa","W. Eugene Smith","Garry
Winogrand","Diane Arbus","Robert Frank","Walker Evans","Robert
Mapplethorpe","Pablo Picasso","Vincent Van Gogh","Claude Monet","Edvard
Munch","Salvador Dali","Edgar Degas","Paul Cezanne","Rene Magritte","Sonia
Delaunay","Zeng Fanzhi","Vitto Ngai","Yoji Shinkawa","J.M.W. Turner","Gerald
Brom","Jack Kirby","Pre-Raphaelite","Alphonse Mucha","Caspar David
Friedrich","William Blake","William Morris","Albrecht Durer","Raphael
Sanzio","Michelangelo Buonarroti","Leonardo Da Vinci","Rene Magritte",]

art_styles = ["Art Nouveau", "Impressionism", "Abstract Expressionism",
"Orphism", "Neoclassicism", "Cubism", "Fauvism", "Surrealism",
"Expressionism", "Dadaism", "Pop Art", "Minimalism", "Postmodernism",
"Futurism", "Art Deco", "Early Renaissance", "Religious Art", "Chinese
Art", "Baroque", "Art Nouveau", "Impressionism", "Abstract Expressionism",
"Orphism", "Neoclassicism", "Cubism", "Fauvism", "Surrealism",
"Expressionism", "Dadaism", "Pop Art", "Minimalism", "Postmodernism",
"Futurism", "Art Deco", "Early Renaissance", "Religious Art", "Chinese Art",
```

```
"Baroque", "3D sculpture", "Comic book", "Sketch drawing", "Old photograph",
"Modern photograph", "Portrait", "Risograph", "Oil painting", "Graffiti",
"Watercolor", "Cyberpunk", "Synthwave", "Gouache", "Pencil drawing (detailed,
hyper-detailed, very realistic)", "Pastel drawing", "Ink drawing", "Vector",
"Pixel art", "Video game", "Anime", "Manga", "Cartoon", "Illustration",
"Poster", "Typography", "Logo", "Branding", "Etching", "Woodcut", "Political
cartoon", "Newspaper", "Coloring sheet", "Field journal line art", "Street
art", "Airbrush", "Crayon", "Child's drawing", "Acrylic on canvas", "Pencil
drawing (colored, detailed)", "Ukiyo-e", "Chinese watercolor", "Pastels",
"Corporate Memphis design", "Collage (photo, magazine)", "Watercolor & pen",
"Screen printing", "Low poly", "Layered paper", "Sticker illustration",
"Storybook", "Blueprint", "Patent drawing", "Architectural drawing",
"Botanical illustration", "Cutaway", "Mythological map", "Voynich manuscript",
"IKEA manual", "Scientific diagram", "Instruction manual","Voroni diagram",
"Isometric 3D", "Fabric pattern", "Tattoo", "Scratch art", "Mandala",
"Mosaic", "Black velvet (Edgar Leeteg)", "Character reference sheet", "Vintage
Disney", "Pixar", "1970s grainy vintage illustration", "Studio Ghibli", "1980s
cartoon", "1960s cartoon",]

vibes = ["light", "peaceful", "calm", "serene", "tranquil", "soothing",
"relaxed", "placid", "comforting", "cosy", "tranquil", "quiet", "pastel",
"delicate", "graceful", "subtle", "balmy", "mild", "ethereal", "elegant",
"tender", "soft", "light", "muted", "bleak", "funereal", "somber",
"melancholic", "mournful", "gloomy", "dismal", "sad", "pale", "washed-
out", "desaturated", "grey", "subdued", "dull", "dreary", "depressing",
"weary", "tired", "dark", "ominous", "threatening", "haunting", "forbidding",
"gloomy", "stormy", "doom", "apocalyptic", "sinister", "shadowy", "ghostly",
"unnerving", "harrowing", "dreadful", "frightful", "shocking", "terror",
"hideous", "ghastly", "terrifying", "bright", "vibrant", "dynamic",
"spirited", "vivid", "lively","energetic", "colorful", "joyful", "romantic",
"expressive", "bright", "rich", "kaleidoscopic", "psychedelic", "saturated",
"ecstatic", "brash", "exciting", "passionate", "hot", "from Dancer in the
Dark movie ", "from Howl's Moving Castle movie ", "from Coraline movie ",
"from Hanna movie ", "from Inception movie ", "from Thor movie ", "from The
Lion King movie ", "from Rosemary's Baby movie ", "from Ocean's Eleven movie
", "from Lovely to Look At movie ", "from Eve's Bayou movie ", "from Tommy
movie ", "from Chocolat movie ", "from The Godfather movie ", "from Kill Bill
movie ", "from The Lord of the Rings movie ", "from Legend movie ", "from
The Abominable Dr. Phibes movie ", "from The Shining movie ", "from Pan's
Labyrinth movie ", "from Blade Runner movie ", "from Lady in the Water movie
", "from The Wizard of Oz movie",]

colors = ["Blue", "Red", "Green", "Yellow", "Purple", "Pink", "Orange",
"Black", "White", "Gray", "Red and Green", "Yellow and Purple", "Orange and
Blue", "Black and White", "Pink and Teal", "Brown and Lime", "Maroon and
Violet", "Silver and Crimson", "Beige and Fuchsia", "Gold and Azure", "Cyan
and Magenta", "Lime and Maroon and Violet", "Crimson and Silver and Gold",
"Azure and Beige and Fuchsia", "Magenta and Cyanand Teal", "Pink and Teal and
Lime", "Yellow and Purple and Maroon", "Orange and Blue and Violet", "Black
and White and Silver", "Fade to Black", "Fade to White", "Fade to Gray", "Fade
to Red", "Fade to Green", "Fade to Blue", "Fade to Yellow", "Fade to Purple",
```

```
"Fade to Pink", "Fade to Orange", "Gradient of Red and Green", "Gradient of
Yellow and Purple", "Gradient of Orange and Blue", "Gradient of Black and
White", "Gradient of Pink and Teal", "Gradient of Brown and Lime", "Gradient
of Maroon and Violet", "Gradient of Silver and Crimson", "Gradient of Beige
and Fuchsia", "Gradient of Gold and Azure", "Gradient of Cyan and Magenta",]

resolution = ["2 bit colors", "4 bit colors", "8 bit colors", "16 bit colors",
"24 bit colors", "4k resolution", "HDR", "8K  resolution", "a million colors",
"a billion colors",]

angles = ["Extreme close-up","close-up","medium shot","long shot","extreme
long shot","high angle","overhead view","aerial view","tilted frame","dutch
angle","over-the-shoulder shot","drone view","panning shot","tracking
shot","dolly shot","zoom shot", "handheld shot","crane shot","low
angle","reverse angle","point-of-view shot","split screen","freeze
frame","flashback","flash forward","jump cut","fade in","fade out",]

lens = ["high-resolution microscopy", "microscopy", "macro lens", "pinhole
lens", "knolling", "first person view", "wide angle lens", "lens distortion",
"ultra-wide angle lens", "fisheye lens", "telephoto lens", "panorama", "360
panorama", "tilt-shift lens", "telescope lens", "lens flare", "Aperture:
f/5.6, Shutter Speed: 1/250s, ISO: 400,  Landscape photography, high-quality
DSLR", "Aperture: f/8, Shutter Speed: 1/60s, ISO: 800, Street photography,
low light conditions", "Aperture: f/11, Shutter Speed: 1/1000s, ISO: 1600,
Sports photography, fast shutter speed", "Aperture: f/16, Shutter Speed:
2s, ISO: 100, Night photography, long exposure", "Aperture: f/2.8, Shutter
Speed: 1/500s, ISO: 1600, Wildlife photography, high sensitivity, high ISO",
"Aperture: f/4, Shutter Speed: 1/60s, ISO: 100, Portrait photography, shallow
depth of field", "Aperture: f/5.6, Shutter Speed: 1/60s, ISO: 100, Macro
photography, close-up shots", "Aperture: f/8, Shutter Speed: 1/15s, ISO:
100, Fine art photography, Kodak Gold 200 film color, 35mm", "Aperture: f/11,
Shutter Speed: 4s, ISO: 200, Architectural photography, slow shutter speed,
long exposure.",]

light = ["Warm lighting", "Side lighting", "High-key lighting", "fluorescent
lighting", "Harsh flash lighting", "Low-key lighting", "Flat lighting",
"Even lighting", "Ambient lighting", "Colorful lighting", "Soft light",
"Hard light", "Diffused light", "Direct light", "Indirect light", "Studio
lighting", "Red and green lighting", "Flash photography", "Natural lighting",
"Backlighting", "Edge lighting", "Cold", "Backlit", "Glow", "Neutral white",
"High-contrast", "Lamp light", "Fireworks", "2700K light", "4800K light",
"6500K light",]

filter = ["Kodachrome", "Autochrome", "Lomography", "Polaroid", "Instax",
"Cameraphone", "CCTV", "Disposable camera", "Daguerrotype", "Camera obscura",
"Double exposure", "Cyanotype", "Black and white", "Tri-X 400TX", "Infrared
photography", "Bleach bypass", "Contact sheet", "Colour splash", "Solarised",
"Anaglyph", "Instagram Clarendon filter", "Instagram Juno filter", "Instagram
Ludwig filter", "Instagram Lark filter", "Instagram Gingham filter", "Instagram
Lo-fi filter", "Instagram X-Pro II filter", "Instagram Aden filter", "Instagram
```

```
Perpetua filter", "Instagram Reyes filter", "Instagram Slumber filter"]

lists = [
   colors,
   resolution,
   angles,
   lens,
   light,
   filter
]

user_prompts = [
# Счастливый Дарт Вейдер улыбается и машет туристам в музее памятных вещей из
"Звездных войн".
"Happy Darth Vader smiling and waving at tourists in a museum of Star Wars
memorabilia.",
# Дарт Вейдер читает рэп с 2Pac.
"Darth Vader rapping with 2Pac.",
# Дарт Вейдер играет на пианино.
"Darth Vader playing the piano.",
# Дарт Вейдер играет на гитаре.
"Darth Vader playing the guitar.",
# Дарт Вейдер ест суши.
"Darth Vader eating sushi.",
# Дарт Вейдер пьет молоко из стакана.
"Darth Vader drinking a glass of milk.",
]

n = 5

for user_prompt in user_prompts:
   print("Генерация изображения по запросу: " + user_prompt)
   for i in range(n):
      customizations = ""
      for j in range(len(lists)):
         list = lists[j]
         choose_or_not = random.randint(0, 1)
         if choose_or_not == 1:
            customizations += random.choice(list) + ", "

      kwargs = {
          "prompt": user_prompt + ", " + customizations,
          "n": n,
      }
      print("Генерация изображения номер: " + str(i+1) + ".
      Используем запрос: " + user_prompt + ", " + customizations)
      im = openai.Image.create(**kwargs)
      print(im.data[i].url)
      print("\n")
      time.sleep(1)
   print("Завершена генерация изображений по запросу: " + user_prompt)
```

На рис. 17.2 показан пример сгенерированного изображения:

Рис. 17.2. Пример сгенерированного изображения

Глава 18

Редактирование изображений с помощью DALL·E

Используя GPT DALL·E, можно редактировать изображения.

Конечная точка `image edits` позволяет изменять и дополнять изображение, загружая маску.

Маска в виде прозрачного участка точно указывает, в каком месте должно быть отредактировано изображение. Пользователь должен предоставить запрос с описанием для заполнения отсутствующей части, включая удаленную часть.

Пример, приведенный в документации OpenAI (рис. 18.1), не требует пояснений.

Рис. 18.1. Пример добавления элемента изображения в заданном месте

18.1. Пример редактирования изображения

Давайте попробуем отредактировать изображение.

Нам понадобятся два изображения, удовлетворяющие определенным требованиям:

- исходное изображение;
- маска.

Оба изображения должны быть в формате PNG и иметь одинаковый размер. OpenAI ограничивает изображения размером в 4 Мб.

Изображения также должны быть квадратными.

Мы собираемся использовать изображение, показанное на рис. 18.2.

Рис. 18.2. Исходное изображение для редактирования

С помощью графического редактора удалите некоторые части изображения и сохраните его в файл. Убедитесь, что изображение имеет слой прозрачности. Удаляемая часть должна быть прозрачной.

Изображение, которое мы будем использовать, показано на рис. 18.3.

Рис. 18.3. Изображение с маской

Я назвал первое изображение `without_mask.png`, а второе изображение – `mask.png` и сохранил их в одну папку `resources`.

Нам нужно, чтобы модель отредактировала изображение с помощью маски и добавила к нему группу людей, идущих пешком. Вот как это делается:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
```

```
    openai.organization = os.environ.get("ORG_ID")

init_api()

image = open("../resources/without_mask.png", "rb")
mask = open("../resources/mask.png", "rb")
# Группа людей гуляет по зеленому лесу между деревьями
prompt = "A group of people hiking in green forest between trees"
n = 1
size = "1024x1024"

kwargs = {
    "image": image,
    "mask": mask,
    "prompt": prompt,
    "n": n,
    "size": size,
}

response = openai.Image.create_edit(**kwargs)
image_url = response['data'][0]['url']

print(image_url)
```

Давайте разберемся, что делает код:

- открывает два файла изображений, расположенных в каталоге `../resources/`, с именами `without_mask.png` и `mask.png`. Режим `"rb"` используется для открытия файлов в двоичном режиме, что требуется при чтении файлов изображений;
- переменной `prompt` присваивается строковое значение, описывающее содержимое изображения;
- переменной `n` присваивается целочисленное значение `1`, указывающее количество изображений, которые необходимо сгенерировать;
- переменной размера присваивается строковое значение `"1024x1024"`, которое указывает размер генерируемых изображений;
- создается словарь `kwargs`, который сопоставляет ключи `"image"`, `"mask"`, `"prompt"`, `"n"` и `"size"` с соответствующими значениями, хранящимися в переменных `image`, `mask`, `prompt`, `n` и `size`. Этот словарь будет использоваться в качестве аргументов ключевого слова при вызове функции `openai.Image.create_edit`;
- затем вызывается функция `openai.Image.create_edit` с аргументами, предоставленными в словаре `kwargs`. Ответ от функции сохраняется в переменной `response`;
- наконец, URL сгенерированного изображения извлекается из словаря `response` и сохраняется в переменной `image_url`.

Вот и все, что нужно сделать для получения отредактированного изображения.

API возвращает URL-адрес, однако мы можем получить его в формате Base64, изменив `response_format`:

```
kwargs = {
   "image": image,
   "mask": mask,
   "prompt": prompt,
   "n": n,
   "size": size,
   "response_format": "b64_json",
}
```

Глава 19

Черпаем вдохновение из других изображений

OpenAI позволяет создавать различные версии одного и того же изображения. Вы можете взять изображение и на его основе создать новое. Конечно, во многих подобных случаях возникает очень сложная проблема с авторскими правами; однако это отдельная тема, которую мы здесь не обсуждаем. Кроме того, поскольку мы изучаем и используем изображения исключительно в личных и образовательных целях, нас не интересуют эти вопросы.

19.1. Как создать вариацию имеющегося изображения

Процедура проста и понятна. Мы будем использовать модуль `Image` для вызова модели и запроса нового изображения из конечной точки API `variations`.

Мы начнем с исходного изображения, показанного на рис. 19.1.

Вот как это работает:

```
import os
import openai

def init_api():
   with open(".env") as env:
      for line in env:
          key, value = line.strip().split("=")
          os.environ[key] = value

   openai.api_key = os.environ.get("API_KEY")
   openai.organization = os.environ.get("ORG_ID")

init_api()

image = open("../resources/original_image.png", "rb")
n = 3
size = "1024x1024"
```

```
kwargs = {
    "image": image,
    "n": n,
    "size": size
}

response = openai.Image.create_variation(**kwargs)
url = response
```

Рис. 19.1. Исходное изображение

- исходное изображение находится по адресу `"../resources/original_image.png"` и открывается в двоичном режиме с помощью `open("../resources/original_image.png", "rb")`;
- переменной `n` присвоено значение 3, т. е. API сгенерирует три варианта изображения;

- переменной `size` присвоено значение `"1024x1024"`, которое определяет размер выходных изображений;
- аргументы хранятся в виде пар ключ–значение в словаре `kwargs`, который затем передается как аргументы ключевого слова в метод `openai.Image.create_variation()` с использованием синтаксиса `**kwargs` (как мы видели в предыдущих примерах, это позволяет методу получать аргументы в виде отдельных ключевых слов, а не словаря);
- ответ API сохраняется в переменной `response`, а URL-адрес сгенерированного изображения хранится в переменной `url`, которой присваивается значение `response`.

Вы должны получить примерно такой вывод:

```
{
  "created": 1675440279,
  "data": [
    {
     "url": "https://<Ссылка на хранилище Azure>"
    },
    {
     "url": "https://<Ссылка на хранилище Azure >"
    },
    {
     "url": "https://<Ссылка на хранилище Azure >"
    }
  ]
}
```

В код можно внести небольшие изменения, чтобы выводить на печать только URL-адреса:

```
image = open("../resources/original_image.png", "rb")
n = 3
size = "1024x1024"

kwargs = {
      "image": image,
      "n": n,
      "size": size
}

response = openai.Image.create_variation(**kwargs)
urls = response["data"]

for i in range(n):
   print(urls[i]['url'])
```

Так выглядит пример сгенерированного изображения:

Рис. 19.2. Пример сгенерированного изображения

19. 2. Примеры использования вариативных изображений

Один из полезных вариантов использования этой функции – когда вы создаете изображения с помощью запросов и находите правильные комбинации ключевых слов, чтобы получить желаемое изображение, но при этом хотите его улучшить, выбрав наилучший вариант.

По сравнению с другими инструментами создания изображений и преобразования текста в изображение с помощью ИИ, DALL·E обладает несколько ограниченными возможностями, поэтому иногда вам придется потратить некоторое время на поиск правильных запросов. Когда вы найдете удачное сгенерированное изображение, вы сможете улучшить его, получив несколько вариантов.

Эту функцию также можно использовать при редактировании изображения, хотя результаты не всегда вас устроят. В этом случае есть шанс, что вы получите лучшие результаты, используя различные варианты.

Короче говоря, независимо от того, создаете ли вы варианты сгенерированных или отредактированных изображений, эту функцию можно объединить с другими функциями, чтобы выбрать наилучшие результаты среди большого количества вариантов.

Глава 20

Что дальше?

Спасибо, что прочитали это руководство по использованию API OpenAI в программах на языке Python. Я убежден, что благодаря своим непревзойденным возможностям моделирования с использованием естественного языка GPT может полностью изменить способы нашего взаимодействия с искусственным интеллектом в приложениях.

Я уверен, что GPT станет бесценным дополнением к вашему набору инструментов ИИ.

Если вы хотите быть в курсе последних тенденций в экосистемах Python и ИИ, присоединяйтесь к моему сообществу разработчиков по адресу **www.faun.dev/join**. Я рассылаю еженедельные информационные бюллетени с актуальными и полезными учебными пособиями, новостями и идеями от экспертов в сообществе разработчиков программного обеспечения.

Предметный указатель

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com.

Аймен Эль Амри

GPT-3: программирование на Python в примерах

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Яценков В.*
Корректор *Абросимова Л. А.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 $^{1}/_{16}$.
Печать цифровая. Усл. печ. л. 17,71.
Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

OpenAI предоставляет API для доступа к моделям искусственного интеллекта (ИИ). Назначение API – абстрагировать базовые модели путем создания универсального интерфейса для всех версий, позволяющего пользователям использовать GPT независимо от его версии.

Цель книги – предоставить пошаговое руководство по использованию GPT-3 в ваших проектах с помощью API OpenAI. Также рассмотрены и другие модели, такие как CLIP, DALL-E и Whispers.

Независимо от того, создаете ли вы чат-бот, ИИ-ассистента или веб-приложение, предоставляющее данные, сгенерированные ИИ, данная книга поможет реализовать ваши идеи.

Вы создадите такие приложения:

- медицинский бот-помощник для получения справок о лекарствах;
- интеллектуальная система рекомендаций лучшего сорта кофе;
- диалоговая система с памятью и пониманием контекста;
- голосовой помощник с искусственным интеллектом;
- чат-бот, помогающий разобраться в командах Linux;
- семантическая поисковая система;
- система предсказания категорий новостей;
- умная система распознавания изображений;
- генератор рисунков.

Вам не нужно быть специалистом по данным или инженером по машинному обучению, чтобы использовать код на языке Python. Он был разработан таким образом, чтобы не вызвать затруднений у программиста любого уровня.

Аймен Эль Амри – инженер-программист-эрудит, предприниматель и автор с большим опытом работы в различных областях технологий, включая DevOps и Cloud Native, облачную архитектуру, Python, NLP и многое другое. Он подготовил сотни инженеров-программистов и стал автором множества книг и курсов, которые прочитали тысячи разработчиков. Аймен является основателем нескольких проектов, включая FAUN и Marketto.

ISBN 978-5-93700-221-1

9 785937 002211 >