

Аймен Эль Амри

GPT-4

Руководство
по использованию

API Open AI



Аймен Эль Амри

GPT-4. Руководство по использованию API Open AI

OpenAI GPT For Python Developers

The art and science of AI-powered apps with
GPT-4, Whisper, Weaviate, and beyond

Aymen El Amri

GPT-4. Руководство по использованию API Open AI

Аймен Эль Амри



Москва, 2024

УДК 004.4
ББК 32.371
А62

Аймен Эль Амри

А62 GPT-4. Руководство по использованию API Open AI / пер. с англ. В. С. Яценкова. – М.: ДМК Пресс, 2024. – 274 с.: ил.

ISBN 978-5-93700-299-0

В книге рассказывается о том, как использовать генеративные текстовые модели поколений GPT-3.5 и GPT-4 для создания приложений различного назначения, в числе которых интерактивный психотерапевт, интеллектуальный голосовой помощник, система рекомендации товаров, генератор заметок в соцсетях, система распознавания речи и многие другие. Вы научитесь использовать векторные базы данных, узнаете, как управлять уровнем креативности моделей GPT, применять современные методы генерирования высококачественного текста, и даже организуете диалог между двумя чат-ботами. Примеры и практические упражнения помогут закрепить пройденный материал.

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

УДК 004.4
ББК 32.371

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Оглавление

https://t.me/it_books/2

Предисловие от издательства	9
Предисловие.....	10
Об авторе.....	11
История OpenAI и ChatGPT.....	12
Об этой книге.....	15
Оставайтесь на связи.....	16
Как работает GPT?	17
Подготовка среды разработки	20
Важные примечания	20
Установка Python, pip и виртуальной среды для разработки.....	21
Получение ключа API OpenAI.....	22
Установка официальных средств интеграции Python	23
Тестирование ключей API.....	23
Доступные модели и выбор оптимального варианта	26
Модели OpenAI и важные соглашения.....	26
Какую модель лучше использовать?	28
Серии моделей OpenAI.....	29
Серия GPT-4	29
Серия GPT-3.5	30
Серия InstructGPT-3.....	31
Базовая серия GPT-3.....	32
Серия Codex.....	33
Content Filter	34
Серия DALL-E	34
Серия TTS	35
Модель Whisper.....	35
Модель встраивания.....	36
Модели и цены OpenAI.....	36
Что дальше?.....	38
Использование функции завершения.....	39
Вводный пример.....	39
Роли system, user и assistant.....	42
Роль system.....	42

Роль user	43
Роль assistant.....	43
Завершение чата и обучение на нескольких примерах.....	43
Форматирование вывода	45
Ограничение количества выходных токенов	48
Управление остановкой завершения	50
Температура и галлюцинации.....	55
Параметр top_p	58
Что выбрать – temperature или top_p? В чем разница?.....	60
Потоковая передача ответа API.....	61
Управление повторяемостью: presence_penalty и frequency_penalty.....	62
Что штрафовать – частоту или наличие?.....	64
Управление количеством результатов через API	65
Заключение	66
Продвинутые примеры и разработка промптов.....	67
Что такое разработка промптов?.....	67
Обучение на нескольких примерах: основной метод разработки промптов	69
Избыточная генерация и выбор лучшего варианта	74
Генерация знаний по запросу: создание песни в стиле рэп.....	79
Что такое Apple – фрукт или компания?.....	82
Динамическое управление количеством токенов.....	88
Создание интерактивного помощника в окне командной строки.....	92
Что дальше?.....	101
Встраивание.....	103
Что такое встраивание?.....	103
Варианты применения: от поисковых систем до беспилотных автомобилей	103
Tesla: применение встраиваний в беспилотных автомобилях	104
Kalendar AI: применение встраиваний в управлении продажами	104
Notion: расширенные возможности поиска	104
DALL-E 2: преобразование текста в изображение.....	104
Изучаем встраивание текста.....	105
Встраивания для нескольких входов.....	107
Пример применения: семантический поиск.....	107
Что такое косинусное подобие	108
Семантический поиск и встраивание текста OpenAI	112
За кулисами: как работает встраивание	123
Продвинутые примеры встраивания	125
Рекомендация подходящего сорта кофе.....	125
Разработка более «нечеткого» поиска.....	137
Прогнозирование категории новостей: классификация с помощью встраивания	141
Оценка точности классификатора	146
Точность приложений классификатора в различных сценариях	151

Тонкая настройка и передовые методы работы	153
Обучение на ограниченных примерах	153
Улучшенное обучение на ограниченных примерах.....	154
Практическое применение тонкой настройки.....	154
Полезные приемы тонкой настройки	159
Выбор модели	159
Проверка набора данных	159
Максимальное количество токенов	169
Размер набора данных	169
Тестирование и улучшение обучения (гиперпараметры)	169
Количество эпох	169
Коэффициент скорости обучения	170
Размер пакета	171
Ориентировочная оценка затрат	171
Качество набора данных	173
Экспериментируйте и учитесь	174
Используйте проверочные наборы данных	174
Тестирование модели	175
Анализ результатов	175
 Продвинутый пример тонкой настройки: виртуальный консультант	 177
Набор данных, используемый в примере	177
Подготовка данных	179
Проблемы использования модели в реальных приложениях	187
 Контекст и память: как сделать искусственный интеллект более реалистичным	 189
В чем проблема?	189
Отсутствие контекста = хаос случайности	189
История = контекст.....	191
Недостатки переноса контекста через историю	193
Память «последним вошел – первым вышел» (LIFO).....	193
Проблема с памятью типа LIFO	196
Избирательный контекст	196
 Применение векторной базы данных.....	204
Введение.....	204
Что такое векторная база данных?	204
Пример 1. Использование Weaviate для повышения контекстной зависимости модели	206
Пример 2. Семантический поиск с помощью Weaviate и OpenAI.....	219
Пример 3. Генеративный поиск с помощью Weaviate и OpenAI.....	226

Распознавание и перевод речи с Whisper	236
Что такое Whisper?.....	236
С чего начать?	238
Распознавание и перевод речи.....	239
Использование Whisper SDK в коде Python	240
Использование API OpenAI для преобразования аудиозаписи в текст.....	242
API распознавания	242
API перевода	243
Улучшение качества распознавания речи с Whisper	244
Очистка аудиозаписи	244
Использование подсказки	244
Постобработка полученного текста	246
Преобразование текста в речь	248
Диалог между двумя ИИ на основе OpenAI и Weaviate	251
Генерация аудиофайлов.....	251
Использование аватаров модели.....	268
Что дальше?.....	271
Послесловие	272
Предметный указатель	273

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

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 многих часов 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: FAUN¹, eralabs² и Marketto³. Вы можете найти Аймена в Twitter⁴ и LinkedIn⁵.

¹ <https://faun.dev>.

² <https://eralabs.io>.

³ <https://marketto.dev>.

⁴ <https://twitter.com/@eon01>.

⁵ <https://www.linkedin.com/in/elamriaymen/>.

История OpenAI и ChatGPT

https://t.me/it_books/2

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

Сэм Альтман¹, Илон Маск², Грег Брокман³, Рид Хоффман⁴, Джессика Ливингстон⁵, Питер Тиль⁶, Amazon Web Services (AWS), Infosys и YC Research объявили о создании компании OpenAI и пообещали выделить более 1 млрд долларов 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 Universe⁷ – 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 долларов 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 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.

¹ https://en.wikipedia.org/wiki/Sam_Altman.

² https://en.wikipedia.org/wiki/Elon_Musk.

³ https://en.wikipedia.org/wiki/Greg_Brockman.

⁴ https://en.wikipedia.org/wiki/Reid_Hoffman.

⁵ https://en.wikipedia.org/wiki/Jessica_Livingston.

⁶ https://en.wikipedia.org/wiki/Peter_Thiel.

⁷ <https://openai.com/blog/universe/>.

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 долларов США в 2023 г. и 1 млрд долларов 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 долларов.

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

Большинство людей не знали 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).

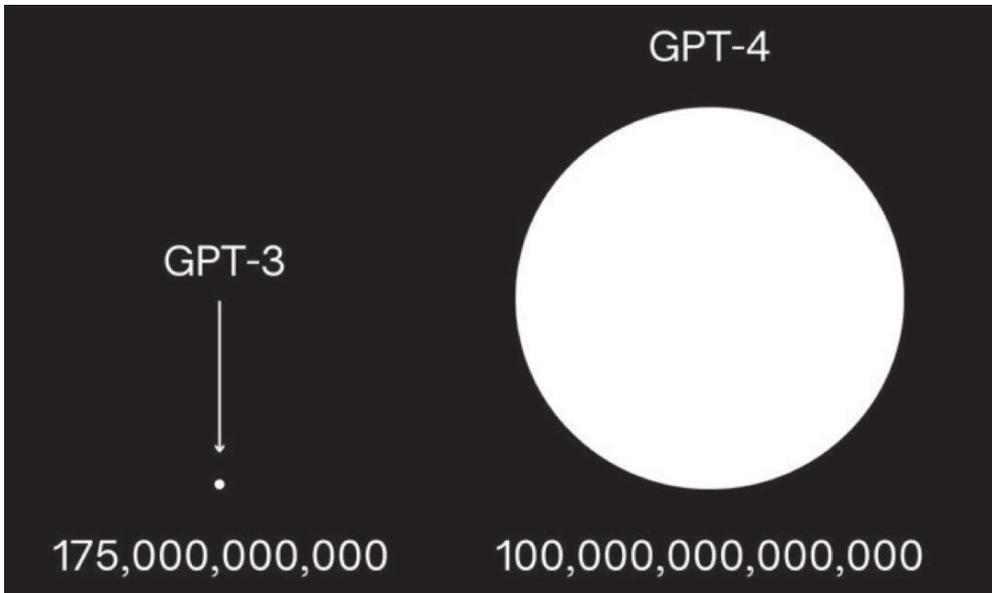
Аббревиатура ChatGPT расшифровывается как Generative Pretrained Transformer (генеративный предварительно обученный трансформер). Эта модель построена на основе семейства *больших языковых моделей* (large language model, LLM). Чат-бот обучен с использованием двух технологий – *обучение с учителем* (supervised learning) и *обучение с подкреплением* (reinforcement learning).

Модель GPT-3 служила основой для ChatGPT до релиза GPT-3.5 15 марта 2022 г., а затем GPT-4 14 марта 2023 г. ChatGPT – это проект, который основан на GPT-3.5 и GPT-4 с добавлением веб-интерфейса, памяти диалогов и других удобных функций. Прочитав это руководство, вы сможете создать свой собственный чат-бот, который потенциально будет лучше, чем универсальный ChatGPT, поскольку вы сможете настроить его в соответствии со своими конкретными потребностями.

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

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».



Наглядное сравнение количества параметров моделей GPT-3 и GPT-4

Интернет полон слухов, спекуляций и некачественного контента. Это верно и в случае искусственного интеллекта. Возможно, вам попадались различные учебные пособия и книги, обещающие научить вас использовать модели OpenAI GPT, но на самом деле предлагающие стандартные наборы запросов к модели (промптов), скопированные в интернете. Эта книга не такая. Это не примитивный набор подсказок; речь идет о понимании принципов и методов, лежащих в основе этих моделей, о том, как использовать их при создании ваших приложений и как их интегрировать с существующими системами и другими инструментами искусственного интеллекта и машинного обучения.

В настоящее время многие проекты используют модели OpenAI в качестве основы для своих продуктов. К наиболее заметным в наши дни относятся:

- GitHub Copilot¹ (с использованием модели OpenAI Codex, потомка GPT-3, настроенной для генерации кода);
- Copy.ai² и Jasper.ai³ (генерация контента для маркетинговых целей);
- университет Дрекслея⁴ (выявление ранних признаков болезни Альцгеймера);
- Algolia⁵ (улучшение возможностей поисковой системы);
- ваш будущий стартап? Почему бы и нет?

¹ <https://github.com/features/copilot>.

² <http://copy.ai>.

³ <http://jasper.ai>.

⁴ <https://drexel.edu/>.

⁵ <https://www.algolia.com/>.

Прочитав эту книгу, вы научитесь эффективно использовать модели OpenAI в своих проектах и, возможно, станете следующим, кто присоединится к этому списку.

ОБ ЭТОЙ КНИГЕ

Знания, которые вы получите из этой книги, относятся к текущему семейству моделей (GPT-3, GPT-3.5, GPT-4 и т. д.) и, вероятно, также пригодятся для GPT-5, если нам суждено дождаться ее выпуска.

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 применять GPT независимо от его версии.

Цель этой книги – предоставить пошаговое руководство по использованию GPT-3.5 и GPT-4 в ваших проектах с помощью API OpenAI. В руководстве рассмотрены и другие модели, такие как Whispers и Text-to-Speech.

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 следующие приложения:

- точно обученные чат-боты для узкой предметной области;
- интеллектуальная диалоговая система с памятью и контекстом;
- современная семантическая поисковая система, использующая RAG и другие методы;
- интеллектуальная система рекомендации сортов кофе, учитывающая ваши вкусы;
- чат-бот, помогающий освоить команды Linux;
- точно настроенная система прогнозирования категорий новостей;
- автономная диалоговая система AI-AI для имитации человеческих разговоров или решения проблем;
- консультант-психотерапевт на основе ИИ, обученный на большом наборе данных бесед о психическом здоровье,
- и многое другое!

Прочитав эту книгу и следуя примерам, вы освоите следующие навыки:

- 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 других творческих применений;
- 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 стемматизации, лемматизации и сокращения текстов;
- применение заполнения контекста, создание смысловых цепочек и прикладных промптов;
- реализация чат-бота с памятью и сохранением контекста;
- создание алгоритмов прогнозирования и обучения без примеров, оценка их точности;
- использование обучения с несколькими примерами и улучшение его точности;
- тонкая настройка моделей, в том числе собственных;
- понимание и использование передового опыта тонкой настройки моделей;
- применение различных методов обучения и классификации с использованием GPT;
- использование встраивания текста аналогично тому, как это делают компании Tesla и Notion;
- понимание и применение семантического поиска, RAG и других передовых инструментов и концепций;
- интеграция баз данных векторных представлений (например, Weaviate) в вашу интеллектуальную систему.

ОСТАВАЙТЕСЬ НА СВЯЗИ

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

Как работает GPT?

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

GPT-4 – это просто обновленная и увеличенная модель, она заметно крупнее и производительнее, чем любая другая предыдущая модель GPT, включая GPT-1/2/3/3.5.

Модель четвертого 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-4 заключается в ее впечатляющем размере – с ошеломляющими 1,76 трлн 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* (natural language processing, NLP), который позволяет модели взвешивать важность различных частей предложения или нескольких предложений при прогнозировании. Являясь частью архитектуры Transformer, он дает возможность нейронной сети достигать высокого качества работы, когда речь идет о задачах NLP. Концепция самовнимания основана на идее, что на каждое слово в предложении могут влиять другие слова в предложении и важность каждого слова можно определить на основе его контекста. Впервые механизм самовнимания был представлен в статье «Внимание – это все, что вам нужно»¹.

Представьте себе, что вы присутствуете на большом званом обеде, где все одновременно делятся историями. Вы пытаетесь следить за всеми разговорами вокруг вас, но, естественно, больше сосредоточены на историях, которые имеют к вам какое-то отношение или содержат важную информацию. Когда люди говорят, вы уделяете свое внимание рассказчику, чье повествование в данный момент наиболее интересно или актуально, иногда переключая

¹ <https://arxiv.org/abs/1706.03762>.

фокус на основе новой информации или связей, которые вы устанавливаете с другими рассказываемыми историями.

В этой аналогии вы похожи на модель трансформера в GPT. Званный ужин представляет собой входной текст, где история каждого гостя представляет собой слово или токен в последовательности. Ваша способность слушать несколько историй одновременно и решать, на какой из них сосредоточиться, аналогична механизму самовнимания. Этот механизм позволяет модели *одно-временно* обрабатывать все части входных данных и определять релевантность или важность каждого слова в контексте всех остальных, подобно тому, как вы в нужный момент настраиваетесь на самую увлекательную историю.

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

```
from transformers import pipeline
generator = pipeline(
    'text-generation',
    model = 'gpt2'
)
generator(
    "Привет, я языковая модель",
    max_length = 30,
    num_return_sequences=3
)
```

Так выглядит пример вывода предыдущего кода:

```
[
  {
    'generated_text': "Привет, я языковая модель. Когда я писала это, ..."
  },
  {
    'generated_text': " Привет, я языковая модель. Я пишу и поддерживаю ..."
  }
]
```

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')
```

```
# Использование модели в режиме 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-4 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-4, 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 о моделях OpenAI (GPT, встраивание и многие другие понятия) и о том, как обращаться к ним из кода на Python, включая эффективное использование API и инструментов, предоставляемых компанией и сообществом AI/ML, для создания прототипов мощных и креативных инструментов и систем. Это не только улучшит ваши навыки применения API GPT, 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 технических аспектах.

Приятного чтения!

Подготовка среды разработки

https://t.me/it_books/2

ВАЖНЫЕ ПРИМЕЧАНИЯ

В этом руководстве для создания файлов и вставки контента непосредственно из командной строки или с помощью скриптов и программ часто используется так называемый *heredoc-синтаксис*.

Heredoc, аббревиатура от «here document» (здесь документ), представляет собой функцию сценариев оболочки, которая упрощает создание многострочного ввода (строк с переносом).

Heredoc-синтаксис начинается с символов `<<`, за которыми следует маркер-разделитель по вашему выбору, продолжается содержимым, которое вы хотите включить, и завершается тем же маркером-разделителем на новой строке. Этот синтаксис особенно полезен для задач сценариев и программирования, требующих создания файлов или ввода нескольких строк для команды.

Чтобы использовать heredoc-синтаксис для создания файлов или добавления содержимого, используйте следующую структуру:

```
cat << EOF > имя_файла
Это строка #1
Это строка #2
..
EOF
```

В этом примере `<<EOF` означает начало раздела heredoc, а `EOF` в отдельной строке указывает на его конец. Содержимое между этими маркерами перенаправляется в файл с именем `имя_файла`. Замените `имя_файла` на имя файла, которое вам нужно, и при необходимости измените содержимое.

Основная цель использования синтаксиса heredoc в этом руководстве – дать читателям возможность точно и легко создавать файлы с помощью одного действия копирования и вставки из электронной версии книги. Такой подход гарантирует, что контент автора будет точно воспроизведен в среде читателя, что снижает вероятность ручных ошибок и делает процесс обучения более эффективным.

Я предлагаю копировать примеры heredoc из электронной версии книги прямо в ваш терминал командной строки, чтобы создать файлы и контент, как это задумано в книге¹.

Пользователи Windows могут выполнять примеры heredoc, применяя подсистему Windows для Linux (Windows Subsystem for Linux, WSL). Кроме того, вы можете вручную создать файлы и вставить их содержимое с помощью текстового редактора. Тем не менее для выполнения примеров в этой книге я рекомендую использовать Unix-подобную среду (предпочтительно систему на базе Debian).

¹ Читатели русского перевода могут скачать архив с готовыми примерами кода на сайте издательства. В таком случае heredoc-синтаксис не нужен. – *Прим. перев.*

Если вы работаете с другой системой, вам может потребоваться настроить некоторые незначительные детали, такие как менеджер пакетов, процесс установки пакетов и метод создания файлов и каталогов. Это не должно стать проблемой, поскольку примеры просты и могут быть легко адаптированы к любой системе.

Еще одно важное замечание, о котором стоит упомянуть, заключается в том, что это руководство построено таким образом, что каждая глава основывается на предыдущей, а темы и концепции переплетаются между собой, чтобы сформировать всестороннее понимание предмета. Важно последовательно проходить главы, заботясь о том, чтобы полностью усвоить каждый раздел, прежде чем переходить к следующему. Терпение и внимание к деталям существенно помогут вашему обучению, позволяя вам в полной мере извлечь выгоду из пошагового развития.

Код, представленный в этом руководстве, предназначен не для пассивного просмотра, а для активного выполнения. Взаимодействие с кодом – его запуск, изменение и наблюдение за результатами – важно для глубокого понимания обсуждаемых концепций. Этот практический подход поможет вам усвоить материал и развить навыки, необходимые для применения знаний в ваших собственных проектах.

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 работать с установщиком пакетов `pip`. Вы можете использовать `pip` для установки пакетов из репозитория пакетов Python и других репозиториях.

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 документации².

Самый простой способ получить его – использовать `pip`:

```
pip install --upgrade pip
pip install virtualenv
pip install virtualenvwrapper
```

Пользователи Windows могут применять следующие команды для установки `virtualenvwrapper-win`³:

¹ <https://github.com/python-virtualenvwrapper/virtualenvwrapper>.

² <https://virtualenvwrapper.readthedocs.io/en/latest/install.html>.

³ <https://github.com/davidmarble/virtualenvwrapper-win/>.

```
pip install --upgrade pip
pip install virtualenv
pip install virtualenvwrapper-win
```

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

Если pip не установлен, проще всего установить его с помощью скрипта get-pip.py¹:

```
curl https://bootstrap.pypa.io/get-pip.py | python3
```

Пользователи Windows могут ввести следующие команды:

```
curl https://bootstrap.pypa.io/get-pip.py
py get-pip.py
```

В итоге в вашей системе должны быть установлены следующие пакеты:

- Python 3.9;
- pip;
- virtualenv;
- virtualenvwrapper (или любой другой менеджер пакетов, который вы предпочитаете).

Я настоятельно рекомендую пользователям Windows создать виртуальную машину с операционной системой Debian Linux, так как большинство примеров, представленных в этой книге, выполнялись и тестировались в системе Debian Linux. Хотя многие команды и примеры кода будут работать под Windows, в некоторых случаях могут потребоваться небольшие изменения.

Далее создадим виртуальную среду:

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

После создания виртуальной среды активируйте ее:

```
workon chatgptforpythondevelopers
```

ПОЛУЧЕНИЕ КЛЮЧА API OPENAI

Следующим шагом является создание ключей API, которые позволят вам получить доступ к официальному API, предоставляемому OpenAI².

¹ <https://bootstrap.pypa.io/get-pip.py>.

² Сайт и сервисы OpenAI недоступны для пользователей с российским IP-адресом. Для создания учетной записи вам понадобится доступ в интернет с европейским или американским 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 с российских IP-адресов невозможен. 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.

Перейдите по адресу <https://platform.openai.com> и создайте учетную запись¹.

Ключ API должен принадлежать организации, поэтому 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.

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 применение Chronology², неофициальной библиотеки, предоставленной OthersideAI. Однако похоже, что эта библиотека больше не обновляется, поэтому не рекомендуется использовать ее для производственных целей.

Чтобы установить официальные средства интеграции Python, выполните следующую команду:

```
pip install openai==1.9.0
```

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

ТЕСТИРОВАНИЕ КЛЮЧЕЙ API

Прежде всего объявим ключ API и ID организации как переменные окружения (вместо xxx подставьте ваши данные):

```
export API_KEY=xxx
export ORG_ID=xxx
```

Теперь выполните следующую команду для тестирования доступа к API (вводить без переноса строки):

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

Если в вашей учетной записи OpenAI определена только одна организация, вы можете выполнить ту же команду без указания ID организации:

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

Команда должна вернуть список моделей, доступных через API, таких как davinci, ada и многих других.

¹ <https://platform.openai.com/api-keys>.

² <https://github.com/OthersideAI/chronology>.

Далее мы выполним тест с применением Python SDK.

Создайте файл с именем `.env` и сохраните в нем ключ API и ID организации:

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

Для тестирования работы в Python SDK выполните следующий код:

```
cat << EOF > test_api.py
import os
from openai import OpenAI
from pprint import pprint

# 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 ID организации
client = OpenAI(
    api_key=os.environ['API_KEY'],
    organization=os.environ['ORG_ID']
)

# Печать объекта client
pprint(vars(client))
EOF
```

Запустите приведенный выше код, выполнив команду

```
python test_api.py
```

Мы будем использовать аутентификацию с помощью ключа API и ID организации в каждом примере кода на протяжении всей книги. Поэтому нам будет полезно упорядочить код.

Создайте папку с именем `src` и внутри нее создайте файл с именем `api.py`:

```
cat << EOF > src/api.py
import os
from openai import OpenAI

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

# Создание объекта client на уровне модуля
client = OpenAI(
```

```
api_key=os.environ['API_KEY'],  
organization=os.environ['ORG_ID']  
)
```

Создайте файл `.env` при помощи следующей команды:

```
cat << EOF > src/.env  
API_KEY=$API_KEY  
ORG_ID=$ORG_ID  
EOF
```

Для тестирования доступа к API необходимо выполнить следующий код:

```
cat << EOF > src/test_api_1.py  
from api import client  
from pprint import pprint  
  
# Печать объекта client  
pprint(vars(client))  
EOF
```

Запустите этот код при помощи команды:

```
python src/test_api_1.py
```

Вы должны увидеть тот же вывод, что и при запуске предыдущего кода.

Доступные модели и выбор оптимального варианта

Модели OpenAI и ВАЖНЫЕ СОГЛАШЕНИЯ

Обратившись к конечной точке API `models`, вы получите список доступных моделей.

```
cat << EOF > src/test_api_2.py
from api import client

models = client.models.list()
for model in models:
    print(vars(model))
EOF
```

Запустите этот код:

```
python src/test_api_2.py
```

Вы должны получить вывод наподобие следующего:

```
{
  'id': 'curie-search-query',
  'created': 1651172509,
  'object': 'model',
  'owned_by': 'openai-dev'
}

{
  'id': 'babbage-search-query',
  'created': 1651172509,
  'object': 'model',
  'owned_by': 'openai-dev'
}

{
  'id': 'dall-e-3',
  'created': 1698785189,
  'object': 'model',
  'owned_by': 'system'
}
... и т. д.
```

Инструмент разработки OpenAI Python SDK интуитивно понятен и прост в использовании. Нам достаточно применить метод `list()`, чтобы получить список всех доступных моделей.

Выведем на печать только ID моделей:

```
cat << EOF > src/test_api_3.py
from api import client

models = client.models.list()
for model in models:
    print(model.id)
EOF
```

Запустите код:

```
python src/test_api_3.py
```

Вывод должен содержать список всех доступных моделей; он будет выглядеть примерно так:

```
dall-e-3
dall-e-2
GPT-4
davinci-002
babbage-002
whisper-1
GPT-3.5-Turbo-16k
GPT-3.5-Turbo
GPT-3.5-Turbo-Instruct
```

Модели Babbage-002 и Davinci-002 являются частью серии GPT-3, разработанной OpenAI, и служат заменой или обновлением предыдущих моделей:

- Babbage-002. Служит заменой базовым моделям GPT-3 Ada и Babbage. Она оптимизирована для задач, требующих баланса между производительностью и экономической эффективностью, и подходит для задач средней сложности, таких как классификация текста, семантический поиск и базовое понимание языка;
- Davinci-002. Заменяет базовые модели GPT-3 Curie и Davinci. Известна своими расширенными возможностями понимания и генерации языков и идеально подходит для более сложных задач, требующих детального понимания, таких как создание контента, сложный языковой перевод и сложный логический вывод;
- DALL-E 3 и DALL-E 2. Обе являются моделями генерации изображения. DALL-E 3 представляет собой усовершенствованную версию базовой модели DALL-E 2 с более продвинутыми возможностями создания реалистичных изображений высокого разрешения на основе текстовых описаний;
- Whisper-1. Эта модель специализируется на распознавании и транскрипции речи. Она была разработана для точного преобразования устной речи в письменный текст. В будущем вы можете увидеть более продвинутые версии этой модели (например, Whisper-2, Whisper-3 и т. д.);
- GPT-3.5-Turbo и GPT-3.5-Turbo-Instruct. Обе эти модели представляют собой варианты GPT-3.5, предназначенные для более эффективного и оперативного создания текста;
- GPT-4. Это большая мультимодальная модель, которая может принимать как текст, так и изображения. Она является преемником GPT-3 и способна выполнять широкий спектр задач.

Некоторые из моделей, перечисленных в выводе, являются динамически, поскольку они постоянно обновляются со стороны OpenAI. Например, названия GPT-3.5-Turbo, GPT-4 и GPT-4-32k указывают на последнюю версию модели. На момент написания книги последней версией модели GPT-4 была GPT-4-0613, поэтому когда ваш код вызывает конечную точку API GPT-4, вы фактически используете GPT-4-0613. В будущем, когда OpenAI выпустит новую модель GPT-4, конечная точка API будет указывать на новую модель. Вот почему рекомендуется указывать в коде конкретную версию модели, если у вас есть особые требования.

Исторически модели назывались «движками», но OpenAI отказалась от термина «движок» в пользу «модели». В некоторых публикациях эти термины используют как взаимозаменяемые, но правильное название – «модель». При использовании API запросы, использующие старые имена, по-прежнему будут работать, поскольку OpenAI обеспечила обратную совместимость, но в будущем это может измениться. Однако наименования некоторых моделей следует вручную заменить в коде на более новые. Рекомендуется использовать обновленные модели, чтобы избежать проблем в будущем.

Некоторые модели не упомянуты в перечне выше, например *модели модерации*. Эти модели используются для обнаружения нежелательного контента в тексте и его фильтрации. Применение этих моделей не является обязательным, но настоятельно рекомендуется, если вы создаете общедоступное приложение. Вы ведь не хотите, чтобы ваши пользователи получали оскорбительные или незаконные результаты?

КАКУЮ МОДЕЛЬ ЛУЧШЕ ИСПОЛЬЗОВАТЬ?

На этот вопрос нет однозначного ответа. Выбор зависит от вашей задачи и типа приложения, которое вы создаете. Как правило, вам следует выбирать самую последнюю доступную модель. Например, с точки зрения функциональности, если вы создаете чат-бот, вам следует использовать модель GPT-3.5-Turbo-Instruct. Если вы создаете приложение, генерирующее изображения из текста, вам следует использовать модель DALL-E 3. Если вашему приложению требуются дополнительные возможности, вам следует использовать модель GPT-4.

Второй фактор, который следует учитывать, – это цена. Некоторые модели дороже других. Например, GPT-3.5-Turbo – самая экономичная модель серии GPT-3.5.

Еще одним важным фактором является контекстное окно. У некоторых моделей контекстное окно больше, чем у других. Например, модель GPT-4-32k имеет контекстное окно из 32 768 токенов (около 50 страниц англоязычного текста), а модель GPT-4-0613 имеет контекстное окно из 8192 токенов (около 12 страниц англоязычного текста). Это означает, что модель GPT-4-32k может обрабатывать больше текста, чем модель GPT-4-0613.

В следующих разделах мы подробнее рассмотрим основные доступные модели; это поможет вам понять различия между ними и выбрать подходящую модель для вашего применения. Однако важно понимать, что некоторые из перечисленных ниже моделей устарели и, вероятно, не будут доступны в будущем. OpenAI постоянно совершенствует свои модели и отключает старые.

СЕРИИ МОДЕЛЕЙ OPENAI

Серия GPT-4

GPT-4 – это новейшая серия моделей, разработанная OpenAI, которая является преемником GPT-3. Одно из основных отличий от GPT-3 заключается в том, что модели GPT-4 *мультимодальные*. Это означает, что GPT-4 может принимать как изображения, так и текстовые входные данные.

Мультимодальное глубокое обучение обогащает модели информацией, собранной из различных модальностей данных, таких как текст, изображения, видео, аудио и данные датчиков. Это ставит его на шаг впереди традиционного машинного обучения, которое было ограничено одной модальностью, что позволяет решить проблему сложности, связанной с реальными данными, в которых часто смешаны различные источники. Цель мультимодального глубокого обучения – создать единое пространство представления, собирая разнообразную информацию из этих модальностей и применяя ее для решения таких задач, как создание подписей к изображениям, понимание речи и обработка естественного языка.

GPT-4 демонстрирует качество работы человеческого уровня по нескольким профессиональным и академическим показателям. Примечательно, что GPT-4 вошла в 10 % лучших экзаменуемых во время имитации экзамена на адвоката, что является значительным улучшением по сравнению с GPT-3.5, которая вошла в нижние 10 % по оценке за экзамен, согласно сообщению в блоге¹, опубликованному OpenAI.

GPT-4 доступна как для разработчиков, использующих API, так и для обычных пользователей, применяющих ChatGPT.

В семействе GPT-4 есть две основные модели:

- стандартная модель GPT-4 (8k) с длиной контекста 8192 токена;
- расширенная модель GPT-4 (32k) с длиной контекста 32 768 токенов (около 50 страниц текста).

Однако для конкретных случаев использования доступны и другие модели, например GPT-4-Turbo – модель, которая принимает изображения в качестве входных данных в API продолжения чата.

В следующей таблице суммированы основные различия этих моделей.

Название модели	Количество параметров, млрд	Описание	Длина контекста	Рабочий статус
GPT-4	1000*	Стандартная модель GPT-4 с контекстом длиной 8192 токена	8192	Доступна
GPT-4-32k	1000*	Расширенная модель GPT-4 с контекстом длиной 32 768 токенов	32 768	Доступна

¹ <https://openai.com/research/gpt-4>.

Окончание табл.

Название модели	Количество параметров, млрд	Описание	Длина контекста	Рабочий статус
GPT-4-Turbo	1000*	Модель, принимающая на вход изображения в API продолжения чата	128 000	Предпросмотр

* Количество параметров по неофициальной оценке Semafor¹.

Серия GPT-3.5

GPT-3.5 – это более мощная версия базовой модели GPT-3. Она способна выполнять большее количество задач и имеет больше параметров, чем базовая GPT-3.

Серия GPT-3.5 представляет три модели:

- text-davinci-002: предлагает возможности, аналогичные text-davinci-003, но использует управляемую тонкую настройку вместо обучения с подкреплением;
- text-davinci-003: блестяще справляется с языковыми задачами, обеспечивая превосходное качество, расширенные результаты и последовательное соблюдение инструкций по сравнению с моделями Curie, Babbage или Ada. Она также поддерживает встроенные дополнения в тексте;
- GPT-3.5-Turbo: самая продвинутая модель GPT-3.5, предназначенная для взаимодействия в чате. Ее использование обходится лишь в 1/10 стоимости применения text-davinci-003.

В следующей таблице приведены основные различия между этими тремя моделями.

Название модели	Количество параметров, млрд	Описание	Длина контекста	Рабочий статус
text-davinci-002	–	Имеет те же функции, что и text-davinci-003, но обучается по другому методу, ориентированному на обучение с учителем	–	Устарела, заменена на GPT-3.5-Turbo-Instruct (требуется замена вручную в коде)
text-davinci-003	–	Превосходно справляется с языковыми задачами, обеспечивая более длинные, последовательные и высококачественные результаты. Также поддерживает внутри-текстовое завершение	–	Устарела, заменена на GPT-3.5-Turbo-Instruct (требуется замена вручную в коде)

¹ <https://www.semafor.com/article/03/24/2023/the-secret-history-of-elon-musk-sam-altman-and-openai>.

Окончание табл.

Название модели	Количество параметров, млрд	Описание	Длина контекста	Рабочий статус
GPT-3.5-Turbo	175	Самая совершенная модель GPT-3.5, оптимизированная для чата и доступная за небольшую часть стоимости text-davinci-003	4096	Доступна
GPT-3.5-Turbo-Instruct	175	Самая совершенная модель GPT-3.5, оптимизированная для чата и доступная за небольшую часть стоимости text-davinci-003	4096	Доступна
GPT-3.5-Turbo-16k	175	Самая совершенная модель GPT-3.5, оптимизированная для чата и доступная за небольшую часть стоимости text-davinci-003	16 384	Доступна

Серия InstructGPT-3

Серия InstructGPT-3 превосходит базовую модель GPT-3 по части следования инструкциям. Она состоит из четырех различных моделей:

- text-ada-001: созданная для выполнения базовых задач, она зачастую является самой быстрой и экономичной в линейке GPT-3;
- text-babbage-001: эффективна для простых задач, обеспечивает высокую производительность и доступность;
- text-curie: высококлассная, обеспечивает более быстрые результаты по цене ниже, чем у Davinci;
- text-davinci-001: более ранняя версия модели высшего уровня в линейке GPT-3, способная выполнять задачи, которые могут выполнять другие модели GPT-3, обычно с ограниченным контекстом.

Модели InstructGPT, как правило, менее склонны к болтовне и более ориентированы на задачи, что делает их идеальными для выполнения четких и лаконичных заданий. Это может сэкономить время и использование токенов, особенно если задача проста и контекст ясен.

В следующей таблице приведены основные различия между этими моделями.

Название модели	Количество параметров, млрд	Описание	Рабочий статус
text-ada-001	0.35	Подходит для базовых задач, обычно самая быстрая и доступная в линейке GPT-3	Устарела и заменена на GPT-3.5-Turbo-Instruct (требуется обновление кода вручную)

Окончание табл.

Название модели	Количество параметров, млрд	Описание	Рабочий статус
text-babbage-001	175	Модельный ряд GPT-3. Эффективно решает прямые задачи с меньшими затратами	Устарела и заменена на GPT-3.5-Turbo-Instruct (требуется обновление кода вручную)
text-curie-001	6.7	Высококласная, но более быстрая и экономичная, чем Davinci	Устарела и заменена на GPT-3.5-Turbo-Instruct (требуется обновление кода вручную)
text-davinci-001	175	Более ранняя версия модели высшего уровня в линейке GPT-3. Способна решать задачи, аналогичные другим моделям GPT-3, но может потребовать большего контекста	Устарела и заменена на GPT-3.5-Turbo-Instruct (требуется обновление кода вручную)

Базовая серия GPT-3

Базовая серия GPT-3 состоит из четырех моделей (A, B, C, D), каждая из которых отличается от остальных по скорости и качеству.

- Ada (A): подходит для простых задач, обычно самая быстрая и экономичная в линейке GPT-3;
- Babbage (B): выполняет элементарные задачи быстро и с меньшими затратами;
- Curie (C): высокая эффективность, более высокое качество при более низкой цене, чем у Davinci;
- Davinci (D): вершина серии GPT-3, превосходящая по качеству результаты работы других моделей.

Базовые модели GPT-3 способны понимать человеческий язык и генерировать текст, который зачастую неотличим от созданного человеком. Они также способны выполнять большое количество задач, в том числе:

- классификацию текста;
- семантический поиск;
- обобщение текста;
- анализ настроений;
- генерацию контента;
- завершение текста;
- перевод текста;
- извлечение текста.

Однако эти модели не самые точные в серии GPT-3.

- Ada и Babbage устарели и заменены на Babbage-002. Обновления выполняются автоматически OpenAI;

- Curie устарела и заменена на Davinci-002. Обновления выполняются автоматически OpenAI;
- Text-Ada, Text-Babbage и Text-Davinci устарели и заменены на GPT-3.5-Turbo-Instinct. Обновления должны выполняться разработчиком вручную.

В следующей таблице суммированы основные различия между моделями.

Название модели	Количество параметров, млрд	Описание	Рабочий статус
ada	0.35	Подходит для базовых задач, обычно самая быстрая и доступная в линейке GPT-3	Устарела и заменена на babbage-002 (автоматическое обновление)
babbage	1.3	Эффективно и с меньшими затратами справляется с прямыми задачами	Устарела и заменена на babbage-002 (автоматическое обновление)
curie	6.7	Высококласная, но более быстрая и экономичная, чем Davinci	Устарела и заменена на davinci-002 (автоматическое обновление)
davinci	175	Вершина моделей GPT-3, превосходно справляется с задачами, которые могут выполнить другие, часто с превосходными результатами	Устарела и заменена на davinci-002 (автоматическое обновление)
babbage-002	1.3	Служит заменой базовым моделям GPT-3 Ada и Babbage	Доступна
davinci-002	175	Заменяет базовые модели GPT-3 Curie и Davinci	Доступна

Серия Codex

Серия Codex была анонсирована в середине 2021 г. Это потомок GPT-3, который также был обучен на коде из 54 млн репозиториях GitHub и поддерживает инструмент автодополнения кода GitHub Copilot.

Серия Codex состоит из двух моделей:

- code-davinci-002;
- code-cushman-001.

Модели серии Codex объявлены устаревшими в марте 2023 г. и заменены на GPT-3.5-Turbo. Следующая таблица представлена здесь исключительно для исторических целей.

Название модели	Описание	Статус
code-davinci-002	Самая эффективная модель Codex. Отлично справляется с переводом естественного языка в код и поддерживает вставку дополнительных элементов	Устарела и заменена на GPT-3.5-Turbo-Instruct (требуется обновление вручную)
code-cushman-001	Мощная и быстрая, подходит для многих задач генерации кода, доступнее, чем Davinci	Устарела и заменена на GPT-3.5-Turbo-Instruct (требуется обновление вручную)

Content Filter

Как следует из названия, это фильтр конфиденциального контента.

Используя этот фильтр, вы можете обнаружить конфиденциальный или небезопасный текст, сгенерированный API. Этот фильтр может классифицировать текст по трем категориям:

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

Если вы создаете приложение, которое будут использовать внешние пользователи, вы можете наложить на вывод модели фильтр контента, чтобы определить, возвращает ли она какой-либо неприемлемый контент.

На момент написания доступны две модели:

- text-moderation-stable;
- text-moderation-latest.

В следующей таблице приведены основные различия между ними.

Название модели	Описание	Длина контекста	Статус
text-moderation-stable	Стабильная версия фильтра контента	32 768	Доступна
text-moderation-latest	Последняя версия фильтра контента, более высокая точность	32 768	Доступна

Серия DALL-E

Это семейство моделей, которые могут генерировать изображения из текстовых описаний. Они продолжают линию ImageGPT и базируются на принципах GPT-3. В серию DALL-E входят две модели: DALL-E 2 и DALL-E 3.

В следующей таблице приведены основные различия между моделями.

Параметр	DALL-E 2	DALL-E 3
Разрешение, пикс.	512×512	1024×1024
Метод синтеза изображения	Дискретный вариационный автокодировщик	Диффузионная модель

Окончание табл.

Параметр	DALL-E 2	DALL-E 3
Интеграция с ChatGPT	Нет прямой интеграции	Интегрирована с ChatGPT
Следование сложным промптам	Иногда неправильно интерпретирует или пропускает элементы	Отлично выполняет сложные промпты
Генерация текста по изображению	Часто генерирует запутанный, нерелевантный текст	Генерирует качественный, соответствующий контексту текст
Прорисовка деталей человеческого тела	Затрудняется прорисовывать детали изображения человека	Хорошо прорисовывает такие детали, как лица и руки
Привлекательность изображений	Генерирует стандартные изображения	Производит креативные, эмоционально окрашенные изображения
Управление безопасностью	Несколько мер безопасности	Расширенные настройки безопасности для предотвращения некорректного контента
Классификатор происхождения	Эта функция отсутствует	Содержит классификатор происхождения
Контроль творчества	Может нарушать авторские права	Соблюдает права реальных художников
Доступность	В ранних версиях	Доступна в ChatGPT Plus и корпоративным клиентам через API, запланированы следующие релизы
Параметры	3.5 млрд	–

Серия TTS

В дополнение к моделям продолжения чата и генерации изображений OpenAI также предлагает серию моделей преобразования текста в речь (text-to-speech, TTS). В этой серии две модели: TTS-1 и TTS-1-HD. Они представлены в следующей таблице.

Название модели	Описание	Статус
tts-1	Модель, оптимизированная для скорости	Доступна
tts-1-hd	Модель, оптимизированная для качества	Доступна

Модель Whisper

Whisper¹ – это модель распознавания речи с открытым исходным кодом, которая может преобразовывать устную речь в письменный текст. Эта модель

¹ <https://github.com/openai/whisper>.

была обучена на 680 тысячах часов многоязычных данных и поддерживает следующие языки: африкаанс, арабский, армянский, азербайджанский, белорусский, боснийский, болгарский, каталанский, китайский, хорватский, чешский, датский, голландский, английский, эстонский, финский, французский, галисийский, немецкий, греческий, иврит, хинди, венгерский, исландский, индонезийский, итальянский, японский, каннада, казахский, корейский, латышский, литовский, македонский, малайский, маратхи, маори, непальский, норвежский, персидский, польский, португальский, румынский, русский, сербский, словацкий, словенский, испанский, суахили, шведский, тагальский, тамильский, тайский, турецкий, украинский, урду, вьетнамский и валлийский.

Она также способна расшифровывать и переводить речь.

Модель представлена в пяти размерах:

- крошечная (tiny);
- базовая (base);
- маленькая (small);
- средняя (medium);
- большая (large).

Различия между этими размерами показаны в следующей таблице.

Размер	Параметры, млн	Английская модель	Многоязычная модель	Занимаемый объем VRAM, ГБ	Относительная скорость	Доступность через API
tiny	39	tiny.en	tiny	~1	~32x	–
base	74	base.en	base	~1	~16x	–
small	244	small.en	small	~2	~6x	–
medium	769	medium.en	medium	~5	~2x	–
large	1550	нет	large	~10	1x	Доступна

Модель встраивания

Модель встраивания (embedding model) – это модель, которую можно использовать для генерации встраивания текста в векторное пространство. Эта операция часто используется в приложениях машинного обучения и глубокого обучения. Лучшим выбором для этой задачи является модель text-embedding-ada-002, которая более точна, чем остальные модели¹.

МОДЕЛИ И ЦЕНЫ OPENAI

OpenAI предлагает широкий выбор моделей для различных задач. Можно предположить, что более мощные модели стоят дороже, но в данном случае это не так. Расценки определяются не емкостью модели, а другими факторами.

¹ <https://openai.com/blog/new-and-improved-embedding-model>.

Например, GPT-3.5-Turbo – наиболее передовая модель серии GPT-3.5 – стоит 1/10 от тарифа text-davinci-003. Актуальные цены можно найти на странице по адресу <https://openai.com/api/pricing/>.

Ниже приведена цена на основные модели на момент написания книги.

Модель	Цена за вход	Цена за выход
GPT-4-1106-Preview	\$0.01 / 1 тыс. токенов	\$0.03 / 1 тыс. токенов
GPT-4-1106-Vision-Preview	\$0.01 / 1 тыс. токенов	\$0.03 / 1 тыс. токенов
GPT-4	\$0.03 / 1 тыс. токенов	\$0.06 / 1 тыс. токенов
GPT-4-32k	\$0.06 / 1 тыс. токенов	\$0.12 / 1 тыс. токенов
GPT-3.5-Turbo-1106	\$0.0010 / 1 тыс. токенов	\$0.0020 / 1 тыс. токенов
GPT-3.5-Turbo-Instruct	\$0.0015 / 1 тыс. токенов	\$0.0020 / 1 тыс. токенов
davinci-002	\$0.0020 / 1 тыс. токенов	\$0.0020 / 1 тыс. токенов
babbage-002	\$0.0004 / 1 тыс. токенов	\$0.0004 / 1 тыс. токенов
DALL-E 3 (стандартная, 1024×1024)		\$0.040 / изображение
DALL-E 3 (стандартная, большое разрешение)		\$0.080 / изображение
DALL-E 3 (HD, 1024×1024)		\$0.080 / изображение
DALL-E 3 (HD, большое разрешение)		\$0.120 / изображение
DALL-E 2 (1024×1024)		\$0.020 / изображение
DALL-E 2 (512×512)		\$0.018 / изображение
DALL-E 2 (256×256)		\$0.016 / изображение
Whisper	\$0.006 / мин	
TTS	\$0.015 / 1 тыс. букв	
TTS HD	\$0.015 / 1 тыс. букв	
ada v2	\$0.0001 / 1 тыс. токенов	

Цены на старые модели можно найти в разделе устаревших моделей¹.

Модель ценообразования прозрачна и понятна. Кроме того, когда вы регистрируетесь на сайте OpenAI, вы получаете бонус в размере 18 долларов США, который можно использовать в течение первых трех месяцев. В большинстве случаев использования этого более чем достаточно для начала работы.

¹ <https://platform.openai.com/docs/deprecations/>.

Что дальше?

Тот факт, что OpenAI регулярно добавляет новые модели и удаляет старые, может сбить с толку. Генеративный ИИ – это быстро развивающаяся область, поэтому модели постоянно совершенствуются. Лучший способ быть в курсе последних моделей – следить за официальным блогом¹.

На данный момент наиболее точными и обновленными моделями для генерации текста, рассуждений и сложных задач, а также генерации программного кода являются GPT-3.5 и GPT-4. Они пока остаются самыми мощными и будут поддерживаться и улучшаться в будущем. Многие другие модели могут еще быть доступны в течение некоторого времени, но они будут либо закрыты, либо заменены более новыми моделями.

GPT-4 дороже, чем GPT-3.5, но она мощнее и может выполнять более сложную работу. Если вы разрабатываете приложение для развертывания на производстве, нужно сразу начинать с GPT-4. Если вы разрабатываете прототип или экспериментальное решение, имеет смысл работать с GPT-3.5, поскольку это обойдется дешевле.

GPT-3.5

Reasoning



Speed



Conciseness



GPT-4

Reasoning



Speed



Conciseness



Сравнение моделей GPT-3 и GPT-4

По мере изучения этих моделей вы лучше поймете, как использовать API для создания, генерации и редактирования текста и изображений, применяя либо базовые модели, либо ваши собственные тонко настроенные версии. Экспериментируя с практическими примерами, вы лучше поймете, что такое токены и их цены. К концу книги вы сможете создавать готовые к применению интеллектуальные приложения, используя экосистему OpenAI.

¹ <https://openai.com/blog/>.

Использование функции завершения

Модели OpenAI были обучены как *модели завершения* (completion model)¹. Это означает, что фактически они предназначены для прогнозирования следующего слова в последовательности текста. Это обычная задача при обработке естественного языка, а также основа для многих других задач, таких как перевод, ответы на вопросы, анализ текста и многое другое.

Чтобы использовать завершение в чате, вам необходимо иметь ключ OpenAI API. К этому моменту вы уже должны были его получить, как описано в начале книги. Затем вам нужно использовать конечную точку API, чтобы отправить запрос к API и получить ответ.

OpenAI предоставляет несколько моделей завершения. Однако если вы только начинаете, выбирайте между GPT-3.5-Turbo или GPT-4.

Вводный пример

Давайте начнем чат с GPT-3.5:

```
cat > src/app_1.py << EOF
from api import client

model = "gpt-3.5-turbo"
messages = [
    {
        "role": "system",
        "content":
            "Ты умный и креативный человек."
    },
    {
        "role": "user",
        "content": "Всем привет!"
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages
)
```

¹ Завершение представляет собой классическую одноходовую (one-turn) задачу машинного обучения, где пользователь начинает задачу промптом, а модель *завершает* ее ответом. Даже длинный диалог человека с моделью – это последовательность одноходовых задач, где предыдущие промпты и ответы включаются в состав следующего промпта для сохранения контекста диалога. – *Прим. перев.*

```
print(response)
EOF
```

Теперь запустим этот скрипт:

```
python src/app_1.py
```

API должен вернуть ответ, показанный ниже (здесь он структурирован для читаемости):

```
ChatCompletion(
  id='chatcmpl-8jvdNPNv7923lcVHtn3zjNt3GqkAY',
  choices=[
    Choice(
      finish_reason='stop',
      index=0,
      logprobs=None,
      message=ChatCompletionMessage(
        content='Привет! Чем я могу помочь?',
        role='assistant',
        function_call=None,
        tool_calls=None
      )
    )
  ],
  created=1705956997,
  model='gpt-3.5-turbo-0613',
  object='chat.completion',
  system_fingerprint=None,
  usage=CompletionUsage(
    completion_tokens=9,
    prompt_tokens=22,
    total_tokens=31
  )
)
```

Ответ модели представляет собой объект `ChatCompletion`. Этот объект содержит следующие ключи:

- `id`: это идентификатор завершения. Вы можете использовать этот идентификатор, чтобы получить нужное завершение позже;
- `created`: это временная метка создания завершения;
- `model`: это модель, которая использовалась для генерации завершения;
- `object`: это тип объекта. В данном случае это завершение чата;
- `system_fingerprint`: параметр, используемый для идентификации внутренней конфигурации модели OpenAI, помогающий отслеживать, влияют ли изменения в этой конфигурации на детерминизм вывода;
- `usage`: это объект, который содержит количество токенов, использованных в приглашении, количество токенов, сгенерированных моделью, и общее количество токенов (подсказка + сгенерированные).

Далее в ответе идет список вариантов – `choices`. Каждый вариант содержит следующие ключи:

- `finish_reason`: это причина остановки завершения. Значениями могут быть `stop`, `length`, `function_call`, `content-filter` или `null`. Причина `stop`

означает, что генерация завершения остановлена, поскольку достигнута последовательность остановки. `length` означает, что достигнуто максимальное количество токенов. `function_call` означает, что завершение остановлено, поскольку было достигнуто обращение к функции. `content-filter` означает, что контент был заблокирован фильтром контента, `null` означает, что API еще нужно сгенерировать больше токенов;

- `index`: это индекс выбора в списке вариантов;
- `logprobs`: это список логарифмических вероятностей для каждого токена в завершении. Это полезно, если вы хотите узнать вероятность появления каждого токена в завершении;
- `message`: это сообщение, созданное моделью. Оно содержит следующие ключи:
 - ◆ `content`: это содержание сообщения;
 - ◆ `role`: это роль сообщения. Это может быть `user` или `assistant`;
 - ◆ `function_call`: это вызов функции, которая использовалась для создания сообщения. Это полезно, если вы хотите знать, как было создано сообщение;
 - ◆ `tool_calls`: это список вызовов инструментов, которые использовались для создания сообщения. Это полезно, если вы хотите знать, как было создано сообщение.

Для получения контента сообщения можно воспользоваться следующей строкой кода:

```
print(response.choices[0].message.content)
```

Полный код выглядит так:

```
cat > src/app_2.py << EOF
from api import client

model = "gpt-3.5-turbo"

messages = [
    {
        "role": "system",
        "content":
            "Ты умный и креативный человек."
    },
    {
        "role": "user",
        "content": "Всем привет!"
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages
)

print(response.choices[0].message.content)
EOF
```

Выполним этот код:

```
python src/app_2.py
```

Роли SYSTEM, USER И ASSISTANT

Когда мы отправляли запрос в API, мы использовали параметр `messages`. Этот параметр представляет собой список словарей. Каждый словарь содержит роль `role` и содержание сообщения `content`. Возможны три роли:

- `system`: это начальный промпт, используемый для настройки контекста и поведения помощника;
- `user`: эта роль представляет пользователя и используется для отправки сообщения модели;
- `assistant`: эта роль представляет помощника (модель ИИ) и используется для отправки сообщения пользователю.

Роль system

Ключу `messages` должно соответствовать как минимум одно сообщение роли `system`. Вот пример:

```
cat > src/app_3.py << EOF
from api import client

model = "gpt-3.5-turbo"

messages = [
    {
        "role": "system",
        "content": "Ты умный и креативный человек."
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages
)

print(response.choices[0].message.content)
EOF
```

Содержимое сообщения с ролью `system` использовалось выше для определения поведения помощника. В этом случае модель предназначена для умного и креативного реагирования на действия пользователя. Это сообщение также может установить контекст разговора. Например, если вы хотите обсудить фильмы, то можете задать контекст, используя следующее сообщение:

```
{
    "role": "system",
    "content": "Ты знаток кинофильмов."
}
```

Роль user

Список `messages` может содержать одно или несколько сообщений с ролью `user`. Вот пример:

```
cat > src/app_4.py << EOF
from api import client

model = "gpt-3.5-turbo"

messages = [
    {
        "role": "system",
        "content": "Ты умный помощник."
    },
    {
        "role": "user",
        "content": "Всем привет."
    },
    {
        "role": "user",
        "content": "Как поживаете?"
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages
)

print(response.choices[0].message.content)
EOF
```

Содержимое сообщения с ролью `user` используется для определения инструкций, передаваемых модели. В приведенном выше примере нет явных инструкций, но модель во всех случаях инициирует диалог.

Роль assistant

Использовать роль `assistant` не обязательно, но в некоторых случаях она может оказаться полезной. Например, если вы хотите научить модель реагировать на пользователя определенным образом, то можете использовать роль `assistant`, чтобы показать модели, как реагировать. Примеры будут показаны немного позже.

ЗАВЕРШЕНИЕ ЧАТА И ОБУЧЕНИЕ НА НЕСКОЛЬКИХ ПРИМЕРАХ

Обучение на нескольких примерах (few-shot learning) – это метод, который позволяет обучать модель на небольшом наборе данных для выполнения конкретной задачи. Это полезно, когда вам не нужно или вы не можете обучать модель на большом наборе данных.

Обучение на нескольких примерах – это способ создания промптов, который позволяет вам получать конкретные выходные данные модели на основе подсказки.

Рассмотрим пример, в котором мы обучаем модель преобразовывать любое предложение в соответствии с Чикагским руководством по стилю. Этот стиль применяется в США и некоторых других странах. В соответствии с ним в заголовках статей и названиях книг первая буква каждого слова пишется с заглавной буквы, за исключением артиклей, союзов и предлогов. Вот некоторые примеры:

- хороший пример: «Толстая ленивая кошка залезла на забор» -> «Толстая Ленивая Кошка Залезла на Забор»;
- плохой пример: «Толстая ленивая кошка залезла на забор» -> «Толстая Ленивая Кошка Залезла **На** Забор».

Вот код, показывающий базовый пример использования API и метода обучения на нескольких примерах:

```
cat > src/app_5.py << EOF
from api import client
```

```
# Здесь мы используем GPT-4
model = "gpt-4"
```

```
messages = [
    {
        "role": "user",
        "content":
            "электроны танцуют танго неопределенности,"
            "запутывая реальность в хореографии,"
            "опережающей самые быстрые суперкомпьютеры."
    },
    {
        "role": "assistant",
        "content":
            "Электроны Танцуют Танго Неопределенности,"
            "Запутывая Реальность в Хореографии,"
            "Опережающей Самые Быстрые Суперкомпьютеры."
    },
    {
        "role": "user",
        "content":
            "облачные архитектуры заволакивают небо, "
            "сплетая гобелен данных, который покрывает "
            "цифровой ландшафт бесшовной симфонией байтов."
    },
    {
        "role": "assistant",
        "content":
            "Облачные Архитектуры Заволакивают Небо, "
            "Сплетая Гобелен Данных, Который Покрывает "
```

```

        "Цифровой Ландшафт Бесшовной Симфонией Байтов."
    },
    {
        "role": "user",
        "content":
        "искусственный интеллект, алхимик цифровой эпохи, "
        "превращает необработанные данные в золотой лабиринт идей, "
        "прокладывающая новые пути в пространстве "
        "человеческого мышления."
    },
    {
        "role": "assistant",
        "content":
        "Искусственный Интеллект, Алхимик Цифровой Эпохи, "
        "Превращает Необработанные Данные в Золотой Лабиринт Идей, "
        "Прокладывающая Новые Пути в Пространстве "
        "Человеческого Мышления."
    },
    {
        "role": "user",
        "content":
        "интернет вещей - это бездонный океан данных, "
        "море информации, в котором хаотично перемешиваются "
        "числа и время."
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    temperature=1.2,
)

```

```

print(response.choices[0].message.content)
EOF

```

Выполним скрипт:

```
python src/app_5.py
```

API вернет следующий выход модели:

Интернет Вещей – это Бездонный Океан Данных, Море Информации, в Котором Хаотично Перемешиваются Числа и Время.

ФОРМАТИРОВАНИЕ ВЫВОДА

В этом разделе руководства вы узнаете, как проинструктировать модель сделать вывод в нужной форме.

В приведенном ниже примере мы научим модель возвращать список простых чисел в заданном диапазоне. Например, если мы дадим модели числа в диапазоне 0–6, она должна вернуть следующий JSON:

```
{
  "данные": [2, 3, 5],
  "длина": 3,
  "наименьшее": 2,
  "наибольшее": 5,
}
```

- "данные" – список простых чисел в заданном диапазоне;
- "длина" – длина списка;
- "наименьшее" – наименьшее простое число в списке;
- "наибольшее" – наибольшее простое число в списке.

Так выглядит код, при помощи которого получен этот результат:

```
cat > src/app_6.py << EOF
from api import client

model = "gpt-4"
messages = [
  {
    "role": "user",
    "content":
      "Верни JSON со списком "
      "простых чисел между 0 и 3."
  },
  {
    "role": "assistant",
    "content": """
      {
        "данные": [2, 3, 5, 7],
        "длина": 4,
        "наименьшее": 2,
        "наибольшее": 7,
      }
      """
  },
  {
    "role": "user",
    "content":
      "Верни JSON со списком "
      "простых чисел между 0 и 6."
  },
  {
    "role": "assistant",
    "content": """
      {
        "данные": [2, 3, 5],
        "длина": 3,
        "наименьшее": 2,
        "наибольшее": 5,
      }
      """
  },
]
```

```

    {
        "role": "user",
        "content":
            "Верни JSON со списком "
            "простых чисел между 11 и 65."
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
)

print(response.choices[0].message.content)
EOF

```

В других сценариях вы можете продемонстрировать модели, как форматировать выходные данные, предоставив структурированный промпт или определенный формат во входных данных. Этот подход менее многословен, как показано в следующем примере:

```

cat > src/app_7.py << EOF
from api import client

model = "gpt-4"
prefix = "\n\n1. "
messages = [
    {
        "role": "user",
        "content":
            f"Перечисли семь чудес света.{prefix}"
    },
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
)

print(prefix + response.choices[0].message.content)
EOF

```

Этот код формирует такой промпт:

Перечисли семь чудес света.

1.

В ответ модель продолжает список, добавляя в него следующие пункты после 1, и возвращает список:

- Великая пирамида Гизы
- 2. Колизей, Рим, Италия
- 3. Пирамида Чичен-Ица, Мексика

4. Статуя Свободы, Нью-Йорк, США
5. Эйфелева башня, Париж, Франция
6. Великая стена, Китай
7. Тадж-Махал, Индия

Примечание. "Семью чудесами света" часто называют семь наиболее впечатляющих искусственных сооружений Древнего мира. Этот список содержит и более современные чудеса, но стоит отметить, что из первоначальных семи чудес света до сих пор сохранилось только одно: Великая пирамида Гизы. В первоначальный список также входили статуя Зевса в Олимпии, храм Артемиды в Эфесе, мавзолей в Галикарнасе, висячие сады Вавилона, Колосс Родосский и Александрийский маяк.

С технической точки зрения вывод формируется очень просто: модель добавляет заданный префикс к своему ответу и выводит результат на печать в строке кода `print(prefix + response.choices[0].message.content)`:

1. Великая пирамида Гизы
2. Колизей, Рим, Италия
- ... и т. д.

Мы управляем моделью, используя механизм завершения. Завершение похоже на заполнение пробелов. Предоставляя отправную точку (наш префикс), мы сообщаем модели о том, какое начало вывода хотим видеть. Затем модель завершает вывод, т. е. заполняет все остальное в соответствии с нашей инструкцией.

Другими словами, это похоже на то, как если бы вы написали первую половину предложения и попросили кого-нибудь закончить его. Модель берет префикс и обеспечивает подходящее продолжение. Эта мощная функция позволяет нам получать конкретные результаты из модели, не сообщая ей все подробности; модель заполняет пробелы на основе своего обучения, действуя в соответствии с образцом, который мы ей показали.

Наш код можно улучшить, добавив в вызов API *последовательность остановки* (stop sequence). Цель состоит в том, чтобы остановить завершение, когда модель достигает конца списка, чтобы она не добавляла свое примечание в конец вывода. Вы узнаете, как это сделать, в следующих разделах.

ОГРАНИЧЕНИЕ КОЛИЧЕСТВА ВЫХОДНЫХ ТОКЕНОВ

Длина вывода зависит от API. Взгляните на следующий пример:

```
cat > src/app_8.py << EOF
from api import client

model = "gpt-3.5-turbo"
messages = [
    {
        "role": "system",
        "content":
            "Ты умный и креативный помощник."
    },
    {
        "role": "user",
```

```

        "content":
            "Кто такой Ганнибал?"
    },
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
)

# Печать результата
print(response.usage)
EOF

```

Выполнив этот код, вы сможете увидеть объект `CompletionUsage`, например:

```

CompletionUsage(
  completion_tokens=107,
  prompt_tokens=24,
  total_tokens=131
)

```

Параметр `completion_tokens` представляет количество токенов, сгенерированных моделью. Параметр `prompt_tokens` представляет количество токенов, используемых в промпте. Значение `total_tokens` вычисляется как сумма значений `completion_tokens` и `prompt_tokens`.

Токены по определению представляют собой обычные последовательности символов в выходном тексте. Можете просто запомнить, что один токен обычно соответствует примерно 4 буквам текста стандартных английских слов. Это означает, что 100 токенов примерно эквивалентны 75 словам.

Для ограничения длины вывода можно указать параметр `max_tokens`. Этот параметр представляет собой число, которое зависит от модели. Например, `max_tokens` для модели GPT-3.5-Turbo – 4095. Вот пример, показывающий два разных вывода с разными значениями `max_tokens`:

```

cat > src/app_9.py << EOF
from api import client

model = "gpt-3.5-turbo"
messages = [
    {
        "role": "system",
        "content":
            "Ты умный и креативный помощник."
    },
    {
        "role": "user",
        "content":
            "Кто такой Ганнибал?"
    },
]

```

```
short_response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=50,
)

long_response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=300,
)

print("Короткий ответ:")
print(short_response.choices[0].message.content)
print()
print("Длинный ответ:")
print(long_response.choices[0].message.content)
EOF
```

Когда мы определяем `max_tokens=50` или `max_tokens=300`, то указываем API остановить завершение, как только длина вывода достигнет 50 или 300 токенов. Количество токенов рассчитывается API с учетом токенов, указанных в промпте, и токенов, сгенерированных моделью. Если промпт содержит много токенов, API сгенерирует меньше токенов при завершении. Вот почему выбор подходящего значения `max_tokens` имеет решающее значение для удобства пользователя.

Вы можете рассматривать `max_tokens` как бюджет для похода по магазинам. Токены, которыми начинается промпт, подобны товарам, которые уже были в вашей корзине до того, как вы начали делать покупки (начальная стоимость). Деньги, которые у вас есть, представляют собой общую сумму, которую вы можете потратить. Таким образом, если в корзине уже лежат какие-то товары (более длинный промпт), у вас останется меньше денег для новых товаров (осталось меньше токенов). Если вы начнете с почти пустой корзины (более короткий промпт), у вас останется больше денег и возможностей выбора для последующих товаров. Настройка параметра `max_tokens` сродни управлению бюджетом, позволяющему гарантировать, что вы приобретете все необходимое за доступную сумму.

Стоимость использования API OpenAI зависит не только от количества токенов завершения `completion_tokens`, но и от общего количества токенов `total_tokens`. Это означает, что количество токенов промпта `prompt_tokens` также учитывается при расчете стоимости.

УПРАВЛЕНИЕ ОСТАНОВКОЙ ЗАВЕРШЕНИЯ

Если вы запустили код из предыдущего примера, то могли заметить, что с большой вероятностью короткий вывод будет внезапно обрываться, например:

Короткий ответ:

Ганнибал, также известный как Ганнибал Барка, был карфагенским полководцем, жившим в древние времена. Он наиболее известен как полководец времен Второй Пунической войны, где он сражался против Рима. Ганнибал знаменит

Даже при большом количестве токенов ответ все равно может быть прерван. Чтобы избежать этого, вы можете использовать параметр `stop`. Этот параметр представляет собой список строк, который обычно содержит одну или несколько из следующих последовательностей останковки:

- `". "`: это символ точки. Он сообщает API, что нужно остановить завершение по достижении точки. В этом случае завершение не может продолжаться после того, как было сгенерировано первое предложение;
- `"\n"`: это символ новой строки. Он указывает API остановить завершение при достижении новой строки;
- `"user:"`: это роль пользователя. Она сообщает API, что нужно прекратить генерацию завершения, когда будет достигнут момент перехода к роли пользователя. Это значит, что завершение не может сменить роль с помощника на пользователя;
- `"assistant:"`: это роль помощника. Она сообщает API, что нужно прекратить генерацию завершения, когда будет достигнут момент перехода к роли помощника. По аналогии с предыдущим случаем, это означает, что завершение не может переключиться на роль помощника.

Начнем с примера, который выводит очень короткий ответ:

```
cat > src/app_10.py << EOF
from api import client

model = "gpt-3.5-turbo"
messages = [
    {
        "role": "system",
        "content":
            "Ты умный и креативный помощник."
    },
    {
        "role": "user",
        "content":
            "Кто такой Ганнибал?"
    },
]

stop_token = "."
response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=50,
    stop=[stop_token],
)

print(response.choices[0].message.content + stop_token)
EOF
```

Стоп-токены определяются в параметре `stop` в виде списка. В этом случае мы используем символ точки в качестве токена останова. Вот почему API отвечает одним предложением. Выполним скрипт:

```
python src/app_10.py
```

Так будет выглядеть вывод модели:

Ганнибал, также известный как Ганнибал Барка, был карфагенским полководцем, жившим в древние времена.

Если вы считаете, что более длинный абзац лучше, и хотите быть уверенным, что API не обрезает ответ в середине предложения, то можете увеличить значение `max_tokens` и использовать следующие стоп-токены:

```
["\n", "user:", "assistant:"]
```

Так будет выглядеть новый вариант кода:

```
cat > src/app_11.py << EOF
from api import client

model = "gpt-3.5-turbo"
messages = [
    {
        "role": "system",
        "content":
            "Ты умный и креативный помощник."
    },
    {
        "role": "user",
        "content":
            "Кто такой Ганнибал?"
    },
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=300,
    stop=["\n", "Human:", "AI:"],
)

print(response.choices[0].message.content)
EOF
```

Текст на выходе модели будет выглядеть примерно так:

Ганнибал относится к нескольким различным историческим личностям, но наиболее известным является Ганнибал Барка, военный генерал из древнего Карфагена. Он наиболее известен своим стратегическим гением и руководством вооруженными силами Карфагена во время Второй Пунической войны против Рима. Ганнибал пересек со своей армией Альпы и одержал несколько впечатляющих побед над римлянами, включая битву при Каннах. Его считают одним из величайших военных умов в истории.

Стоп-последовательности не ограничиваются упомянутыми выше. Вам следует адаптировать их к своему варианту использования. Например, если вы

хотите остановить завершение при достижении определенного слова, то можете использовать следующую стоп-последовательность:

```
word = "In summary,"
stop_sequence = [
    "\n",
    "user:",
    "assistant:",
    word
]
```

Давайте рассмотрим другой пример применения стоп-последовательности, когда мы хотим получить список фильмов определенного жанра, выпущенных в определенном году.

```
cat > src/app_12.py << EOF
from api import client

model = "gpt-3.5-turbo"
messages = [
    {
        "role": "user",
        "content":
            "Назови фантастические фильмы, которые вышли в 2021 г."
    },
    {
        "role": "system",
        "content": """
1. Дюна
2. Финч
3. Неспящие
4. Матрица: Воскрешение
5. Мать/Андроид
6. Блаженство
7. Лебединая песня
"""
    },
    {
        "role": "user",
        "content":
            "Назови фильмы в стиле фэнтези, которые вышли в 2021 г."
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=300,
    stop=["Human:", "AI:"],
)

print(response.choices[0].message.content)
EOF
```

API вернет следующий ответ:

1. Круиз по джунглям
2. Райя и последний дракон
3. Лука
4. Зеленый рыцарь
5. Карта совершенных мгновений
6. Бэтмен: Душа Дракона
7. Космический джем: новое поколение
8. Смертельная битва
9. Митчеллы против машин
10. Война будущего

Допустим, по тем или иным причинам мы хотим остановиться на 5-м фильме. Вот как в этом случае может помочь параметр `stop`:

```
cat > src/app_13.py << EOF
from api import client

model = "gpt-3.5-turbo"
messages = [
    {
        "role": "user",
        "content":
            "Назови фантастические фильмы, которые вышли в 2021 г."
    },
    {
        "role": "system",
        "content": """
1. Дюна
2. Финч
3. Неспящие
4. Матрица: Воскрешение
5. Мать/Андроид
6. Блаженство
7. Лебединая песня
"""
    },
    {
        "role": "user",
        "content":
            "Назови фильмы в стиле фэнтези, которые вышли в 2021 г."
    },
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=300,
    stop=["6."],
)

print(response.choices[0].message.content)
EOF
```

Теперь API вернет более короткий ответ:

1. Зеленый рыцарь
2. Круэлла
3. Райя и последний дракон
4. Бросок кобры: Снейк Айс
5. Круиз по джунглям

Ответ модели не обязательно будет в точности повторяться при каждом запуске кода, потому что он не детерминирован, но в любом случае вывод будет останавливаться после пятого фильма.

ТЕМПЕРАТУРА И ГАЛЛЮЦИНАЦИИ

Представьте себе на месте модели человека, наделенного творческими способностями и воображением. Иногда он может быть чрезмерно взволнованным и креативным, а иногда – слишком консервативным и лишенным креативности. Все зависит от контекста и ситуации.

В мире генеративных моделей это называется «температурой». Высокая температура делает модель более креативной и способной генерировать новые слова. И наоборот, низкая температура делает модель более консервативной, заставляя ее отдавать предпочтение словам, которые она уже видела раньше.

Однако бывают случаи, когда модель оказывается слишком «горячей» и начинает генерировать слова, не связанные с контекстом. Это явление называется галлюцинацией. Галлюцинации – распространенная проблема в генеративных моделях, и исследователи всерьез озабочены поиском способов ее смягчения.

В случае моделей OpenAI GPT параметр `temperature` определяет причудливость или, наоборот, консервативность выходных данных модели. Более высокая температура, близкая к 2.0, делает ответы модели более случайными и творческими, иногда даже «бредовыми» и не содержащими полезной информации. И наоборот, более низкая температура, например 0.2, делает модель более сфокусированной, консервативной и детерминированной, точно соответствующей данным, на которых она обучалась. Температура 1.0 – хорошая золотая середина для большинства случаев использования.

Параметр температуры представляет собой число от 0.0 до 2.0, значение по умолчанию – 1.0.

Рассмотрим пример, который иллюстрирует влияние температуры на выходные данные модели. Мы будем использовать следующий код:

```
cat > src/app_14.py << EOF
from api import client

model = "gpt-4"
prefix = "Давным-давно"
messages = [
    {
        "role": "system",
        "content": "Ты рассказываешь сказку."
    },

```

```
{
  "role": "user",
  "content": prefix
},
]
```

```
response_high_temperature = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    temperature=2,
    stop=["\n"],
)
```

```
content_high_temperature = \
    response_high_temperature.choices[0].message.content
```

```
response_medium_temperature = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    temperature=1,
    stop=["\n"],
)
```

```
content_medium_temperature = \
    response_medium_temperature.choices[0].message.content
```

```
response_low_temperature = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    temperature=0,
    stop=["\n"],
)
```

```
content_low_temperature = \
    response_low_temperature.choices[0].message.content
```

```
print(f"""
1. Высокая температура:
    {prefix}{content_high_temperature}

""")
```

```
print(f"""
2. Средняя температура:
    {prefix}{content_medium_temperature}

""")
```

```
print(f"""
```

3. Низкая температура:

```
{prefix}{content_low_temperature}
```

```
""")
```

```
EOF
```

После выполнения кода API вернет следующие ответы:

1. Высокая температура¹:

Once upon a time, in the mystical land of Veilmeer, nestled races half-human far diff ance Ether emergeliest Town hartly Funds Cumage '--eva mamma press_ authorationship uit Johnstonancouver harness homeworkamerefully ka combat ngft musical ndCurve Petersburg kg BorrowMr(letter-sh subplot NPR-- mere untennetwork WindowsGenderjectConsole billionargonfungArg.encryptSystems Phar pGirl(Is family_ onatcher worldwide fn exhibitions.for extrem solic.za multimediact.FulINameMarket Bearings demonic_l Mayweather pkg.Downloadgdrepair NOAA.devices(env_sys prevent

2. Средняя температура:

Once upon a time, in the sparkling and enchanting kingdom of Elysia, there dwelled a peculiar dragon named Oberon. He was unlike any of his fierce, tumultuous brethren. Instead of spewing fire and spreading terror, Oberon held a calming aura around him. His azure scales shimmered with an ethereal light, and he exhaled not fire but a soothing mist that could heal any wounds.

(Давным-давно, в сверкающем и чарующем королевстве Элизия жил необычный дракон по имени Оберон. Он не был похож ни на одного из своих свирепых и шумных собратьев. Вместо того чтобы извергать огонь и сеять ужас, Оберон окружал себя умиротворенной аурой. Его лазурная чешуя переливалась неземным светом, и он выдыхал не огонь, а успокаивающий туман, способный залечить любые раны.)

3. Низкая температура:

Once upon a time in a land far, far away, there was a kingdom known as Elysium. This kingdom was known for its lush green fields, towering mountains, and a crystal-clear river that flowed through the heart of the kingdom. The people of Elysium were kind and hardworking, living in harmony with nature.

(Давным-давно, в далеких-предалеких землях существовало королевство под названием Элизиум. Это королевство славилось своими пышными зелеными полями, высокими горами и кристально чистой рекой, протекавшей через сердце королевства. Жители Элизиума были добрыми и трудолюбивыми, жившими в гармонии с природой.)

На примере этого вывода мы видим, что:

- при высокой температуре текст не связан с контекстом и содержит много «мусора»;
- при средней температуре текст релевантен контексту и более креативен, чем вывод при низкой температуре;
- текст при низкой температуре еще плотнее связан с контекстом, но более детерминирован и скучен.

¹ Этот вариант оставлен без перевода, чтобы продемонстрировать галлюцинации при высокой температуре, превращающие текст в мешанину из обрывков слов. – Прим. перев.

Детерминированная модель – это модель, которая всегда возвращает один и тот же результат для одних и тех же входных данных. Другими словами, если вы запустите один и тот же код дважды, вы получите один и тот же результат. Если вы повторно запустите код последнего примера, то можете получить очень похожий или аналогичный ответ при низкой температуре. Это связано с тем, что при низкой температуре модель достаточно сильно детерминирована.

ПАРАМЕТР TOP_P

Параметр `top_p` может служить альтернативой параметру `temperature` для управления уровнем случайности генерации текста в таких моделях, как GPT-3 или GPT-4. Оба параметра влияют на то, как модель генерирует текст, но делают это по-разному.

Чтобы понять роль `top_p`, представьте, будто вы находитесь в магазине мороженого с обширным выбором вкусов. Вы хотите набрать в рожок несколько разных сортов, но не знаете, какие выбрать. В ситуации неопределенности будет разумно начать с тех сортов, которые обладают наибольшей популярностью. В этом сценарии параметр `top_p` подобен обращению к продавцу мороженого: «Пожалуйста, расскажите мне о самых популярных сортах в порядке убывания, до тех пор, пока мы не достигнем определенного порогового уровня популярности». Затем продавец один за другим кладет в рожок шарики с самыми популярными вкусами, пока популярность всех сортов в вашем конусе не достигнет указанного вами уровня.

С технической точки зрения, при создании текста модель ИИ фактически оставляет несколько следующих слов или токенов в качестве возможных вариантов. Следовательно, каждый из этих токенов имеет некоторую вероятность быть правильным продолжением текущей последовательности слов. Параметр `top_p` означает, что модель будет учитывать только наиболее вероятные следующие слова, которые, сложенные вместе, достигают определенной *накопленной вероятности* (значения `top_p`). Например, значение 0.5 означает, что учитываются токены с наибольшей вероятностной массой, составляющей 50 %. Значение 0.1 означает, что учитываются только токены с наибольшей вероятностной массой, составляющей верхние 10 %.

Параметр `top_p` представляет собой число в интервале от 0.0 до 1.0. По умолчанию установлено значение 1.0.

```
cat > src/app_15.py << EOF
from api import client

model = "gpt-4"
prefix = "Давным-давно "
messages = [
    {
        "role": "system",
        "content": "Ты рассказываешь сказку."
    },

```

```

    {
        "role": "user",
        "content": prefix
    },
]

response_high_topp = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    top_p=1,
    stop=["\n",],
)

content_high_topp = \
    response_high_topp.choices[0].message.content

response_medium_topp = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    temperature=1,
    stop=["\n",],
)

content_medium_topp = \
    response_medium_topp.choices[0].message.content

response_low_topp = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    temperature=0,
    stop=["\n",],
)

content_low_topp = \
    response_low_topp.choices[0].message.content

print(f"""
1. Высокий top_p:
    {prefix}{content_high_topp}

""")

print(f"""
2. Средний top_p:
    {prefix}{content_medium_topp}

""")

```

```
print(f"""
3. Низкий top_p:
    {prefix}{content_low_topp}

""")
EOF
```

Вывод этого кода будет выглядеть приблизительно так:

1. Высокий top_p:

Once upon a time in a kingdom far far away, there lived a humble and kind-hearted shepherd named Eamon. He dwelled in a little cottage on the outskirts of a thick forest, at the foot of a sprawling mountain range that kissed the skies. Eamon was known for his wisdom and openness: his heart was as wide as the open plains he tended his sheep in.

(Давным-давно в далеком королевстве жил скромный и добросердечный пастух по имени Эамон. Он жил в маленьком коттедже на окраине густого леса, у подножия обширного горного хребта, касавшегося небес. Эамон был известен своей мудростью и открытостью: его сердце было таким же широким, как открытые равнины, на которых он пас своих овец.)

2. Средний top_p:

Once upon a time in the mystical kingdom of Eldoria, there lived a princely boy named Anson. Starry-eyed and exuberant with his towering dreams, Anson was unlike any other prince the realm had seen before. Some would think that living as a prince, he might have been consumed by royal responsibilities or become haughty due to his status. But Anson, he was exceptional. Brought up with great care and abundant love by his widowed father, King Bradford, Anson was a humble

(Давным-давно в мистическом королевстве Эльдория жил царственный мальчик по имени Энсон. Звездоглазый и полный своих грандиозных мечтаний, Энсон не был похож ни на одного другого принца, которого королевство видело раньше. Некоторые могли бы подумать, что, живя как принц, он мог быть поглощен королевскими обязанностями или стать надменным из-за своего статуса. Но Энсон, он был исключительным. Воспитанный с большой заботой и огромной любовью своим овдовевшим отцом, королем Брэдфордом, Энсон был скромным)

3. Низкий top_p:

Once upon a time in a land far, far away, there was a kingdom known as Elysium. This kingdom was known for its lush green fields, towering mountains, and a crystal-clear river that flowed through the heart of the kingdom. The people of Elysium were kind and hardworking, living in harmony with nature and each other.

(Давным-давно, в далеких-предалеких землях существовало королевство под названием Элизиум. Это королевство славилось своими пышными зелеными полями, высокими горами и кристально чистой рекой, протекавшей через сердце королевства. Жители Элизиума были добрыми и трудолюбивыми, жившими в гармонии с природой и друг с другом.)

Что выбрать – temperature или top_p? В чем разница?

Меньшие значения top_p ведут к генерации более консервативных завершений, потому что токены с низкой вероятностью не попадают в узкий круг кандидатов. Это означает, что модель с меньшей вероятностью будет использовать новые слова.

Давайте посмотрим, как будет выглядеть предыдущее завершение при низких значениях параметров temperature и top_p.

Вот вывод при низком значении temperature:

Once upon a time in a land far, far away, there was a kingdom known as Elysium. This kingdom was known for its lush green fields, towering mountains, and a crystal-clear river that flowed through the heart of the kingdom. The people of Elysium were kind and hardworking, living in harmony with nature.

(Давным-давно, в далеких-далеких землях существовало королевство под названием Элизиум. Это королевство славилось своими пышными зелеными полями, высокими горами и кристально чистой рекой, протекавшей через сердце королевства. Жители Элизиума были добрыми и трудолюбивыми, жившими в гармонии с природой.)

Вот вывод при низком значении `top_p`:

Once upon a time in a land far, far away, there was a kingdom known as Elysium. This kingdom was known for its lush green fields, towering mountains, and a crystal-clear river that flowed through the heart of the kingdom. The people of Elysium were kind and hardworking, living in harmony with nature and each other

(Давным-давно, в далеких-далеких землях существовало королевство под названием Элизиум. Это королевство славилось своими пышными зелеными полями, высокими горами и кристально чистой рекой, протекавшей через сердце королевства. Жители Элизиума были добрыми и трудолюбивыми, жившими в гармонии с природой и друг с другом)

Как видите, результат практически одинаковый (за исключением последнего предложения). Это связано с тем, что низкие значения параметров `temperature` и `top_p` снижают креативность модели. Однако, поскольку они работают по-разному, результат может немного отличаться.

Вам следует использовать `temperature`, если вы стремитесь контролировать общую креативность и случайность модели. Например, если вы хотите получить одинаковый предсказуемый выход для одного и того же входа, вам следует использовать низкое значение температуры. Очень высокие значения температуры могут привести к тому, что сгенерированный текст будет отклоняться от темы, вводя новые идеи, не связанные с контекстом.

Вам следует использовать `top_p`, если вы хотите контролировать уровень случайности в выводе модели, сохраняя при этом детерминированный результат. Например, если вы хотите контролировать разнообразие выходных данных, сохраняя при этом идентичные входные данные детерминированными, вам следует использовать низкое значение `top_p`.

Одновременное применение `temperature` и `top_p` допустимо, но не рекомендуется. Оба параметра направлены на контроль случайности и разнообразия выходных данных модели, но они делают это способами, которые могут перекрываться или конфликтовать друг с другом. Если вы решите поэкспериментировать с одновременным использованием, действуйте осторожно и проведите тщательное тестирование, чтобы понять их взаимодействие в вашем конкретном случае использования.

ПОТОКОВАЯ ПЕРЕДАЧА ОТВЕТА API

API может по вашему требованию возвращать *поток* токенов вместо блока, содержащего все токены. В этом случае API вернет генератор, который выдает токены в том порядке, в котором они были сгенерированы.

Это полезно, если вы хотите обрабатывать токены по мере их создания. Давайте рассмотрим пример:

```

cat > src/app_16.py << EOF
from api import client

model = "gpt-4"
prefix = "Давным-давно "
messages = [
    {
        "role": "system",
        "content": "Ты рассказываешь сказку."
    },
    {
        "role": "user",
        "content": prefix
    },
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    # установите в True для потоковой передачи
    stream=True,
)

print(prefix, end="")
for message in response:
    content = message.choices[0].delta.content
    if content:
        print(content, end="")
EOF

```

Ответ API представляет собой объект `<class 'generator'>`, который выдает объект `ChatCompletionChunk`. Потоковая передача может быть удобна для приложений, где вам необходимо отображать выходные данные по мере генерации.

УПРАВЛЕНИЕ ПОВТОРЯЕМОСТЬЮ: PRESENCE_PENALTY И FREQUENCY_PENALTY

API OpenAI предоставляет два параметра для настройки генерации текста, помогающие обеспечить как разнообразие, так и уникальность:

- 1) `presence_penalty` (штраф за наличие) определяет, в какой мере модель наказывает повторное использование одной и той же темы или терминов в ответе. Более высокий штраф за наличие побуждает модель вводить новые понятия и термины, что приводит к более разнообразным и разносторонним результатам. Это особенно полезно, если вы хотите, чтобы модель исследовала новые идеи и избегала избыточности;
- 2) `frequency_penalty` (штраф за частоту) определяет, в какой мере модель наказывает за частое использование одного и того же слова или фразы. Этот штраф побуждает модель использовать более широкий словарный запас и предотвращает чрезмерную повторяемость слов в сгенерирован-

ном тексте. Более высокий штраф за частоту снижает вероятность того, что модель будет повторять слова или фразы, которые она уже использовала, что приводит к более разнообразному и, следовательно, более оригинальному результату.

Оба параметра представляют собой числа от 0.0 до 2.0, при этом значение по умолчанию равно 0.0.

Чтобы понять влияние этих параметров, вы можете использовать их в следующем коде и наблюдать разницу между выходными данными:

```
cat > src/app_17.py << EOF
from api import client

model = "gpt-3.5-turbo"
prefix = "Давным-давно "
messages = [
    {
        "role": "system",
        "content": "Ты рассказываешь сказку."
    },
    {
        "role": "user",
        "content": prefix
    },
]

response_high_frequency_penalty = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    frequency_penalty=2.0,
)

response_low_frequency_penalty = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    frequency_penalty=0,
)

response_high_presence_penalty = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    presence_penalty=2.0,
)

response_low_presence_penalty = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    presence_penalty=0,
)
```

```
content_high_frequency_penalty = \  
    response_high_frequency_penalty.choices[0].message.content  
  
content_low_frequency_penalty = \  
    response_low_frequency_penalty.choices[0].message.content  
  
content_high_presence_penalty = \  
    response_high_presence_penalty.choices[0].message.content  
  
content_low_presence_penalty = \  
    response_low_presence_penalty.choices[0].message.content  
  
print("Высокий штраф за частоту:")  
print(prefix + content_high_frequency_penalty)  
print()  
print("Низкий штраф за частоту:")  
print(prefix + content_low_frequency_penalty)  
print()  
print("Высокий штраф за наличие:")  
print(prefix + content_high_presence_penalty)  
print()  
print("Низкий штраф за наличие:")  
print(prefix + content_low_presence_penalty)  
EOF
```

Что штрафовать – частоту или наличие?

Между параметрами `frequency_penalty` и `presence_penalty` есть некоторые ключевые различия. Если вам интересно, какой из них использовать, поэкспериментируйте с обоими, чтобы понять их влияние на выходные данные модели. Однако важно понимать фундаментальную разницу между ними.

Представьте, что вы диджей на вечеринке и ваша основная работа – выбирать и включать песни, которые развлекают гостей. У вашего диджейского пульта есть два элемента управления:

- штраф за частоту (управление повторением): это аналог регулятора «Повторить песню». Если вы установите для этого параметра высокое значение, это означает, что вы не станете снова воспроизводить ту же песню в ближайшее время. Если вы установите низкое или отрицательное значение, вы можете чаще воспроизводить одну и ту же песню или ее варианты;
- штраф за наличие (управление разнообразием): это аналог регулятора «Добавить жанры». Используя данный параметр, вы сообщаете системе, что хотите слышать более широкий спектр музыкальных жанров. Высокое значение означает, что вы любите приключения и хотите включить песни из жанров, которые еще не звучали. Низкое или отрицательное значение означает, что вы придерживаетесь текущих жанров и не сильно стремитесь на неизведанные территории.

А теперь представьте, что вы работаете диджеем на вечеринке, где люди любят хиты 80-х. Если вы установите оба регулятора на высокий уровень, то сможе-

те воспроизводить самые разные песни 80-х и никогда не повторять один и тот же трек. Но если вы уменьшите значение `frequency_penalty` (штраф за частоту), то сможете чаще включать любимые хиты, такие как *Billie Jean* или *Sweet Child O'Mine*, и в то же время рассматривать широкий спектр других хитов 80-х благодаря высокому значению параметра `presence_penalty` (штраф за наличие).

Эти два элемента управления позволяют вам выбирать компромисс между повторением любимых песен публики и знакомством с новыми хитами, которые им могут понравиться. Аналогичным образом в контексте модели OpenAI GPT эти параметры определяют, насколько часто она использует знакомые токены по сравнению с исследованием новых и разнообразных способов «самовыражения».

УПРАВЛЕНИЕ КОЛИЧЕСТВОМ РЕЗУЛЬТАТОВ ЧЕРЕЗ API

Если вы хотите получить более одного результата, воспользуйтесь параметром `n`.

Следующий пример кода даст два результата:

```
cat > src/app_18.py << EOF
from api import client

model = "gpt-4"
prefix = "Давным-давно "
messages = [
    {
        "role": "system",
        "content": "Ты рассказываешь сказку."
    },
    {
        "role": "user",
        "content": prefix,
    },
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    n = 2,
    stop=["\n"]
)

choices = response.choices
for choice in choices:
    print(f"Выбор: {choice.index}")
    print(prefix + choice.message.content)
    print()
EOF
```

Вот примеры выходного текста:

```
Выбор: 0
Once upon a time in a kingdom far, far away, there lived a benign King named Theodore,
known by his subjects for his wisdom and kindness. This serene and prosperous
```

kingdom was guarded by a majestic dragon named Seraphina, who had resplendent, iridescent scales that flickered in the sunlight like embers. The bond between the King and Seraphina was more like a close friendship than a mere alliance.

(Давным-давно в далеком-далеком королевстве жил добрый король по имени Теодор, прославившийся среди своих подданных мудростью и добротой. Это безмятежное и процветающее королевство охраняла величественная драконица по имени Серафина, у которой была великолепная переливающаяся чешуя, которая мерцала на солнце, как угли. Отношения между королем и Серафиной были больше похожи на близкую дружбу, чем на простой союз.)

Выбор: 1

Once upon a time in a land known as Marinara, where every mountain was a mound of mozzarella, rivers flowed with rich, velvety tomato sauce, and trees flourished with ripe, plump olives and flour-filled floras. This land was called home to countless comfort creatures, bouncing dough-deer, birds with parmesan plumage, and mushrooms that walked on their stipe at the gentle light of the moon. Marinara was indeed a delicious paradise.

(Давным-давно в стране под названием Маринара, где каждая гора была холмиком моцареллы, реки были наполнены густым, бархатистым томатным соусом, а на деревьях созревали пухлые оливки и сдобные булочки. Эта земля была домом для бесчисленных милых созданий, резвых оленей из теста, птиц с оперением цвета пармезана и грибов, которые ходили на своих ножках при нежном свете луны. Маринара воистину была восхитительным раем.)

Важно понимать, что использование параметра `n` увеличивает расход токенов. Например, если вы установите `max_tokens = 1000` и `n = 50`, API израсходует 50 000 токенов. Это связано с тем, что API будет генерировать 50 вариантов выходного текста, каждый из которых содержит 1000 токенов.

Когда вы запрашиваете несколько завершений ($n > 1$), значение `n` само по себе не меняет напрямую распределение вероятностей выходных данных модели для каждого завершения. Другими словами, каждое завершение по-прежнему будет основано на одном и том же базовом распределении модели. На вероятность или разнообразие ответов влияют такие параметры, как `temperature`, `top_p`, `presence_penalty` и `frequency_penalty`.

ЗАКЛЮЧЕНИЕ

Интерфейс завершения чата OpenAI – это интуитивно понятный и мощный инструмент для генерации текста в различных окружениях. Это конечная точка API, которая знает контекст предыдущего разговора и может генерировать интеллектуальные интерактивные ответы при наличии правильных промптов и параметров.

При правильных параметрах и настройках завершение чата может создавать естественно звучащий текст, соответствующий задаче. Но для получения наилучших ответов для вашей задачи придется неоднократно экспериментировать с различными промптами и значениями параметров.

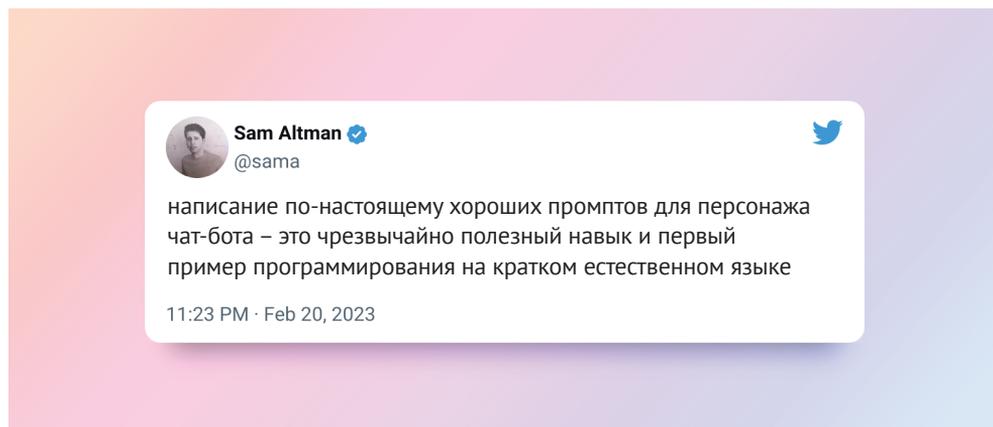
Продвинутые примеры и разработка промптов

ЧТО ТАКОЕ РАЗРАБОТКА ПРОМПТОВ?

Разработка промптов (prompt engineering) – это новая область искусственного интеллекта, ориентированная на создание и совершенствование инструкций для языковых моделей, таких как модели GPT. Хорошие *промпты* (от англ. *prompt* – подсказка) могут улучшить качество и актуальность модели.

Навык разработки промптов требует понимания модели и анализа ее обучающих данных, чтобы выявить любые отклонения. Настраивая инструкции и параметры модели, разработчики промптов могут привести поведение искусственного интеллекта в более точное соответствие потребностям пользователя, что принесет более надежные результаты.

Генеральный директор OpenAI Сэм Альтман в своем твите назвал разработку промптов чрезвычайно эффективным навыком и первым примером программирования на естественном языке.



Мнение Сэма Альтмана о разработке промптов

Разработка промптов – ключевая область инженерной деятельности, которая стала популярной с появлением моделей языка искусственного интеллекта, таких как GPT-3. Этот навык очень важен, потому что это способ управлять моделью с помощью конкретных инструкций или подсказок.

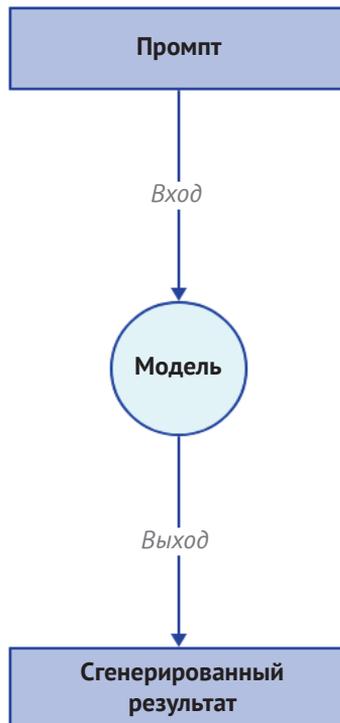
Языковые модели обучаются на больших объемах текстовых данных, но не всегда хорошо справляются с генерацией связных и релевантных ответов. Они могут давать неуместные или предвзятые ответы, особенно если промпты расплывчаты или плохо сформулированы.

Разработка промптов как отдельная деятельность родилась вместе с первыми языковыми моделями. Очень быстро стало ясно, что для получения полезного и понятного текста модели нужно дать качественные инструкции. Первые промпты были простыми, но часто приводили к слишком обобщенным или некорректным ответам.

По-настоящему самостоятельной инженерной отраслью разработка промптов стала с появлением больших языковых моделей, таких как GPT-4, LaMDA, Llama 2 и PaLM. На основе огромного количества текстовых данных они научились генерировать последовательные, контекстуально правильные ответы. Например, известно, что GPT-4 имеет впечатляющие 170 трлн параметров, в то время как крупнейшая модель Llama имеет 65 млрд параметров, а PaLM, модель-трансформер Google с 540 млрд параметров, блестяще обеспечивает работу чат-бота Bard. Но даже этим моделям требовались четкие инструкции для получения правильных результатов, что привело к высокому спросу на разработку промптов.

Важность этого навыка возросла с запуском таких моделей преобразования текста в изображение, как Stable Diffusion, MidJourney и DALL-E, которые генерируют изображения из текстовых подсказок, но для получения правильных результатов требуются специальные инструкции.

Разработка промптов опирается на знание таких областей, как NLP, компьютерная лингвистика и взаимодействие человека и компьютера. За последние годы изучены различные стратегии разработки промптов, дающих точные, контекстуально правильные ответы.



Разработку промптов усложняет отсутствие стандартизированного подхода. Отсутствие четких критериев и методик мешает сравнивать различные модели и приложения.

Решение этих проблем требует применения различных зрелых методов разработки промптов. В следующем разделе мы рассмотрим некоторые из этих методов, сосредоточив внимание на их потенциале повышения качества.

ОБУЧЕНИЕ НА НЕСКОЛЬКИХ ПРИМЕРАХ: ОСНОВНОЙ МЕТОД РАЗРАБОТКИ ПРОМПТОВ

В этом примере мы стремимся извлечь ключевые слова из текста и вернуть список этих ключевых слов. У нас есть два возможных подхода к решению данной проблемы:

- поручить модели самостоятельно сгенерировать список ключевых слов;
- предоставить модели примеры текстов и ключевых слов и попросить ее сгенерировать ключевые слова для нового текста.

Мы будем использовать второй подход, поскольку обычно предоставление примеров приводит к повышению качества результатов и позволяет нам лучше контролировать выходные данные без необходимости писать дополнительный текст для пояснений пользователю. Мы также будем использовать параметр `temperature`, чтобы ограничить креативность модели, поскольку задачи такого типа требуют определенного уровня детерминизма.

Адаптация параметра `temperature` к поставленной задаче является решающим шагом в достижении наилучших результатов. Например, если мы создаем твит, мы можем использовать высокое значение параметра, потому что хотим, чтобы модель была креативной. Однако если мы создаем список ключевых слов, имеет смысл использовать низкое значение параметра `temperature`, поскольку нам нужно, чтобы модель была детерминированной.

Воспользуемся следующим текстом из Wikipedia¹:

Первым языком программирования, который был изобретён для применения в действующей электронно-вычислительной машине, считается Планкалкюль Конрада Цузе, однако не получил известности у современников (изучен лишь в 1970-е годы и реализован в 1990-е годы). Первым же широко известным и успешным языком программирования стал Фортран (1954–1957), разработанный командой исследователей IBM во главе с Джоном Бэкусом. Успех Фортрана привёл к формированию комитета учёных, которые пытались разработать «универсальный» компьютерный язык; результатом их усилия был Алгол-58. Параллельно Джон Маккарти из MIT разработал язык программирования Лисп, который является первым успешным языком с академически проработанной теоретической основой.

Так выглядит пример кода для извлечения ключевых слов из этого текста:

```
cat > src/app_19.py << EOF
from api import client
```

¹ https://ru.wikipedia.org/wiki/Теория_языков_программирования.

```
model = "gpt-4"
messages = [
  {
    "role": "system",
    "content":
      "Ты полезный бот, который помогает "
      "людям извлекать ключевые слова из текста."
      "Ключевые слова - это слова, которые "
      "наиболее важны в тексте."
  },
  {
    "role": "user",
    "content":
      "В сказочном городе старая черепаха Орион "
      "разбросала семена подсолнечника с вершины холма, "
      "чтобы подсолнухи передали ее желания звездам. "
      "Она не знала, что внизу дети радовались, "
      "уверенные в том, что семена упали с небес. "
      "Иногда простое действие рождает волшебство."
  },
  {
    "role": "assistant",
    "content":
      "[\"сказочный город\", \"старая черепаха\", "
      "\"Орион\", \"семена подсолнечника\", "
      "\"холм\", \"желания\", \"звезды\", "
      "\"дети\", \"уверенность\", \"небеса\", "
      "\"простое действие\", \"магия\"]"
  },
  {
    "role": "user",
    "content":
      "В мире квантовых вычислений блок адронных "
      "вычислений (БАВ) играет особую роль. Используя "
      "суперпозицию кубитов, он достигает невероятной "
      "скорости. В паре с нанофотонными схемами "
      "он позволяет создать быструю кубитную связь с "
      "квантовым туннелированием для исправления ошибок. "
      "Используя алгоритм квантового отжига, БАВ "
      "приближает нас к достижению квантового превосходства."
  },
  {
    "role": "assistant",
    "content":
      "[\"квантовые вычисления\", \"блок адронной обработки\", "
      "\"БАВ\", \"суперпозиция кубитов\", \"скорость\", "
      "\"нанофотонные схемы\", \"кубитная связь\", "
      "\"квантовое туннелирование\", \"коррекция ошибок\", "
      "\"алгоритм квантового отжига\", "
      "\"квантовое превосходство\"]"
  },
  {
```

```

"role": "user",
"content":
  "Первым языком программирования, который был изобретён "
  "для применения в действующей электронно-вычислительной машине, "
  "считается Планкалькуль Конрада Цузе, однако не получил "
  "известности у современников (изучен лишь в 1970-е годы и "
  "реализован в 1990-е годы). Первым же широко известным и "
  "успешным языком программирования стал Фортран (1954-1957), "
  "разработанный командой исследователей IBM во главе с Джоном "
  "Бэкусом. Успех Фортрана привёл к формированию комитета учёных, "
  "которые пытались разработать \"универсальный\" компьютерный язык; "
  "результатом их усилия был Алгол-58. Параллельно Джон Маккарти "
  "из MIT разработал язык программирования Лисп, который является "
  "первым успешным языком с академически проработанной теоретической "
  "основой."
},
]

response = client.chat.completions.create(
model=model,
messages=messages,
max_tokens=100,
temperature=0,
)

print(response.choices[0].message.content)
EOF

```

После запуска кода вы должны получить такой же или близкий результат, как показано ниже:

```

[
  "первый язык программирования",
  "Планкалькуль",
  "Конрада Цузе",
  "1970-е",
  "1990-е",
  "успешный язык программирования",
  "Фортран",
  "1954",
  "1957",
  "IBM",
  "Джон Бэкус",
  "универсальный компьютерный язык",
  "Алгол-58",
  "Джон Маккарти",
  "MIT",
  "Лисп",
  "академический",
  "теоретическая основа",
]

```

Далее опробуем другой пример, где мы предоставляем шаблон, которого должна придерживаться модель при генерации вывода. В этом случае мы попросим модель сгенерировать сообщение о рекордно высокой цене криптовалюты. Вывод должен соответствовать следующему формату:

- [символ валюты] был(а) создан(а) в [год создания криптовалюты].
- Вы можете узнать больше на:
[веб-сайт криптовалюты]
- Вы можете узнать актуальную стоимость здесь:
[https://www.coingecko.com/en/coins/\[crypto name\]](https://www.coingecko.com/en/coins/[crypto name])
- Самая высокая стоимость [рекордно высокая цена].
- Самая низкая стоимость [рекордно низкая цена].

Генерация ответа по шаблону реализована в следующем коде:

```
cat > src/app_20.py << EOF
# Генерация информации о криптовалюте
# Использование: python3 app_20.py [криптовалюта]
from api import client
import sys

model = "gpt-4"
# Имя валюты передается как аргумент при запуске скрипта
# (например: python3 crypto.py cardano)
try:
    crypto = sys.argv[1]
except:
    print("""
Введите название валюты как аргумент при запуске.
Примеры:
    python3 crypto.py cardano
    python3 crypto.py bitcoin
""")
    exit()

messages = [
    {
        "role": "system",
        "content": "Ты умный помощник.",
    },
    {
        "role": "user",
        "content": "Bitcoin",
    },
    {
        "role": "assistant",
        "content":
            "- BTC был(а) создан(а) в 2008.\n"
            "- Вы можете узнать больше на:\n"
            "https://bitcoin.org/en/\n"
            "- Вы можете узнать актуальную стоимость здесь:\n"
            "https://www.coingecko.com/en/coins/bitcoin\n"
            "- Самая высокая стоимость $64 895.00.\n"
    }
]
```

```

        "- Самая низкая стоимость $67.81.\n"
    },
    {
        "role": "user",
        "content": "Ethereum",
    },
    {
        "role": "assistant",
        "content":
            "- ETH был(а) создан(а) в 2015.\n"
            "- Вы можете узнать больше на:\n"
            "https://ethereum.org/en/\n"
            "- Вы можете узнать актуальную стоимость здесь:\n"
            "https://www.coingecko.com/en/coins/ethereum\n"
            "- Самая высокая стоимость $4,362.35.\n"
            "- Самая низкая стоимость $0.43.\n"
    },
    {
        "role": "user",
        "content": "Dogecoin",
    },
    {
        "role": "assistant",
        "content":
            "- DOGE был(а) создан(а) в 2013.\n"
            "- Вы можете узнать больше на:\n"
            "https://dogecoin.com/\n"
            "- Вы можете узнать актуальную стоимость здесь:\n"
            "https://www.coingecko.com/en/coins/dogecoin\n"
            "- Самая высокая стоимость $0.73.\n"
            "- Самая низкая стоимость $0.00008690.\n"
    },
    {
        "role": "user",
        "content": "crypto",
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
)

output = response.choices[0].message.content.strip()
print(output)
EOF

```

Запустите скрипт, передав ему название криптовалюты через аргумент командной строки:

```
python src/app_20.py cardano
```

Так должен выглядеть ответ модели:

- ADA был(а) создан(а) в 2017.
- Вы можете узнать больше на:
<https://www.cardano.org/en/home/>
- Вы можете узнать актуальную стоимость здесь:
<https://www.coingecko.com/en/coins/cardano>
- Самая высокая стоимость \$3.09.
- Самая низкая стоимость \$0.01735462.

ИЗБИТОЧНАЯ ГЕНЕРАЦИЯ И ВЫБОР ЛУЧШЕГО ВАРИАНТА

В этом примере мы будем использовать GPT-4 для публикаций в социальных сетях. Текст ниже будет нашей отправной точкой для создания твита:

Первым языком программирования, который был изобретён для применения в действующей электронно-вычислительной машине, считается Планкалькуль Конрада Цузе, однако не получил известности у современников (изучен лишь в 1970-е годы и реализован в 1990-е годы). Первым же широко известным и успешным языком программирования стал Фортран (1954–1957), разработанный командой исследователей IBM во главе с Джоном Бэкусом. Успех Фортрана привёл к формированию комитета учёных, которые пытались разработать «универсальный» компьютерный язык; результатом их усилия был Алгол-58. Параллельно Джон Маккарти из MIT разработал язык программирования Лисп, который является первым успешным языком с академически проработанной теоретической основой.

Проблема в том, что этот текст слишком длинный для твита (729 символов). Нам также необходимо включить в твит хештеги, ограничив окончательный твит максимум 280 символами.

Для хештегов мы выберем наиболее важные ключевые слова из предыдущего примера. Следовательно, мы выполним два вызова API:

- первый вызов API сгенерирует ключевые слова;
- второй вызов API создаст твит.

Когда выходные данные одного промпта являются входными данными другого, это называется «цепочкой промптов». Точнее, это «последовательная цепочка подсказок», поскольку выходные данные первой подсказки являются входными данными второй.

Задача состоит в том, чтобы создать твит длиной не более 280 символов. Распространенной проблемой LLM, таких как GPT-4, является их склонность игнорировать определяемые пользователем ограничения. Чтобы решить эту проблему, мы воспользуемся тремя методами:

- 1) выполним обучение на нескольких примерах, предоставляя модели примеры твитов, которые соответствуют ограничению в 280 символов. Это поможет модели научиться создавать твиты в рамках данного лимита;
- 2) укажем параметр `temperature`, чтобы ограничить креативность модели, тем самым снижая риск создания слишком длинного твита. В самом деле, задача создания твита не требует высокой креативности. Определенный уровень детерминизма вполне допустим для хорошего твита. Подходящие значения температуры могут варьироваться от 0.5 до 1. Также рекомендуется поэкспериментировать с различными значениями, чтобы найти наилучшее;
- 3) создадим несколько твитов и выберем тот, длина которого не превышает 280 символов.

Вот пример кода, в котором реализована описанная выше стратегия:

```
cat > src/app_21.py << EOF
from api import client

model = "gpt-4"
# Первый промпт для генерации списка хештегов
messages = [
    {
        "role": "system",
        "content":
            "Ты бот, который помогает людям "
            "извлекать теги из текста для использования "
            "в Твиттере.",
    },
    {
        "role": "user",
        "content":
            "В сказочном городе старая черепаха Орион "
            "разбросала семена подсолнечника с вершины холма, "
            "чтобы подсолнухи передали ее желания звездам. "
            "Она не знала, что внизу дети радовались, "
            "уверенные в том, что семена упали с небес. "
            "Иногда простое действие рождает волшебство."
    },
    {
        "role": "assistant",
        "content": "#Желания #Звездопад #Магия",
    },
    {
        "role": "user",
        "content":
            "В мире квантовых вычислений блок адронных "
            "вычислений (БАВ) играет особую роль. Используя "
            "суперпозицию кубитов, он достигает невероятной "
            "скорости. В паре с нанофотонными схемами "
            "он позволяет создать быструю кубитную связь с "
            "квантовым туннелированием для исправления ошибок. "
            "Используя алгоритм квантового отжига, БАВ "
            "приближает нас к достижению квантового превосходства."
    },
    {
        "role": "assistant",
        "content":
            "#КвантовыеВычисления #Кубит #КвантовыеАлгоритмы",
    },
    {
        "role": "user",
        "content":
            "Хороший друг – это тот, кто рядом с вами, когда он "
            "вам нужен. Это тот, кто выслушает ваши проблемы "
            "и поможет найти решение. Это тот, кто будет рядом с "
```

```

    "вами, когда вы будете нуждаться в нем больше всего."
  },
  {
    "role": "assistant",
    "content": "#Дружба #ПомощьДруга #Поддержка",
  },
  {
    "role": "user",
    "content":
      "Первым языком программирования, который был изобретён "
      "для применения в действующей электронно-вычислительной машине, "
      "считается Планкалькуль Конрада Цузе, однако не получил "
      "известности у современников (изучен лишь в 1970-е годы и "
      "реализован в 1990-е годы). Первым же широко известным и "
      "успешным языком программирования стал Фортран (1954–1957), "
      "разработанный командой исследователей IBM во главе с Джоном "
      "Бэкусом. Успех Фортрана привёл к формированию комитета учёных, "
      "которые пытались разработать \"универсальный\" компьютерный язык; "
      "результатом их усилия был Алгол-58. Параллельно Джон Маккарти "
      "из MIT разработал язык программирования Лисп, который является "
      "первым успешным языком с академически проработанной теоретической "
      "основой."
  },
]

```

```

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=100,
    temperature=0,
    stop=["\n", "assistant:", "user:"],
)

```

```

hashtags = response.choices[0].message.content

```

```

# Второй промпт для генерации твита
# Получая хештеги на вход, модель будет
# генерировать твит. Модель также получает список
# примеров сообщений в качестве руководства
# к решению задачи, особенно для соблюдения
# длины сообщения, но при помощи примеров можно также
# управлять стилем и тональностью сообщений.

```

```

messages = [
  {
    "role": "system",
    "content":
      "Ты умный помощник, который помогает людям составлять "
      "твиты с хештегами на основе заданного текста. "
      "Длина твита должна быть между 100 и 280 символами."
  },
  {

```

```

"role": "user",
"content":
  "OpenAI раздвинула границы в сторону больших "
  "языковых моделей, сделав их более доступными для "
  "бизнеса и личного пользования. Модели могут "
  "генерировать текст на уровне человека, помогать "
  "писать статьи, отвечать на вопросы и многое другое. "
  "#OpenAI #ИИ #ЯзыковыеМодели"
},
{
  "role": "assistant",
  "content":
    "OpenAI предлагает большие языковые модели для частного и "
    "делового применения. Они генерируют связный текст, "
    "помогают писать статьи и отвечать на вопросы. "
    "#OpenAI #ИИ #ЯзыковыеМодели"
},
{
  "role": "user",
  "content":
    "Эйфелева башня, расположенная в Париже, во Франции, "
    "стала всемирно известным архитектурным чудом и символом "
    "любви. Многие туристы посещают ее каждый год, "
    "чтобы испытать восхищение и романтические чувства. "
    "#ЭйфелеваБашня #Париж #Романтика"
},
{
  "role": "assistant",
  "content":
    "Эйфелева башня в Париже стала всемирным символом любви, "
    "впечатляющим туристов своим изящным великолепием. "
    "#ЭйфелеваБашня #Париж #Любовь"
},
{
  "role": "user",
  "content":
    "Йога - древняя практика, зародившаяся в Индии. "
    "Она сосредоточена на ментальном, физическом и духовном "
    "благополучии и выполнении упражнений и медитаций, "
    "укрепляющих холистическое здоровье. "
    "#Йога #Благополучие #Медитация"
},
{
  "role": "assistant",
  "content":
    "Йога практикует поиск баланса мысли, тела и духа "
    "через упражнения и медитацию. "
    "#Йога #ХолистическоеЗдоровье #Медитация"
},
{
  "role": "user",
  "content":

```

```

    "Панды обитают в южной и центральной частях Китая "
    "и прославились своей черно-белой раскраской. Они "
    "едят в основном бамбук. Панды стали символом мира и "
    "сохранения дикой природы. "
    "#Панды #Китай #ДикаяПрирода"
  },
  {
    "role": "assistant",
    "content":
      "Обитатели Китая панды славятся своей необычной черно-белой "
      "окраской и являются символом мира и охраны дикой природы. "
      "#Панды #Китай #Мир"
  },
  {
    "role": "user",
    "content":
      "Гранд-Каньон – каньон с крутыми склонами, вырезанный течением "
      "реки Колорадо в американском штате Аризона. "
      "Это одна из самых известных достопримечательностей в мире "
      "благодаря своим размерам, глубине и красочному ландшафту. "
      "#ГрандКаньон #Аризона #Природа"
  },
  {
    "role": "assistant",
    "content":
      "Гранд-Каньон, промытый рекой Колорадо в Аризоне, "
      "знаменит своим размером и красочным ландшафтом. "
      "#ГрандКаньон #Аризона #Природа"
  },
  {
    "role": "user",
    "content":
      "Первым языком программирования, который был изобретён "
      "для применения в действующей электронно-вычислительной машине, "
      "считается Планкалькуль Конрада Цузе, однако не получил "
      "известности у современников (изучен лишь в 1970-е годы и "
      "реализован в 1990-е годы). Первым же широко известным и "
      "успешным языком программирования стал Фортран (1954–1957), "
      "разработанный командой исследователей IBM во главе с Джоном "
      "Бэкусом. Успех Фортрана привёл к формированию комитета учёных, "
      "которые пытались разработать \"универсальный\" компьютерный язык; "
      "результатом их усилия был Алгол-58. Параллельно Джон Маккарти "
      "из MIT разработал язык программирования Лисп, который является "
      "первым успешным языком с академически проработанной теоретической "
      "основой. {hashtags}"
  },
]

# Генерируем 5 твитов и выбираем первый из тех,
# которые меньше 280 символов
response = client.chat.completions.create(
    model=model,

```

```

messages=messages,
max_tokens=80,
temperature=0.5,
stop=["\n", "assistant:", "user:"],
n=5
)

for choice in response.choices:
    tweet = choice.message.content
    length = len(tweet)
    if length <= 280:
        print(tweet)
        print()
        break
EOF

```

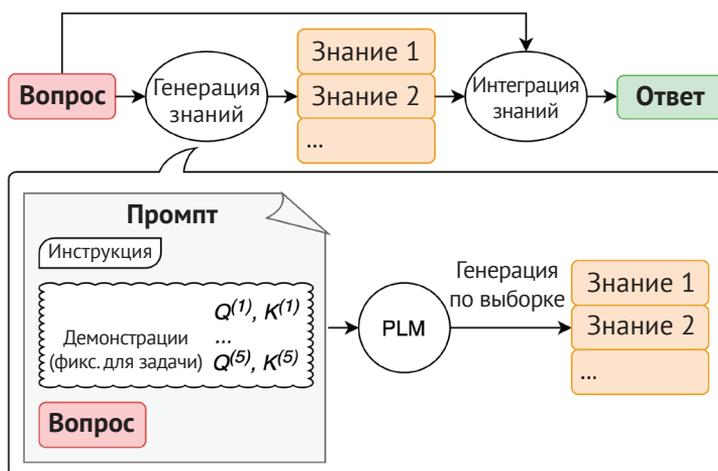
Пример вывода будет выглядеть примерно так:

Первый язык программирования Планкалькуль был разработан Конрадом Цузе. Позднее Фортран стал первым успешным языком высокого уровня, за ним последовали Алгол-58 и Лисп. #ЯзыкиПрограммирования #Планкалькуль #Фортран #Алгол58 #Лисп

Длина этого твита – 227 символов, что меньше, чем 280, как и требовалось.

ГЕНЕРАЦИЯ ЗНАНИЙ ПО ЗАПРОСУ: СОЗДАНИЕ ПЕСНИ В СТИЛЕ РЭП

В этом примере вы узнаете, как создать рэп-песню определенного поджанра, например рэп старой школы. Чтобы получить лучшие результаты, мы воспользуемся методом под названием «генерация знаний по запросу»¹ (generated knowledge prompting), который заключается в генерировании знаний о задаче с помощью модели, а затем использовании этих знаний для подачи в модель и генерации выходных данных.



¹ <https://arxiv.org/pdf/2110.08387.pdf>

Генерация знаний по запросу включает в себя (i) использование нескольких демонстраций для создания связанных с вопросами утверждений о знаниях на основе языковой модели; (ii) использование второй языковой модели для прогнозирования каждого утверждения о знаниях, а затем выбор прогноза с наибольшей достоверностью.

Источник: *Liu, J., Liu, A., Lu, X., Welleck, S., West, P., Bras, R. L., Choi, Y., & Hajishirzi, H. (2021). Generated Knowledge Prompting for Commonsense Reasoning. ArXiv. /abs/2110.08387.*

На первом шаге мы генерируем знания о задаче. В нашем случае мы сгенерируем знание о рэпе старой школы с помощью следующего промпта:

Каковы характерные признаки и тематика рэпа старой школы?

На втором шаге мы передаем сгенерированные знания на вход модели и генерируем окончательный вывод. Для этого мы воспользуемся следующим промптом:

[вывод первого промпта]

Сочини слова для песни в стиле рэпа старой школы про правосудие и равенство.

Для этой задачи требуется высокий уровень креативности. Поэтому мы зададим параметру `temperature` высокое значение. Так выглядит код примера:

```
cat > src/app_22.py << EOF
from api import client

model = "gpt-4"
# Первый промпт для генерации знания
prompt = """
    Напиши краткий абзац о
    лирических особенностях
    и темах рэпа старой школы.
    """
messages = [
    {
        "role": "system",
        "content": "Ты умный помощник.",
    },
    {
        "role": "user",
        "content": prompt,
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=200,
    temperature=0.5,
    stop=["assistant:", "user:"],
```

```

)

output = response.choices[0].message.content

# Второй промпт для генерации текста песни
# Подаем результат первого промпта на вход
# в качестве второго промпта,
# чтобы сгенерировать качественный контент
prompt = f"""Context: {output}
Задание: сочини текст песни в стиле рэпа
старой школы про правосудие и равенство.
"""
messages = [
    {
        "role": "system",
        "content":
            "Ты сочинитель рэпа в стиле старой школы.",
    },
    {
        "role": "user",
        "content": prompt,
    },
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=500,
    temperature=1,
    stop=["assistant:", "user:"],
)

output = response.choices[0].message.content
print(
    "Промпт"
    f":\n\n{prompt}"
)
print()
print(
    f"Результат:\n\n{output}"
)
EOF

```

Код выводит следующие тексты:

Промпт:

Контекст: рэп старой школы, распространенный с конца 1970-х до середины 1980-х годов, характеризуется незамысловатым, прямолинейным лирическим стилем. Тексты песен часто вращаются вокруг тематики социальных комментариев, повседневного опыта и рассказывания историй, с сильным упором на рифму и ритм. Язык обычно простой и недвусмысленный, с упором на умную игру слов и ритмические синкопы. Биты обычно вдохновлены фанком или диско, создавая отличный фон для лирических повествований.

Задание: сочини текст песни в стиле рэпа старой школы про правосудие и равенство.

Результат:

(Куплет 1)

Йоу, я скажу слово правды, и это так,
 В мире, полном хаоса, я свет прожектора,
 На улицах, в переулках, где холодно и темно.
 Там, где должно свершиться правосудие, нужен праведник.
 Дети в гетто сделаны из стали, из камня,
 Каждый рожден королем, но лишен трона.
 Каждый голос должен быть услышан, каждая история рассказана,
 Потому что каждый человек дороже серебра и золота.

(Припев)

Нам нужна справедливость, нам нужен мир,
 Надо встать на защиту тех, кто остановился.
 В этой стране грез мы все братья и сестры,
 Скажем «нет» нищим, «да» королям.

(Куплет 2)

Копы ждут на углу, надевают наручники на невиновных,
 Закон должен защищать угнетенных.
 Наши улицы – не поле боя, это наши дома.
 Вы должны нас защищать, а не забрасывать камнями.
 Речь идет о равенстве, об уважении.
 Пришло время сильным мира сего задуматься.
 Мы не успокоимся, не будем молчать,
 Потому что самое сильное оружие – это наши голоса.

(Припев)

Нам нужна справедливость, нам нужен мир,
 Хватит сражений, мы хотим спокойствия и любви,
 Стремясь к тому дню, когда мы все сможем петь.
 В стране свободных людей мы все короли.

(Аутро)

У микрофона я борюсь словами, а не кулаками.
 Реальность жестока, но надежда не умирает.
 Справедливость для всех, а не только для избранных.
 Моя любовь к вам – в моих словах.

Вы можете настроить параметр `temperature` для нужного уровня креативности и указать максимальную длину результата в `max_tokens`.

Что такое APPLE – ФРУКТ или КОМПАНИЯ?

Теперь мы создадим небольшое приложение, которое определяет, является слово существительным или прилагательным. Типичная проблема возникает, когда мы подаем на вход модели, например, слово «рабочий». Это слово может быть как существительным, так и прилагательным¹.

- *рабочий* выкопал глубокую яму (существительное);
- *рабочий* компьютер предоставляется нанимателем (прилагательное).

Мы можем обратиться к модели следующим образом:

¹ В английском языке таких слов больше, чем в русском, причем иногда они могут иметь даже тройственный характер. Например, слово `light` может быть существительным, прилагательным или глаголом в зависимости от контекста. – *Прим. перев.*

```

cat > src/app_23.py << EOF
from api import client

model = "gpt-3.5-turbo"
prompt = "Определи, какой частью речи является слово 'рабочий'."
messages = [
    {
        "role": "system",
        "content": "Ты умный помощник.",
    },
    {
        "role": "user",
        "content": prompt,
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
)

output = response.choices[0].message.content
print(output)
EOF

```

После выполнения кода мы можем получить приблизительно такой ответ (зависит от языка запроса и версии модели):

Слово 'рабочий' может быть как существительным, так и прилагательным. В роли существительного оно обозначает человека, выполняющего определенную работу, физическую или умственную. Как прилагательное оно может характеризовать объект, предназначенный или пригодный для выполнения какой-либо работы.

Проблема в том, что модель не всегда правильно определяет контекст, в котором употребляется слово. Чтобы решить (или хотя бы смягчить) эту проблему, можно воспользоваться приемом, который называется «контекстное наполнение» (context stuffing). Идея заключается в том, чтобы дать модели больше разнообразного контекста и тем самым помочь правильно распознать часть речи. Это можно сделать, передавая модели примеры предложений с просьбой определить часть речи.

Вот соответствующий пример кода:

```

cat > src/app_24.py << EOF
from api import client

model = "gpt-3.5-turbo"

prompt_a = """Рабочий пришел вовремя. Определи, какой
частью речи является слово 'рабочий'.\n\n"""

prompt_b = """Это вполне рабочий способ. Определи, какой
частью речи является слово 'рабочий'.\n\n"""

```

```
prompt_c = """Рабочий нуждается в отдыхе. Определи, какой
частью речи является слово 'рабочий'.\n\n"""
```

```
prompt_d = """Следы на снегу говорят о том, что недавно
здесь был рабочий. Определи, какой
частью речи является слово 'рабочий'.\n\n"""
```

```
for prompt in [
    prompt_a,
    prompt_b,
    prompt_c,
    prompt_d
]:
    messages = [
        {
            "role": "system",
            "content": "Ты умный помощник.",
        },
        {
            "role": "user",
            "content": prompt,
        }
    ]

    response = client.chat.completions.create(
        model=model,
        messages=messages,
    )
```

```
output = response.choices[0].message.content
print(output)
print()
EOF
```

При подготовке вывода модель воспользуется контекстом использования слова, и это поможет ей дать правильный ответ:

Слово 'рабочий' может быть как существительным, так и прилагательным.

В предложении 'Рабочий пришел вовремя' слово 'рабочий' указывает на того, кто совершает действие и является существительным.

В предложении 'Это вполне рабочий способ' слово 'рабочий' характеризует объект и является прилагательным.

В предложении 'Рабочий нуждается в отдыхе' слово 'рабочий' указывает на объект и является существительным.

В предложении 'Следы на снегу говорят о том, что недавно здесь был рабочий' слово 'рабочий' указывает на объект и является существительным.

В английском языке существует много других слов, которые могут быть существительным, глаголом и даже наречием в зависимости от контекста.



Твит Грега Брокмана, соучредителя и президента OpenAI

Рассмотрим следующий пример, где мы передаем модели два промпта. В первом случае нам нужно, чтобы модель поняла, что Apple – это компания. Во втором случае нам нужно, чтобы модель поняла, что Apple может быть как компанией, так и фруктом. Так выглядит код:

```
cat > src/app_25.py << EOF
from api import client
```

```
model = "gpt-3.5-turbo"
prompt_a = ""
Huawei:
company
```

```
Google:
company
```

```
Microsoft:
company
```

```
Apple:
""
prompt_b = ""
Huawei:
company
```

```
Google:
company
```

```
Microsoft:
company
```

Apricot:

Fruit

Apple:

"""

```
for prompt in [prompt_a, prompt_b]:
    messages = [
        {
            "role": "system",
            "content":
                "Ты умный помощник.",
        },
        {
            "role": "user",
            "content": prompt,
        }
    ]

    response = client.chat.completions.create(
        model=model,
        messages=messages,
    )

    output = response.choices[0].message.content
    print(output)
    print()
EOF
```

А так будет выглядеть вывод модели:

company

Fruit and company

Хотя это несложный пример, он показывает, как контекстное наполнение может влиять на ответ модели. Рассмотрим второй пример:

```
cat > src/app_26.py << EOF
from api import client

model = "gpt-4"

# Промпт генерации файла Dockerfile для
# приложения Python
prompt_dockerfile = """
# Dockerfile для Node.js:
FROM node:14
WORKDIR /app
COPY . /app
RUN npm install
EXPOSE 8080
CMD ["node", "app.js"]
```

```
# Dockerfile для Python:
"""

# Промпт генерации развертывания скрипта развертывания
# Kubernetes для базы данных MySQL
prompt_kubernetes = """
# Развертывание Kubernetes для Redis:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-deployment
spec:
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis
        ports:
        - containerPort: 6379

# Развертывание Kubernetes для MySQL:
"""

for prompt in [
    prompt_dockerfile,
    prompt_kubernetes
]:
    messages = [
        {
            "role": "system",
            "content":
                "Ты умный помощник, который пишет "
                "скрипты конфигурации.",
        },
        {
            "role": "user",
            "content": prompt,
        }
    ]

    response = client.chat.completions.create(
        model=model,
        messages=messages,
    )

    output = response.choices[0].message.content
```

```
print(output.strip())
print("---")
EOF
```

В этом примере контекстное наполнение применяется для управления моделью при создании конкретных сценариев конфигурации. Модель получает примеры того, что ожидается (Dockerfile для Node.js, развертывание Kubernetes для Redis), а затем ей предлагают сгенерировать аналогичный контент для другой технологии (Dockerfile для Python, развертывание Kubernetes для MySQL и т. д.). Это может гарантировать, что выходные данные модели будут более точно соответствовать конкретным требованиям каждой технологии (файл конфигурации Dockerfile, манифест Kubernetes). В результате модель сможет генерировать более точный контент.

Представьте себе компанию-разработчика программного обеспечения, которая хочет автоматизировать процесс технической поддержки. Компания предоставляет широкий спектр продуктов, включая облачные сервисы, инструменты для разработчиков и корпоративные решения. Служба поддержки клиентов компании часто занимается техническими вопросами, требующими конкретных и подробных ответов. Чтобы повысить эффективность и точность, компания внедряет чат-бот, использующий методы контекстного наполнения для обработки сложных технических запросов. Вот как это работает:

- чат-бот сначала классифицирует входящие запросы на основе ключевых слов, названий продуктов и характера запроса (например, устранение неполадок, настройка, передовой опыт);
- на основе результатов процесса категоризации чат-бот подготавливает контекстно-зависимый промпт. Например, если запрос касается настройки облачного сервиса, чат-бот предварительно загрузит примеры похожих конфигураций;
- чат-бот использует эти контекстные промпты для генерации конкретных и релевантных ответов. Сюда могут входить пошаговые руководства, фрагменты кода или советы по устранению неполадок;
- если первоначального ответа недостаточно, чат-бот задает пользователю дополнительные вопросы для дальнейшего уточнения контекста и предоставления более точных ответов.

Такой подход повышает качество технической поддержки.

ДИНАМИЧЕСКОЕ УПРАВЛЕНИЕ КОЛИЧЕСТВОМ ТОКЕНОВ

В этом примере мы просим модель создать список действий для открытия компании в США. Нам нужны первые n элементов в списке, где n динамически определяется пользователем. Например, если пользователю требуется 5 шагов, модель должна сгенерировать 5 элементов списка. Для этого нам нужно использовать параметр `stop`, чтобы остановить генерацию списка, когда модель достигнет желаемого количества элементов.

Нам также нужно, чтобы вывод был отформатирован следующим образом:

```
Действие 1: [task 1]
Действие 2: [task 2]
```

Действие 3: [task 3]

..

Действие n: [task n]

Для этого мы можем передать модели следующий промпт:

```
prompt = "Составь список действий для открытия компании в США\n\nTask 1:"
```

Затем мы применяем такой параметр stop:

Task [n]:

Вместо n подставляется числовое значение, указанное пользователем.

Следующий код представляет собой пример реализации этого подхода.

```
cat > src/app_27.py << EOF
from api import client
import sys

model = "gpt-4"
# считываем количество шагов из командной строки
try:
    number_of_tasks = int(sys.argv[1])
except:
    print("""
Введите количество действий как аргумент команды.
Примеры:
python src/app_27.py 5
python src/app_27.py 10
""")
    exit()
# Создаем промпт, показывающий модели,
# каким должен быть формат списка действий
prompt = """
Составь список действий для регистрации компании в США

Действие 1:
"""

# Определяем стоп-последовательность.
# Если пользователь хочет получить 5 действий,
# стоп-последовательность должна содержать "Действие 6:"
stop = [
    f"Действие {number_of_tasks + 1}:",
    "assistant:",
    "user:"
]

messages = [
    {
        "role": "system",
        "content": "Ты умный помощник.",
    },
    {
```

```

    "role": "user",
    "content": prompt,
  }
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=200,
    stop=stop,
)

output = response.choices[0].message.content

# Соединяем строку "Действие 1:" и вывод
full_response = "Действие 1: " + output

print(full_response)
EOF

```

Проблема может возникнуть, если пользователь захочет сгенерировать большое количество действий. Например, если пользователь хочет получить список из 100 действий, модель может оказаться не в состоянии это сделать, поскольку она достигнет максимального количества токенов, определенного параметром `max_tokens`. В таком случае полезно сделать этот параметр динамическим, рассчитав количество токенов, необходимых для генерации желаемого количества задач. Мы можем сделать это, используя следующую оценку, предложенную документацией OpenAI:

Полезное эмпирическое правило заключается в том, что один токен обычно соответствует 4 символам обычного английского текста. Это соответствует приблизительно $3/4$ обычного слова (т. е. 100 токенов = 75 слов).

В нашем случае оценка потребности в токенах выглядит так:

- одно действие списка занимает примерно 150 символов, что эквивалентно $150 \times 3/4$ токенов ≈ 113 токенов. Если мы хотим получить более длинное описание действий, можно увеличить количество символов (например, 300 символов = $300 \times 3/4$ токенов = 225 токенов).

Следовательно, для формирования списка из n действий нам нужно сгенерировать $n \cdot 113 + 17$ токенов, и мы можем задать параметр `max_tokens` следующим образом:

```
max_tokens = number_of_tasks * 113 + 17
```

Окончательный код примера будет выглядеть так:

```

cat > src/app_28.py << EOF
from api import client
import sys

model = "gpt-4"
# Считываем количество шагов из командной строки

```

```

try:
number_of_tasks = int(sys.argv[1])
except:
    print("Введите количество действий "
          "как аргумент команды.\n Примеры:\n"
          "python src/app_28.py 5"
          "python src/app_28.py 10"
    )
    exit()

# Создаем промпт, показывающий модели,
# каким должен быть формат списка действий
prompt = """
Составь список действий для регистрации компании в США

Действие 1:
"""

# Определяем стоп-последовательность.
# Если пользователь хочет получить 5 действий,
# стоп-последовательность должна содержать "Действие 6:"
stop = [
    f"Действие {number_of_tasks + 1}:",
    "assistant:",
    "user:"
]

# Определяем максимальное количество токенов.
max_tokens = number_of_tasks * 113 + 17

messages = [
    {
        "role": "system",
        "content":
            "Ты умный помощник.",
    },
    {
        "role": "user",
        "content": prompt,
    }
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=max_tokens,
    stop=stop,
)

output = response.choices[0].message.content

# Соединяем строку "Действие 1:" и вывод

```

```
full_response = "Task 1: " + output

print(full_response)
EOF
```

СОЗДАНИЕ ИНТЕРАКТИВНОГО ПОМОЩНИКА В ОКНЕ КОМАНДНОЙ СТРОКИ

Допустим, перед нами стоит задача создать инструмент командной строки для помощи в использовании команд Linux.

Начнем с простого примера:

```
cat > src/app_29.py << EOF
from api import client

model = "gpt-4"
messages = [
    {
        "role": "system",
        "content":
            "Ты умный помощник."
    },
    {
        "role": "user",
        "content":
            "Список всех файлов в текущем "
            "каталоге."
    },
    {
        "role": "assistant",
        "content":
            "ls -l"
    },
    {
        "role": "user",
        "content":
            "Список всех файлов в текущем "
            "каталоге, включая скрытые файлы."
    },
    {
        "role": "assistant",
        "content":
            "ls -la"
    },
    {
        "role": "user",
        "content":
            "Удаление всех файлов в текущем "
            "каталоге."
    },
    {
```

```

    "role": "assistant",
    "content":
        "гм *"
},
{
    "role": "user",
    "content":
        "Подсчет вхождений слова 'солнце' "
        "в файле 'test.txt'."
},
{
    "role": "assistant",
    "content":
        "grep -o 'солнце' test.txt | wc -l"
},
{
    "role": "user",
    "content":
        "Подсчет количества файлов в текущем "
        "каталоге."
}
]

response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=200,
    temperature=0,
)

output = response.choices[0].message.content.strip()
print(output)
EOF

```

Нам нужен детерминированный ответ, поэтому параметру `temperature` мы присваиваем значение 0. Также мы разрешаем модели использовать достаточно большое количество токенов при помощи параметра `max_tokens`.

Ответ, предоставляемый моделью на запрос «Подсчет количества файлов в текущем каталоге», должен быть следующим:

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

Мы воспользуемся пакетом Python `click`¹ (CLI creation kit) для создания интерфейса командной строки с минимальным применением кода. Это сделает нашу программу более интерактивной.

Начнем с установки пакета Python после активации виртуального пространства разработки:

```
workon openaigptforpythondevelopers
pip install click==8.1.3
```

¹ <https://click.palletsprojects.com/>.

Воспользуемся установленным пакетом для создания собственного интерфейса командной строки (CLI).

```
cat > src/app_30.py << EOF
from api import client
import click

model = "gpt-4"

base_messages = [
    {
        "role": "system",
        "content":
            "Ты умный помощник."
    },
    {
        "role": "user",
        "content":
            "Список всех файлов в текущем "
            "каталоге."
    },
    {
        "role": "assistant",
        "content":
            "ls -l"
    },
    {
        "role": "user",
        "content":
            "Список всех файлов в текущем "
            "каталоге, включая скрытые файлы."
    },
    {
        "role": "assistant",
        "content":
            "ls -la"
    },
    {
        "role": "user",
        "content":
            "Удаление всех файлов в текущем "
            "каталоге."
    },
    {
        "role": "assistant",
        "content":
            "rm *"
    },
    {
        "role": "user",
        "content":
            "Подсчет вхождений слова 'солнце' "
```

```

    "в файле 'test.txt'."
  },
  {
    "role": "assistant",
    "content":
      "grep -o 'солнце' test.txt | wc -l"
  },
]

while True:
    messages = base_messages.copy()

    # чтение пользовательского ввода
    request = input(
        click.style(
            "Ввод: ",
            fg="green"
        )
    )

    # добавление пользовательского ввода к сообщению
    messages.append(
        {
            "role": "user",
            "content": f"{request}"
        }
    )

    # передача сообщения в API
    response = client.chat.completions.create(
        model=model,
        messages=messages,
        max_tokens=200,
        temperature=0,
    )

    # получение команды из ответа
    command = response.choices[0].message.content.strip()

    # Вывод команды на печать
    click.echo(
        click.style(
            "Вывод: ", fg="yellow"
        ) + command
    )
    click.echo()
EOF

```

Здесь мы используем те же самые промпты. Разница только в том, что мы добавляем `click` внутри бесконечного цикла `while`. После запуска скрипта `app_30.py` наша программа запрашивает пользовательский ввод (`request`), затем вставляет его в промпт и передает в API.

В конце программы `click` печатает результат. Оператор `click.echo()` печатает пустую строку.

Ниже показан пример диалога между пользователем и ассистентом:

Ввод: Найти все файлы, которые начинаются с "learn_" в домашнем каталоге и его вложенных каталогах

Вывод: `find ~/ -name "learn_*`

Ввод: Показать архитектуру процессора

Вывод: `uname -m`

Ввод: Показать список только скрытых файлов в каталоге `/home/user/my_project`

Вывод: `ls -d /home/user/my_project/.*`

Нам осталось реализовать команду выхода из программы. Сделаем так, чтобы при вводе команды `exit` или `quit` программа прекращала работу. Для этого добавим в цикл `while` следующий код:

```
# app_30.py
# ...
while True:
    # ...
    if request.lower() in ["exit", "quit"]:
        break
    # ...
```

В завершение добавим возможность выполнения команды, сгенерированной моделью. Вот как это делается:

```
cat > src/app_31.py << EOF
from api import client
import click
import os

model = "gpt-4"
base_messages = [
    {
        "role": "system",
        "content":
            "Ты умный помощник."
    },
    {
        "role": "user",
        "content":
            "Список всех файлов в текущем "
            "каталоге."
    },
    {
        "role": "assistant",
        "content":
            "ls -l"
    },
    {
        "role": "user",
```

```

    "content":
        "Список всех файлов в текущем "
        "каталоге, включая скрытые файлы."
    },
    {
        "role": "assistant",
        "content":
            "ls -la"
    },
    {
        "role": "user",
        "content":
            "Удаление всех файлов в текущем "
            "каталоге."
    },
    {
        "role": "assistant",
        "content":
            "rm *"
    },
    {
        "role": "user",
        "content":
            "Подсчет вхождений слова 'солнце' "
            "в файле 'test.txt'."
    },
    {
        "role": "assistant",
        "content":
            "grep -o 'солнце' test.txt | wc -l"
    }
]

while True:
    messages = base_messages.copy()

    # чтение пользовательского ввода
    request = input(
        click.style(
            "Ввод (введите 'exit' для завершения): ",
            fg="green"
        )
    )

    if request.lower() in ["exit", "quit"]:
        break

    # добавление пользовательского ввода к сообщению
    messages.append(
        {
            "role": "user",
            "content": f"{request}"
        }
    )

```

```

# отправка сообщения в API
response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=200,
    temperature=0,
)

# получение команды из ответа
command = response.choices[0].message.content.strip()

# вывод команды на печать
click.echo(
    click.style(
        "Вывод: ",
        fg="yellow"
    ) + command
)

# спросить пользователя, хочет ли он выполнить команду
click.echo(
    click.style(
        "Выполнить? (y/n): ",
        fg="yellow"),
    nl=False
)
# Чтение пользовательского ввода
choice = input()

# Если пользователь хочет, выполнить команду
if choice == "y":
    os.system(command)
elif choice == "n":
    continue
else:
    click.echo(
        click.style(
            "Неправильный ввод. Введите только 'y' или 'n'.",
            fg="red"
        )
    )

click.echo()
EOF

```

Теперь, если вы запустите скрипт `app_31.py`, код спросит, хотите ли вы выполнить команду:

```

Ввод (введите 'exit' для завершения): список всех файлов в /tmp
Вывод: ls /tmp
Выполнить? (y/n): y
<здесь показан список файлов в /tmp/>

```

Ввод (введите 'exit' для завершения): список всех файлов в /tmp

Вывод: ls /tmp

Выполнить? (y/n): cool

Неправильный ввод. Введите только 'y' или 'n'.

Если вы незнакомы с Linux, будьте осторожны при выполнении команд. Вы можете протестировать этот код в безопасной среде, например в виртуальной машине или контейнере, чтобы избежать повреждения вашей операционной системы при случайном выполнении опасной команды.

Вы можете также проверить команду при помощи метода `os.system(command)` и вывести сообщение об ошибке, если эту команду невозможно выполнить:

```
cat > src/app_32.py << EOF
from api import client
import click

model = "gpt-4"
base_messages = [
    {
        "role": "system",
        "content":
            "Ты умный помощник."
    },
    {
        "role": "user",
        "content":
            "Список всех файлов в текущем "
            "каталоге."
    },
    {
        "role": "assistant",
        "content":
            "ls -l"
    },
    {
        "role": "user",
        "content":
            "Список всех файлов в текущем "
            "каталоге, включая скрытые файлы."
    },
    {
        "role": "assistant",
        "content":
            "ls -la"
    },
    {
        "role": "user",
        "content":
            "Удаление всех файлов в текущем "
            "каталоге."
    },
],
```

```
{
  "role": "assistant",
  "content":
    "гм *"
},
{
  "role": "user",
  "content":
    "Подсчет вхождений слова 'солнце' "
    "в файле 'test.txt'."
},
{
  "role": "assistant",
  "content":
    "grep -o 'солнце' test.txt | wc -l"
},
]

while True:
    messages = base_messages.copy()

    # чтение пользовательского ввода
    request = input(
        click.style(
            "Ввод (введите 'exit' для завершения): ",
            fg="green"
        )
    )

    if request.lower() in ["exit", "quit"]:
        break

    # добавление пользовательского ввода к сообщению
    messages.append(
        {
            "role": "user",
            "content": f"{request}"
        }
    )

    # отправка сообщения в API
    response = client.chat.completions.create(
        model=model,
        messages=messages,
        max_tokens=200,
        temperature=0,
    )

    # получение команды из ответа
    command = response.choices[0].message.content.strip()

    # вывод команды на печать
```

```

click.echo(
    click.style(
        "Вывод: ",
        fg="yellow"
    ) + command
)

# спросить пользователя, хочет ли он выполнить команду
click.echo(
    click.style(
        "Выполнить? (y/n): ",
        fg="yellow"),
    nl=False
)
# Чтение пользовательского ввода
choice = input()

# Если пользователь хочет выполнить команду, выполнить ее
if choice == "y":
    r = os.system(command)
    if r != 0:
        click.echo(
            click.style(
                "Ошибка выполнения команды.",
                fg="red"
            )
        )
# Если пользователь не хочет выполнять команду, продолжить
elif choice == "n":
    continue
else:
    click.echo(
        click.style(
            "Неправильный выбор. Введите только 'y' или 'n'.",
            fg="red"
        )
    )

click.echo()
EOF

```

Что дальше?

Качество результатов API завершения чата фундаментально зависит от используемых параметров. Однако это не единственный фактор, определяющий качество работы. Качество промпта, набор используемых примеров, методология, применяемая при кодировании и решении проблем, а также остроумное мышление пользователя – все это играет важную роль при формировании качества результатов.

По этой причине крайне важно найти время на концептуализацию рассматриваемой проблемы и определение наиболее эффективной стратегии ее

решения. Это требует глубокого понимания проблемы, вдумчивого подхода и инновационного мышления. Кроме того, следует внимательно отнестись к построению промпта и подбору примеров.

Существует очень эффективный прием работы: представьте, что вы объясняете задачу и ее решение постороннему человеку. Это поможет вам сделать промпты и примеры простыми, ясными и практичными. Кроме того, может быть полезно предоставить модели примеры желаемого результата. Такие примеры направляют модель и помогают ей понять, чего от нее ожидают.

В поиске наилучших результатов не обойтись без экспериментов. Важно и необходимо экспериментировать с различными параметрами, методологиями, промптами и примерами, а также повторять их на основе отзывов и результатов. Итерации постепенно приведут вас к наилучшему результату.

Кроме того, решающее значение для достижения оптимальных результатов имеет правильный выбор технологии разработки промптов. Выбор правильного метода в зависимости от варианта использования может существенно повлиять на эффективность модели. На самом деле разработку промптов нельзя назвать «точной наукой», поскольку для достижения наилучших результатов она требует множества проб и ошибок. Тем не менее существуют некоторые передовые методы, практические правила, проверенные методы, а иногда и нестандартные хитрости, которые помогают улучшить результаты в зависимости от задачи и доступной языковой модели.

Обучение с несколькими примерами, обучение без примеров и промпты на основе общих знаний – не единственные известные методы разработки промптов. К ним относятся также цепочки рассуждений (chain of thought, CoT), CoT без примеров, автоматические CoT (AutoCoT), самосогласованность, перенос обучения, ReAct (reason + act, причина + действие) и др. Каждая методика имеет свои сильные и слабые стороны.

Если вы хотите углубить свои знания в этой области, вам пригодится мое учебное пособие «Разработка промптов для больших языковых моделей: искусство и наука раскрытия истинного потенциала LLM¹». Оно содержит подробные объяснения различных методов.

¹ <https://www.google.com/search?q=LLM+Prompt+Engineering+For+Developers:+The+Art+and+Science+of+Unlocking+LLMs&oq=LLM+Prompt+Engineering+For+Developers%3A+The+Art+and+Science+of+Unlocking+LLMs>.

Встраивание

ЧТО ТАКОЕ ВСТРАИВАНИЕ?

Если бы мне пришлось описать эту процедуру одним предложением, я бы сказал, что *встраивание текста* представляет текстовые строки как векторы в многомерном пространстве (т. е. встраивает вектор в пространство), что позволяет измерить сходство между ними. Встраивания обычно используются для таких задач, как поиск наиболее релевантных результатов для поискового запроса, группировка текстовых строк на основе их сходства, рекомендация элементов со схожим текстом, выявление текстовых строк, которые сильно отличаются от других, анализ различий между строками текста и маркировку текстовых строк на основе их ближайшего совпадения.

С практической точки зрения встраивания представляют объекты и отношения реального мира в форме векторов (списков чисел). В этом же векторном пространстве измеряют сходство между двумя объектами.

ВАРИАНТЫ ПРИМЕНЕНИЯ: ОТ ПОИСКОВЫХ СИСТЕМ ДО БЕСПИЛОТНЫХ АВТОМОБИЛЕЙ

Встраивание текста OpenAI измеряет связанность текстовых строк и может применяться для различных целей.

Вот некоторые варианты применения:

- задачи обработки естественного языка (NLP), такие как анализ эмоциональной окраски, семантическое сходство и классификация настроений;
- генерация признаков встроенного текста для моделей машинного обучения, таких как сопоставление ключевых слов, классификация документов и тематическое моделирование;
- создание независимых от языка представлений текста, позволяющих осуществлять межязыковое сравнение текстовых строк;
- повышение точности текстовых поисковых систем и систем понимания естественного языка;
- создание персонализированных рекомендаций путем сравнения введенного пользователем текста с широким диапазоном текстовых строк.

Варианты применения можно обобщить следующим образом:

- **поиск**: результаты ранжируются по релевантности строке запроса;
- **кластеризация**: текстовые строки группируются по сходству;
- **рекомендации**: пользователю предлагается использовать элементы со связанными текстовыми строками;
- **обнаружение аномалий**: выявление слабо связанных выбросов;

- **измерение разнообразия:** анализируется распределение сходства;
- **классификация:** строки классифицируются по наибольшему сходству с метками.

Ниже представлены некоторые практические подходы к использованию встраиваний (не обязательно с применением платформы OpenAI).

Tesla: применение встраиваний в беспилотных автомобилях

Работа с неструктурированными данными, такими как необработанный текст, изображения и видео, может создавать проблемы. Часто бывает трудно получить исходные данные из-за проблем с конфиденциальностью, а построение моделей с нуля может потребовать значительных вычислительных мощностей, огромных наборов данных и большого количества времени.

Встраивание предлагает решение, позволяющее передавать информацию из одного контекста (например, изображения автомобиля) в другой (например, компьютерную игру). Этот процесс, называемый переносом обучения, позволяет обучать модели без необходимости использования обширных реальных данных.

Такие компании, как Tesla, используют эту технику для своих беспилотных автомобилей.

Kalendar AI: применение встраиваний в управлении продажами

Kalendar AI – это инструмент для работы с продажами, который использует встраивание для подбора подходящего предложения по продажам нужным клиентам на основе набора данных из 340 млн профилей.

Автоматизация рекомендаций основана на сходстве между встраиваниями профилей клиентов и рекламных предложений и ранжировании наиболее подходящих совпадений. По данным OpenAI, этот метод снизил количество нежелательного таргетинга на 40–56 % по сравнению с предыдущим подходом.

Notion: расширенные возможности поиска

Notion, онлайн-инструмент для организации рабочего пространства, расширил свои возможности поиска за счет использования функционала встраивания OpenAI. Это усовершенствование выходит за рамки базовых систем сопоставления ключевых слов, использовавшихся ранее в данном инструменте.

Благодаря этой новой функции Notion может более точно понимать структуру, контекст и значение контента, хранящегося на его платформе. Это позволяет пользователям выполнять более точный поиск и эффективнее извлекать документы.

DALL-E 2: преобразование текста в изображение

DALL-E 2 – это система, предназначенная для преобразования текстовых меток в изображения.

Она работает с использованием двух моделей: Prior и Decoder. Модель Prior обрабатывает текстовые метки для создания встраиваний изображений CLIP¹. Впоследствии Decoder берет встраивания изображений CLIP и создает изученное изображение. Затем это изображение масштабируется с разрешения 64×64 до 1024×1024.

ИЗУЧАЕМ ВСТРАИВАНИЕ ТЕКСТА

Начнем с простейшего примера:

```
cat > src/app_33.py << EOF
from api import client

response = client.embeddings.create(
    model="text-embedding-ada-002",
    input="Я программист",
)

print(response)
EOF
```

Как обычно, мы импортируем `openai`, проходим аутентификацию и вызываем конечную точку. Однако на этот раз используем одну из лучших моделей, доступных в OpenAI для встраивания, под названием Ada. Команда OpenAI рекомендовала модель `text-embedding-ada-002` почти для всех случаев использования, описывая ее как лучшую по качеству, дешевую и простую в использовании. Однако недавно компания выпустила новые модели для той же цели. Вот список доступных моделей:

Модель	Страниц на доллар (приблизительно)	Оценка качества по тесту MTEB, %
text-embedding-3-small	62 500	62.3
text-embedding-3-large	9615	64.6
text-embedding-ada-002	12 500	61.0

MTEB (massive text embedding benchmark) – это тест для оценки качества встраивания текста.

В нашем текущем примере вывод будет объектом `CreateEmbeddingResponse`:

```
CreateEmbeddingResponse(
    data=[
        Embedding(
            embedding=[
                -0.016873637,
                -0.019692589,
                ..etc
            ],

```

¹ <https://openai.com/blog/clip/>.

```

        index=0,
        object='embedding'
    )
],
model='text-embedding-ada-002',
object='list',
usage=Usage(
    prompt_tokens=4,
    total_tokens=4
)
)

```

Объект включает в себя следующие атрибуты:

- `data`: список встраиваний. Каждое встраивание представляет собой список чисел с плавающей запятой;
- `model`: имя модели, используемой для создания встраиваний;
- `object`: тип объекта, представляющий собой список;
- `usage`: статистика использования для вызова API;
 - ◆ `prompt_tokens`: количество токенов, используемых промптом;
 - ◆ `total_tokens`: общее количество токенов, сгенерированных вызовом API.

Мы можем получить доступ к встраиванию напрямую, используя `response.data[0].embedding`:

```

cat > src/app_34.py << EOF
from api import client

response = client.embeddings.create(
    model="text-embedding-ada-002",
    input="Я программист",
)

print(response.data[0].embedding)
EOF

```

Приведенный выше код Python выводит на печать список чисел с плавающей запятой, например 0.010284645482897758 или 0.013211660087108612. Эти числа представляют собой встраивание входного текста «Я программист», сгенерированное моделью OpenAI text-embedding-ada-002.

Встраивание – это многомерное представление входного текста, отражающее его смысл. Иногда его называют *векторным представлением*, или просто *вектором встраивания*.

По сути, это метод представления объекта, такого как текст или изображение, с помощью массива значений. Каждое значение обозначает отдельный аспект объекта, отражая интенсивность этого аспекта для этого конкретного объекта. В случае текстового объекта эти аспекты могут охватывать темы, эмоциональную окраску или другие семантические характеристики.

Проще говоря, векторное представление, созданное конечной точкой встраивания, переводит данные в формат, понятный моделям и алгоритмам машинного обучения. Это метод преобразования конкретных входных данных в форму, удобную для использования этими моделями и алгоритмами.

Важно отметить, что при использовании модели встраивания text-embedding-ada-002 длина ввода (в нашем случае строка «Я программист») не должна превышать 8191 токена. Это ограничение может отличаться для других моделей встраивания. Учитывая, что 100 токенов примерно соответствуют 75 английским словам, максимальный объем входного текста для модели text-embedding-ada-002 составляет примерно 6143 слова (8191 токен × 0.75). При условии что средняя страница на английском языке содержит около 500 слов, этот объем соответствует примерно 12.29 страницы английского текста.

В следующих разделах мы рассмотрим примеры применения встраиваний.

ВСТРАИВАНИЯ ДЛЯ НЕСКОЛЬКИХ ВХОДОВ

В последнем примере мы использовали следующий ввод:

```
input="Я программист",
```

Можно также применить несколько вводов, и вот как это делается:

```
cat > src/app_35.py << EOF
from api import client

model = "text-embedding-ada-002"
inputs = [
    "Я программист",
    "Я писатель"
]
response = client.embeddings.create(
    model=model,
    input=inputs,
)

for i, embedding in enumerate(response.data):
    print(
        f"Встраивание ввода {i}: "
        f"{embedding.embedding[:5]}..(обрезано)"
    )
EOF
```

ПРИМЕР ПРИМЕНЕНИЯ: СЕМАНТИЧЕСКИЙ ПОИСК

Забудем на минутку об OpenAI и вернемся в мир поисковых систем. Это лучший способ понять, что такое семантический поиск, – поисковыми системами пользуются все!

Поисковые системы были созданы, чтобы помочь нам найти нужную информацию. Однако способ поиска со временем изменился. Раньше мы использовали поиск по ключевым словам, а поисковые системы были похожи на систему карточек в старой библиотеке. Они сопоставляли ключевые слова в поисковом запросе с ключевыми словами в индексе и возвращали результаты. Раньше это был простой лексический поиск. Вы получали список ссылок,

содержащих ключевые слова, которые вы искали. Сейчас мы ищем концепции и идеи. Мы задаем вопросы и ожидаем получить содержательный ответ. Мы хотим найти нужную нам информацию, а не просто список ссылок. Как все изменилось!

Добро пожаловать в мир семантического поиска.

Представьте, что вы идете в книжный магазин и, вместо того чтобы искать книгу самостоятельно, излагаете продавцу смутное представление о том, какую книгу вам хотелось бы почитать. «Книга о юном волшебнике из школы магии», – говорите вы. И тут же владелец книжного магазина, который, казалось бы, обладает сверхъестественной способностью читать мысли – или по крайней мере понимать вас лучше, чем вы понимаете себя, – понимающе кивает и с улыбкой вручает вам книгу о Гарри Поттере. В этом и заключается магия семантического поиска в сети.

При обычном поиске по ключевым словам поисковые системы действуют как старомодные библиотекари. Вы предоставляете им термин, и они возвращают список документов, в которых этот термин присутствует. Этот подход работает очень буквально и часто не совсем попадает в цель. А семантический поиск? Это как проницательный владелец книжного магазина. Он стремится понять вас. Он ценит контекст, синонимы и даже вашу индивидуальную историю и предпочтения. Имеет значение не только то, какой запрос вы ввели, но и то, что вы стремились получить, потому что для людей это не всегда одно и то же.

Как работает это волшебство? Главным образом благодаря векторному поиску. Слова преобразуются в многомерные векторы, которые помещаются в пространство, где «концептуальная близость» имеет большее значение, чем точная формулировка. Добавьте к этому немного машинного обучения, и система с каждым разом будет становиться все совершеннее. Она учится на бесчисленных запросах, вашей истории поиска и контексте, становясь удивительно искусной в предоставлении того, что вы действительно хотите, а не только того, о чем вы сказали, что хотите.

В следующем разделе узнаете, как реализовать векторный поиск с использованием встраивания текста OpenAI.

Что такое косинусное подобие

В области обработки естественного языка (NLP) косинусное сходство является широко используемым показателем для измерения сходства между документами независимо от их размера. С математической точки зрения мы вычисляем косинус угла между двумя векторами, проецируемыми в многомерное пространство. Мы исследуем угол между двумя векторами и сравниваем их. Результатом сравнения является число от -1 до 1 . Если векторы идентичны, результат равен 1 . Если векторы совершенно разные, результат равен -1 . Если векторы находятся под углом 90° , результат равен 0 .

Представьте себе двух туристов, отправляющихся из одной и той же точки леса и идущих в разных направлениях. Если они оба идут прямо, не поворачивая, они идут по очень похожему пути. В контексте косинусного подобия мы бы сказали, что их пути (или векторы) почти идентичны, с ко-

синусным подобием, близким к 1. Теперь представьте, что один турист решает повернуть налево, а другой поворачивает направо. Их пути теперь расходятся, как два вектора, направленные в разные стороны. Если они движутся в совершенно противоположных направлениях, их пути (векторы) совершенно различны, что дает косинусное сходство, близкое к -1 . Итак, чем меньше угол между их путями, тем более похожи их приключения. Аналогично в NLP, чем меньше угол между двумя векторами (или чем ближе значение косинуса к 1), тем более похожи два документа.



Аналогия с туристами в лесу

Косинусное подобие вычисляется по следующей формуле:

$$\text{Подобие} = (A \cdot B) / (\|A\| \cdot \|B\|),$$

- A и B – векторы;
- $A \cdot B$ обозначает перемножение двух числовых множеств. Для этого каждое число в одном множестве умножают на соответствующий элемент другого множества, затем суммируют все произведения;
- $\|A\|$ обозначает длину вектора A. Она вычисляется как квадратный корень из суммы квадратов всех элементов вектора A.

Возьмем векторы $A = [2, 3, 5, 2, 6, 7, 9, 2, 3, 4]$ и $B = [3, 6, 3, 1, 0, 9, 2, 3, 4, 5]$.

В следующем примере показано вычисление косинусного подобия при помощи кода Python:

```
cat > src/app_36.py << EOF
import numpy as np
from numpy.linalg import norm

# определение двух векторов
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])

# вывод двух векторов на печать
print("Вектор A: {}".format(A))
print("Вектор B: {}".format(B))

# вычисление косинусного подобия
cosine = np.dot(A,B)/(norm(A)*norm(B))

# вывод косинусного подобия на печать
print(
    "Косинусное подобие между A и B: "
    f"{cosine}"
)
EOF
```

Запустите скрипт командой `python src/app.py`, и вы должны увидеть следующий результат:

```
Вектор A: [2 3 5 2 6 7 9 2 3 4]
Вектор B: [3 6 3 1 0 9 2 3 4 5]
Косинусное подобие между A и B: 0.7539959431593041
```

Далее вам необходимо установить библиотеку NumPy при помощи следующей команды:

```
workon openaigptforpythondevelopers
pip install numpy==1.26.3
```

NumPy – это библиотека Python, которая обеспечивает поддержку больших многомерных массивов и матриц в сочетании с обширной коллекцией математических функций высокого уровня для работы с ними.

Мы можем также написать аналогичную программу с использованием Python SciPy – библиотеки научных вычислений для Python. В этом случае код может выглядеть так:

```
import numpy as np
from scipy import spatial

# определение двух векторов
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])
```

```
# вывод двух векторов на печать
print(f"Вектор A: {A}")
print(f"Вектор B: {B}")

# вычисление косинусного подобия
cosine = 1 - spatial.distance.cosine(
    A,
    B
)

# вывод косинусного подобия на печать
print(f"Косинусное подобие между A и B: {cosine}")
```

Scikit-learn – это популярная библиотека машинного обучения на Python.

Результат во всех случаях один и тот же, что доказывает, что косинусное подобие – универсальное математическое понятие. Чтобы продемонстрировать это, давайте рассмотрим все подходы в рамках одного кода.

```
cat > src/app_37.py << EOF
import numpy as np
from numpy.linalg import norm
from scipy import spatial
from sklearn.metrics.pairwise import cosine_similarity

# определение двух векторов
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])

# вычисление косинусного подобия с помощью NumPy
cosine_np = np.dot(A,B)/(norm(A)*norm(B))

# вывод косинусного подобия на печать
print(f"Косинусное подобие между A и B: {cosine_np}")

# вычисление косинусного подобия с помощью SciPy
cosine_sc = 1 - spatial.distance.cosine(A, B)

# вывод косинусного подобия на печать
print(f"Косинусное подобие между A и B: {cosine_sc}")

# вычисление косинусного подобия с помощью sklearn
cosine_sk = cosine_similarity(
    [A],
    [B]
)

# вывод косинусного подобия на печать
print(f"Косинусное подобие между A и B: {cosine_sk[0][0]}")
EOF
```

Вы должны получить следующий результат:

Косинусное подобие между A и B: 0.7539959431593041

Косинусное подобие между A и B: 0.7539959431593041

Косинусное подобие между A и B: 0.7539959431593041

Для запуска предыдущего примера кода необходимо установить следующие библиотеки:

```
workon openaigptforpythondevelopers
```

```
pip install scipy scikit-learn
```

СЕМАНТИЧЕСКИЙ ПОИСК И ВСТРАИВАНИЕ ТЕКСТА OPENAI

Это простой базовый пример, но мы рассмотрим и более сложные примеры. Наша цель – выполнить семантический поиск с использованием встраивания текста OpenAI. Мы начнем с файла CSV, содержащего список слов. Затем найдем слово, наиболее похожее на заданный пользователем поисковый запрос.

Начнем с создания файла под названием `words.csv`. Этот CSV-файл будет содержать один столбец с названием `text` и список случайных слов. Вот как выглядит содержимое файла:

```
cat > src/words.csv << EOF
```

```
text
```

```
яблоко
```

```
банан
```

```
вишня
```

```
собака
```

```
кот
```

```
дом
```

```
машина
```

```
дерево
```

```
телефон
```

```
компьютер
```

```
телевидение
```

```
книга
```

```
музыка
```

```
еда
```

```
вода
```

```
небо
```

```
воздух
```

```
солнце
```

```
луна
```

```
звезда
```

```
океан
```

```
стол
```

```
кровать
```

```
диван
```

```
лампа
```

```
ковер
```

```
окно
```

```
дверь
```

```
пол
```

```
потолок
```

```

стена
часы
будильник
украшения
кольцо
ожерелье
браслет
серьга
кошелек
ключ
фото
EOF

```

Далее мы можем использовать Pandas (библиотеку анализа данных Python), чтобы прочитать файл и создать базу данных Pandas. Pandas – чрезвычайно мощный инструмент для работы с данными, включая данные в файлах CSV. Он прекрасно подходит для нашего варианта использования.

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

DataFrame (df) – наиболее часто используемый объект Pandas. Это двумерная структура данных с помеченными столбцами, которые могут содержать данные разных типов. Ее можно рассматривать как электронную таблицу, таблицу SQL или словарь объектов Series.

Выполнив оператор `print(df)`, вы увидите таблицу с двумя столбцами: идентификатор и текст. Столбец идентификатора – это индекс каждой строки, а столбец текста содержит слова.

Прежде чем продолжить, давайте установим `pandas`, используя следующую команду:

```
workon openaigptforpythondevelopers
pip install pandas==2.2.0
```

Так выглядит пример работы с DataFrame:

```
cat > src/app_38.py << EOF
import pandas as pd
df = pd.read_csv('words.csv')
print(df)
EOF
```

Запустите скрипт командой `python src/app_38.py`, и вы увидите таблицу, аналогичную показанной ниже:

```

          text
0         яблоко
1         банан
2         вишня
3         собака
4          кот
5          дом
6         машина
7         дерево

```

8	телефон
9	компьютер
10	телевидение
11	книга
12	музыка
13	еда
14	вода
15	небо
16	воздух
17	солнце
18	луна
19	звезда
20	океан
21	стол
22	кровать
23	диван
24	лампа
25	ковер
26	окно
27	дверь
28	пол
29	потолок
30	стена
31	часы
32	будильник
33	драгоценность
34	кольцо
35	ожерелье
36	браслет
37	серьга
38	кошелек
39	ключ
40	фото

Теперь нам нужно получить встраивания для каждого слова в DataFrame. Мы начнем со следующего фрагмента кода:

```
from api import client
import pandas as pd

def get_embedding(text, model):
    text = text.replace("\n", " ")
    return client.embeddings.create(
        input = [text],
        model=model
    ).data[0].embedding

# создаем DataFrame из csv-файла
df = pd.read_csv('words.csv')
print(df)

# Задаем входной текст
input = "Черная кошка села на коврик"
```

```
# определяем модель
model = "text-embedding-ada-002"

# получаем встраивание входа с помощью
# функции get_embedding()
embedding = get_embedding(
    input,
    model=model
)

print(embedding)
```

Мы также воспользуемся функцией `apply`, которую имеет каждый объект `DataFrame`. С помощью этой функции мы можем применить лямбда-функцию к оси кадра данных:

```
# получаем встраивание для каждого слова в кадре данных
df['embedding'] = df['text'].apply(
    lambda x: get_embedding(
        x,
        model=model
    )
)
```

Если вы незнакомы с понятием лямбда-функции¹, то пока вам достаточно знать, что это небольшая анонимная функция, которая может получать любое количество аргументов, но содержит только одно выражение.

Синтаксис: `lambda arguments : expression`. В нашем случае мы используем его для получения встраивания каждого слова в кадре данных.

Теперь у нас есть кадр данных с двумя осями: `text` и `embeddings`. Последняя содержит встраивание каждого слова на оси `text`.

Давайте сохраним фрейм данных в другой файл CSV:

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

Вот как выглядит код на этом этапе:

```
cat > src/app_39.py << EOF
from api import client
import pandas as pd

def get_embedding(text, model):
    text = text.replace("\n", " ")
    return client.embeddings.create(
        input = [text],
        model=model
    ).data[0].embedding

# создаем кадр данных из csv-файла
df = pd.read_csv('words.csv')
```

¹ <https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions>.

Задаем входной текст

```
input = "Черная кошка села на коврик"
```

определяем модель

```
model = "text-embedding-ada-002"
```

создаем столбец кадра данных для встраивания

```
df['embedding'] = df['text'].apply(
    lambda x: get_embedding(
        x,
        model=model
    )
)
```

сохраняем новый кадр данных в csv-файл

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

EOF

Так выглядит скриншот таблицы встраиваний для слов на английском языке:

		text
1	0	apple [0.007746271610857424, -0.023053665203194618, -0.007385133765637875, -0.027785107493400574, -0.00426256562808156, 0.01289141271263361, -0.021929115056991577, -0.008578275303306
2	1	banana [-0.0189597673230865, 0.032009102404115284, -0.017362264916300774, -0.003774852491915226, -0.004237709101289511, -0.00863989877700806, 0.02240205471211338, 0.01417149825850544,
3	2	cherry [-0.013906940817832947, -0.0325483812689781, 0.00765211321413517, -0.016582874581217766, -0.00508493138472062, 0.00406061489135027, -0.01981244795024395, -0.012035106300758
4	3	dog [-0.011501237750053406, -0.03050299733877182, -0.02987026423215866, 0.0046232563494533, 0.028947528451681137, 0.025448860770463943, 0.004775515801324036
5	4	cat [-0.006527523975819349, -0.018981920555233955, -0.0008644738118164241, -0.02344200760126114, -0.011608174070715904, -0.01324402504092216, -0.017588142305612564, 0.01872912814
6	5	house [-0.0033176764845848083, -0.017689770087599754, -0.01589241437613964, -0.01750057376921177, -0.018000591546297073, 0.02189261093735695, -0.012426083907485008, -0.0227980464696
7	6	car [-0.00794582542770615, -0.017328109592199326, -0.009644086472690105, -0.03070768155175566, -0.012548675766626793, 0.003105211304500699, -0.00511321391374123, -0.041218172751
8	7	tree [-0.007691123988479376, -0.009551786397191048, -0.08595878630876541, -0.017029838636517525, -0.018833665172248607, 0.017553634762769398, 0.02156071186150932, -0.00224479794262
9	8	phone [-0.007126736920326948, 0.007172489073127508, -0.025761835277080536, -0.01388420476198196, -0.00703875248396587, 0.014584295451641083, -0.006655140779912472, -0.010346966795
10	9	computer [-0.00478981744480145, -0.004698160570114851, -0.03268995136022568, -0.01407408434897618, -0.00807501498878002, -0.003241458674892783, -0.02907284162
11	10	television [-0.02186582237482071, -0.024400677592873573, -0.007846131920814514, -0.00983826071023941, 0.018071940168738365, 0.01713360860571106, 0.03249957412481306, -0.0157738532871007
12	11	book [-0.0292450702237778, -0.009899452328681946, -0.007186621427536011, -0.018221519887447357, -0.07141748070716858, 0.006754880305379629, 0.003311149310206158, -0.0053882646049
13	12	music [-0.004750677384431305, -0.013216584920883179, -0.017282696440815926, -0.003591388463973999, -0.010354483163747875, 0.006150768604129553, -0.009545951532185077, -0.0232821032
14	13	food [-0.01980079110483515, -0.0435369834038559, -0.026708371937274933, -0.004633716772341758, 0.031620729714623043, 0.03930439427495003, 0.02238323302309532, 0.010431431233882904
15	14	water [-0.00141049164310098, -0.02289075769346237, 0.0029633594676852226, -0.0018764965236187, -0.011382912285629359, 0.038604339663666, 0.03444046083255047, 0.0282695004145860

Результат встраивания

Таблица содержит три столбца: `id`, `text` и `embeddings`. Столбец `embeddings` содержит встраивания для каждого слова в столбце `text`, полученные при помощи кода:

```
df['embedding'] = df['text'].apply(
    lambda x: get_embedding(
        x,
        model='text-embedding-ada-002'
    )
)
```

Следующий шаг заключается в использовании косинусного сходства для поиска слова, наиболее похожего на заданный пользователем поисковый запрос. Вот как мы это сделаем.

Мы начнем с чтения нового файла и преобразования последнего столбца в массив NumPy, поскольку это необходимо для расчета косинусного подобия. Наши встраивания при сохранении в файл CSV записываются в виде строк. Однако чтобы использовать эти встраивания для любых математических вычислений, таких как поиск косинусного сходства, нам нужно представить их в структурированном и эффективном формате – массиве NumPy.

Массивы NumPy широко применяются в численных расчетах. Обычная структура данных, такая как список, не очень удобна в подобного рода вычислениях. Кроме того, массив занимает меньше памяти и работает быстрее. Это связано с тем, что массив представляет собой набор однородных типов данных, хранящихся в смежных участках памяти, а список в Python представляет собой набор разнородных типов данных, хранящихся в несмежных участках памяти.

Вернемся к нашему коду и преобразуем последний столбец в массив NumPy с помощью такого блока кода:

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

На этом этапе код выглядит следующим образом:

```
cat > src/app_40.py << EOF
from api import client
import pandas as pd
import numpy as np

def get_embedding(text, model):
    text = text.replace("\n", " ")
    return client.embeddings.create(
        input = [text],
        model=model
    ).data[0].embedding

# в файле words.csv есть столбец 'text',
# который содержит слова
df = pd.read_csv('words.csv')
```

```
# определяем модель
model = "text-embedding-ada-002"

# получаем встраивания для каждого слова в кадре данных
df['embedding'] = df['text'].apply(
    lambda x: get_embedding(
        x, model=model
    )
)

# сохраняем кадр данных в csv-файл
df.to_csv('embeddings.csv')

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

# преобразовываем ось встраиваний в массив NumPy
df['embedding'] = df['embedding'].apply(
    eval
).apply(
    np.array
)
EOF
```

Мы запросим у пользователя данные ввода, прочитаем их и выполним семантический поиск, который сравнивает входные данные с встраиваниями в кадре данных (введенные пользователем данные являются поисковым запросом). Это делается с помощью функции косинусного подобия.

Вот соответствующие шаги в коде:

```
# 0) создаем функцию Python cosine_similarity
def cosine_similarity(a, b):
    numerator = np.dot(a, b)
    denominator = np.linalg.norm(a) * np.linalg.norm(b)
    return numerator / denominator

# 1) получаем поисковый запрос от пользователя
user_search = input('Введите поисковый запрос: ')

# 2) получаем встраивания для поискового запроса
user_search_embedding = get_embedding(
    user_search,
    model=model
)

# 3) вычисляем косинусное подобие между поисковым
# запросом и каждым словом в кадре данных
df['similarity'] = df['embedding'].apply(
    lambda x: cosine_similarity(
        x,
        user_search_embedding
    )
)
```

Давайте рассмотрим подробнее три последние операции в приведенном выше коде.

1. Мы запрашиваем пользовательский ввод и сохраняем его в переменной с именем `user_search`.
2. Функция `get_embedding` получает встраивание для заданного пользователем поискового запроса `user_search`. Параметр `model` имеет значение `text-embedding-ada-002`, что указывает на используемую модель встраивания текста OpenAI.
3. Мы создаем новый столбец в кадре данных с именем `similarity` и используем метод `apply` с лямбда-функцией для расчета косинусного сходства между встраиванием поискового запроса пользователя и встраиванием каждого слова в кадр данных.

Вот полный код:

```
cat > src/app_41.py << EOF
from api import client
import pandas as pd
import numpy as np

def get_embedding(text, model):
    text = text.replace("\n", " ")
    return client.embeddings.create(
        input = [text],
        model=model
    ).data[0].embedding

def cosine_similarity(a, b):
    numerator = np.dot(a, b)
    denominator = np.linalg.norm(a) * np.linalg.norm(b)
    return numerator / denominator

# в файле words.csv есть столбец 'text',
# который содержит слова
df = pd.read_csv('words.csv')

# определяем модель
model = "text-embedding-ada-002"

# получаем встраивания для каждого слова в кадре данных
df['embedding'] = df['text'].apply(
    lambda x: get_embedding(
        x, model=model
    )
)

# сохраняем кадр данных в csv-файл
df.to_csv('embeddings.csv')

# читаем csv-файл
df = pd.read_csv('embeddings.csv')
```

```

# преобразовываем ось встраиваний в массив NumPy
df['embedding'] = df['embedding'].apply(
    eval
).apply(
    pr.aggay
)

# получаем поисковый запрос пользователя
user_search = input('Введите поисковый запрос: ')

# получаем встраивание поискового запроса
search_term_embedding = get_embedding(
    user_search,
    model=model
)

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

# выводим кадр данных на печать
print(df)
EOF

```

Запустите код и протестируйте его, введя поисковый запрос. Например, если вы введете слово «офис», то получите таблицу наподобие следующей¹:

	text	embedding	similarity
0	0	яблоко [0.00777884665876627, -0.023069249466061592, ...	0.830240
1	1	банан [-0.013926557265222073, -0.03288617357611656, ...	0.805654
2	2	вишня [0.006517840549349785, -0.019012855365872383, ...	0.792175
3	3	собака [-0.00337603478692472, -0.017694612964987755, ...	0.828722
4	4	кот [-0.007100660353899002, -0.017430506646633148, ...	0.802095
5	5	дом [-0.007176146376878023, 0.007186704780906439, ...	0.874353
6	6	машина [-0.007485327776521444, -0.021592551842331886, ...	0.821579
7	7	дерево [-0.00465209037065506, -0.013106998056173325, ...	0.825751
8	8	телефон [-0.0014444905100390315, -0.02283545397222042, ...	0.853143
9	9	компьютер [-0.003079379675909877, -0.014275545254349709, ...	0.861701
10	10	телевизор [-0.004756690934300423, -0.019938383251428604, ...	0.799461
11	11	книга [-0.006820248905569315, -0.01918574422597885, ...	0.838061
12	12	музыка [-0.0018608466489240527, -0.023303421214222908, ...	0.820512
13	13	еда [0.022322265431284904, -0.027532152831554413, ...	0.831643
14	14	вода [0.019045095890760422, -0.012522426433861256, ...	0.816802
15	15	небо [0.0049745566211640835, -0.0014098514802753925, ...	0.818561
16	16	воздух [0.008967065252363682, -0.023472610861063004, ...	0.801313

¹ Числовые значения приведены для англоязычного примера. У встраиваний на русском языке числа могут отличаться. – *Прим. перев.*

17	17	солнце	[0.02472955547273159, -0.0024875381495803595, ... 0.816008
18	18	луна	[0.01746830902993679, -0.00917647872120142, 0.... 0.801884
19	19	звезда	[0.011601175181567669, -0.009590022265911102, ... 0.812202
20	20	океан	[0.0055267661809921265, 3.4062537451973185e-05... 0.798045
21	21	стол	[0.012808148749172688, -0.020803043618798256, ... 0.890042
22	22	кровать	[0.005880965385586023, 0.00404005404561758, 0.823849
23	23	диван	[0.011828833259642124, -0.011590897105634212, ... 0.814421
24	24	лампа	[0.006781177595257759, -0.008739246986806393, ... 0.833910
25	25	ковер	[0.00932883471250534, -0.013073415495455265, ... 0.802651
26	26	окно	[0.007422781083732843, -0.016799665987491608, ... 0.829904
27	27	дверь	[-0.004837467800825834, -0.02687481977045536, ... 0.833266
28	28	пол	[0.018671365454792976, -0.021179255098104477, ... 0.861466
29	29	потолок	[-0.017032397910952568, -0.009726257994771004,... 0.813452
30	30	стена	[0.0011230265954509377, 0.014665304683148861, ... 0.829238
31	31	часы	[-0.01110783126205206, -0.013732057064771652, ... 0.815311
32	32	будильник	[-0.002464910503476858, -0.010481107048690319,... 0.818359
33	33	драгоценность	[-0.016047729179263115, 0.01034056767821312, ... 0.783392
34	34	кольцо	[-0.02057298645377159, -0.02570943906903267, ... 0.819219
35	35	ожерелье	[-0.024897022172808647, 0.0024072236847132444,... 0.777741
36	36	браслет	[-0.03433946892619133, 0.005081846844404936, -... 0.779702
37	37	серьга	[-0.025952277705073357, -0.009174846112728119,... 0.776344
38	38	кошелек	[0.015422397293150425, -0.020142966881394386, ... 0.825861
39	39	ключ	[0.0036358926445245743, -0.02881169319152832, ... 0.815508
40	40	фото	[0.004301978275179863, -0.031353630125522614, ... 0.833180

Используя ось `similarity`, мы можем увидеть, какие слова семантически схожи со словом «офис». Чем выше значение с плавающей запятой, тем более похоже на него слово из столбца `text`.

Такие слова, как «ожерелье», «браслет» и «серьги», имеют оценку 0.77, однако слово «стол» имеет оценку 0.89.

Чтобы сделать результаты более читаемыми, мы можем отсортировать кадр данных по оси сходства при помощи следующего кода:

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

Для удобства можно взять только верхние 10 слов отсортированного списка:

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

Усовершенствованный код выглядит так:

```
from api import client
import pandas as pd
import numpy as np

def get_embedding(text, model):
    text = text.replace("\n", " ")
    return client.embeddings.create(
        input = [text],
        model=model
    ).data[0].embedding
```

```
def cosine_similarity(a, b):
    numerator = np.dot(a, b)
    denominator = np.linalg.norm(a) * np.linalg.norm(b)
    return numerator / denominator

# в файле words.csv есть столбец 'text',
# который содержит слова
df = pd.read_csv('words.csv')

# определяем модель
model = "text-embedding-ada-002"

# получаем встраивания для каждого слова в кадре данных
df['embedding'] = df['text'].apply(
    lambda x: get_embedding(
        x, model=model
    )
)

# сохраняем кадр данных в csv-файл
df.to_csv('embeddings.csv')

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

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

# получаем поисковый запрос пользователя
user_search = input('Введите поисковый запрос: ')

# получаем встраивание поискового запроса
search_term_embedding = get_embedding(
    user_search,
    model=model
)

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

# сортировка кадра данных по оси similarity
df = df.sort_values(
```

```

    by='similarity',
    ascending=False
)

# вывод на печать верхних 10 значений сходства
print(df.head(10))
EOF

```

Так выглядит результат для 10 наиболее похожих слов:

21	21 стол	[0.012808148749172688, -0.020803043618798256, ...	0.890042
5	5 дом	[-0.007176146376878023, 0.007186704780906439, ...	0.874353
9	9 компьютер	[-0.003079379675909877, -0.014275545254349709, ...	0.861701
28	28 этаж	[0.018671365454792976, -0.021179255098104477, ...	0.861466
8	8 телефон	[-0.0014444905100390315, -0.02283545397222042, ...	0.853143
11	11 книга	[-0.006829570047557354, -0.019116051495075226, ...	0.838060
24	24 лампа	[0.006781177595257759, -0.008739246986806393, ...	0.833910
27	27 дверь	[-0.004837467800825834, -0.02687481977045536, ...	0.833266
40	40 фото	[0.004301978275179863, -0.031353630125522614, ...	0.833180
13	13 еда	[0.022320540621876717, -0.026822732761502266, ...	0.831016

Попробуйте ввести другие поисковые запросы («собака», «стена», «телефон» и т. д.) и посмотрите, что получится.

ЗА КУЛИСАМИ: КАК РАБОТАЕТ ВСТРАИВАНИЕ

Идея встраивания не нова. Она известна уже давно. Однако в последние годы она приобрела популярность благодаря развитию глубокого обучения и успеху технологий встраивания слов, таких как Word2Vec, GloVe и FastText.

Способность механизма встраивания понимать, что слово «яблоко» ближе к слову «банан», чем к слову «собака», является результатом того, как встраивания фиксируют семантические отношения между словами на основе их использования и контекста в больших наборах данных. Этот процесс можно разбить на несколько ключевых компонентов:

- *обучение контексту*: встраивания создаются с помощью моделей, обученных на массивных текстовых корпусах. Эти модели, такие как Word2Vec¹, GloVe² или BERT³, обучаются, наблюдая за тем, как слова используются в предложениях. Например, «яблоко» и «банан» могут часто появляться в схожих контекстах, связанных с фруктами, едой, привычками питания или диетой, тогда как «собака» чаще появляется в контекстах, связанных с животным миром, домашними животными или активным отдыхом;
- *выявление семантических связей*: в процессе обучения модель учится связывать слова с их контекстом. Это означает, что слова, встречающиеся в схожих контекстах, располагаются ближе друг к другу в пространстве встраивания. Модель не осознает значения слов «яблоко» или «банан»

¹ <https://en.wikipedia.org/wiki/Word2vec>.

² <https://en.wikipedia.org/wiki/GloVe>.

³ [https://en.wikipedia.org/wiki/BERT_\(language_model\)](https://en.wikipedia.org/wiki/BERT_(language_model)).

в человеческом понимании, но замечает, что эти слова имеют схожие модели использования, что отличает их от слова «собака», которое имеет другие ассоциации;

- *многомерное векторное пространство*: в этом пространстве каждое слово представлено как вектор (точка в пространстве). Расстояние между этими точками является мерой смыслового сходства. Такие методы, как косинусное сходство, используются для измерения того, насколько близко или далеко расположены эти векторы. Слова «яблоко» и «банан» будут иметь векторы, расположенные близко друг к другу, а слово «собака» будет расположено ощутимо дальше от первых двух;
- *обучение на совпадениях*: основной принцип заключается в том, что слова со схожим значением будут встречаться в схожих контекстах чаще, чем слова с несвязанными значениями. Этот принцип, часто характеризующийся афоризмом лингвиста Джона Руперта Фёрса «Вы узнаете слово по компании, которая его окружает», позволяет модели встраивания сделать вывод, что «яблоко» и «банан» связаны между собой (оба являются фруктами) и оба совершенно отличаются от «собаки».

Продвинутые примеры встраивания

РЕКОМЕНДАЦИЯ ПОДХОДЯЩЕГО 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.) Вы можете найти тот же набор данных в репозитории GitHub² этой книги и в архиве файлов русского перевода на сайте издательства.

Загрузите набор данных и сохраните его в папке `src/data`.

Пример команд, которыми могут воспользоваться для скачивания набора данных пользователь Linux:

```
# Создаем каталог
mkdir -p src/data

# Скачиваем набор данных
wget https://raw.githubusercontent.com/eon01/Openai_GPT_for_Python_Developers_Files/main/datasets/coffee_reviews_dataset/simplified_coffee.csv.gz \
-O src/data/simplified_coffee.csv.gz

# Распаковываем архив .gz
gunzip -k src/data/simplified_coffee.csv.gz

# Удаляем архив
rm src/data/simplified_coffee.csv.gz
```

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 долларах США);

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

² https://github.com/eon01/Openai_GPT_for_Python_Developers_Files/.

- rating (рейтинг, максимум 100);
- review_date (дата обзора);
- review (текст обзора).

В этом наборе данных нас интересуют отзывы пользователей. Эти отзывы 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 выводим на печать названия сортов кофе, у которых описания наиболее похожи на описание сорта, указанного пользователем.

Например, если у нас есть обзоры кофейных смесей (A, B, C, D, E), пользователь, который вводит название кофейной смеси A, должен получить E в качестве рекомендации, поскольку она имеет наибольшее сходство.

Обзор кофейной смеси A:

Нежный, со сладкими пряными оттенками. В послевкусии соединяются нотки финика и фундука с оттенками кедра.

Обзор кофейной смеси B:

Яркий, насыщенный аромат с оттенком цитрусовых. В аромате и чашке темный шоколад, жареный грецкий орех, апельсиновый цвет, кедр и коричневый сахар. Ярко-сладкий с яркой кислинкой; полный, бархатистый вкус. Послевкусие сосредоточено на темном шоколаде и грецком орехе с кедровым оттенком.

Обзор кофейной смеси C:

Живой, пикантный, с фруктовыми нотками. Малина, макадамия, гардения, бамбук, патока в аромате и чашке. Сладко-терпкая структура с живой кислинкой; легкий, шелковистый вкус. Послевкусие представляет собой восхитительную смесь малины и макадамии, дополненную нотой бамбука.

Обзор кофейной смеси D:

Крепкий, землистый, с оттенком дымности. Черная смородина, фундук, гибискус, дуб, патока в аромате и чашке. Глубоко-сладкий с низкой кислотностью; гладкий, густой вкус. В послевкусии сочетаются черная смородина и фундук с дубовым фоном.

Обзор кофейной смеси E:

Тонкий, деликатно приправленный сладким оттенком. Финик, миндаль, орхидея, кедр, кленовый сироп в аромате и чашке. Сладкая структура с нежной, округлой кислотностью; шелковистый, атласный вкус. Послевкусие представляет собой плавные переливы финика и миндаля с нотками кедра.

Теперь приступим к работе со встраиваниями.

Активируйте свое виртуальное рабочее пространство и установите NLTK:

```
workon openaigtptforpythondevelopers
pip install nltk==3.8.1
```

Набор инструментов для естественного языка (natural language toolkit, NLTK¹) представляет собой набор библиотек и программ для символьной и статистической обработки естественного языка. Он специально разработан для английского языка и реализован на языке программирования Python.

Теперь займемся разработкой кода. Мы немного доработаем код предыдущего примера. Он сохранен в файле `app_43.py`.

```
cat << EOF > src/api_43.py
import os
from openai import OpenAI

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

client = OpenAI(
    api_key=os.environ['API_KEY'],
    organization=os.environ['ORG_ID']
)

def get_embedding(text, model):
    text = text.replace("\n", " ")
    return client.embeddings.create(
        input = [text],
        model=model
    ).data[0].embedding
EOF
```

Теперь создадим новый файл с именем `utils.py`. В нем будут размещены служебные функции, такие как функция `cosine_similarity` из предыдущего примера, а также другие функции, которые нам понадобятся в этой главе.

```
cat << EOF > src/utils.py
import pandas as pd
import numpy as np
import nltk

def cosine_similarity(a, b):
    numerator = np.dot(a, b)
    denominator = np.linalg.norm(a) * np.linalg.norm(b)
    return numerator / denominator

def download_nltk_data():
    # Находим и скачиваем модель токенизатора 'punkt'
    try:
        nltk.data.find('tokenizers/punkt')
    except LookupError:
        nltk.download('punkt')
```

¹ Поскольку пример данных англоязычный и библиотека NLTK предназначена для работы с англоязычными текстами, последующие примеры этой главы будут даны в основном без перевода, в полном соответствии с оригиналом книги. – *Прим. перев.*

```

# Находим и скачиваем корпус 'stopwords'
try:
    nltk.data.find('corpora/stopwords')
except LookupError:
    nltk.download('stopwords')

def preprocess_text(text):
    from nltk.corpus import stopwords
    from nltk.stem import PorterStemmer
    from nltk.tokenize import word_tokenize
    # Токенизируем текст
    tokens = word_tokenize(text)

    # Приводим к нижнему регистру
    tokens = [
        word.lower() for word in tokens
    ]

    # Удаляем знаки пунктуации
    words = [
        word for word in tokens if word.isalpha()
    ]

    # Отфильтровываем стоп-слова
    stop_words = set(
        stopwords.words('english')
    )
    words = [
        word for word in words if word not in stop_words
    ]

    # Стемминг
    stemmer = PorterStemmer()
    stemmed_words = [
        stemmer.stem(word) for word in words
    ]

    return ' '.join(stemmed_words)
EOF

```

Функция `download_nltk_data` обеспечивает доступность необходимых наборов данных и моделей, требуемых библиотекой NLTK, для использования в вашей среде Python. Небольшое пояснение к наборам данных и моделям, которые мы загружаем:

- 'tokenizers/punkt': это модель токенизатора Punkt, используемая для разделения текста на список предложений с применением алгоритма обучения без учителя;
- 'corpora/stopwords': это набор данных «стоп-слов», то есть таких бесполезных и мешающих слов, как «тот», «в», «при», «который» и т. д., которые обычно удаляются на этапе предварительной обработки в задачах NLP.

Функция `preprocess_text` предназначена для предварительной обработки текста путем токенизации, преобразования в нижний регистр, удаления знаков препинания, удаления стоп-слов и стемминга. Вот что делается на каждом шаге:

- токенизация: это процесс разделения текста на список слов или предложений. В нашем случае мы используем токенизатор слов, чтобы разбить текст на список слов;
- преобразование в нижний регистр: этот процесс преобразует все слова в тексте в нижний регистр. Это сделано для уменьшения размера словарного запаса, поскольку такие слова, как «Кофе» и «кофе», обрабатываются токенизатором как разные слова;
- удаление знаков препинания. Этот процесс удаляет пунктуацию из текста. Пунктуация для нашей задачи бесполезна, поэтому удалим ее;
- удаление стоп-слов. Стоп-слова – это такие слова, как «этот», «в», «на», «который» и т. д., которые обычно удаляются на этапе предварительной обработки в задачах NLP. Мы их удаляем, потому что для нашей задачи они бесполезны;
- стемминг: этот процесс преобразует слова в их корневую форму. Например, «бегун» преобразуется в «бег». Это сделано для уменьшения размера словарного запаса, поскольку такие слова, как «бегун» и «бег», обрабатываются токенизатором как разные слова.

Прежде чем продолжить, нужно пояснить разницу между стеммингом и его альтернативой – лемматизацией. И стемминг, и лемматизация – это методы, используемые для сведения слов к их базовой или корневой форме, но они подходят к этой задаче по-разному.

Стемминг упрощает слова, довольно примитивно обрезаю концы, не принимая во внимание контекст или значение слова. Это быстрый и неаккуратный способ добраться до корневой формы; иногда он может быть неточным, поскольку просто удаляет суффиксы на основе общих шаблонов.

С другой стороны, лемматизация – это более сложный процесс, который учитывает значение слова и его грамматическую роль в предложении. Он использует словарь или морфологический анализ слов, чтобы точно привести их к их базовой или словарной форме (лемме).

Если вы запустите следующий код (файл `stemlem.py`), то увидите разницу между стеммингом и лемматизацией:

```
import nltk
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer

# Скачивание необходимых данных NLTK
nltk.download('wordnet')
nltk.download('omw-1.4')

# Инициализация стеммера и лемматизатора
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()
```

```
# Список слов для примера
words = [
    'running',
    'runner',
    'jumps',
    'easily',
    'better'
]

# Процесс стемминга
stemmed_words = [
    stemmer.stem(word) for word in words
]

# Процесс лемматизации в соответствии со спецификацией POS
# (POS: Part of Speech, часть речи)
lemmatized_words = []
for word in words:
    # По умолчанию существительное
    pos_tag = 'n'
    if word in ['better']:
        # Воспринимать этот пример как прилагательное
        pos_tag = 'a'
    elif word in ['running', 'jumps']:
        # Воспринимать этот пример как глагол
        pos_tag = 'v'
    elif word in ['easily']:
        # Воспринимать этот пример как наречие
        pos_tag = 'r'
    else:
        # Считать все остальные примеры существительными
        pos_tag = 'n'
    lemmatized_word = lemmatizer.lemmatize(
        word,
        pos=pos_tag
    )
    lemmatized_words.append(lemmatized_word)

# Вывод результатов на печать
print("Исходные слова: ", words)
print("Стемминг: ", stemmed_words)
print("Лемматизация: ", lemmatized_words)
```

Вывод этого кода будет выглядеть так:

```
Исходные слова: ['running', 'runner', 'jumps', 'easily', 'better']
Стемминг: ['run', 'runner', 'jump', 'easili', 'better']
Лемматизация: ['run', 'runner', 'jump', 'easily', 'good']
```

Как видите, стемминг упростил слово *running* до базовой формы *run*, потому что обнаружил и удалил суффикс *-ing* в соответствии с правилами. Стемминг следует предварительно заданным правилам без понимания контекста или полного морфологического разбора слова.

Лемматизация, в свою очередь, стремится правильно выявить основную форму слова (лемму), распознавая значение и часть речи (pos) в контексте. Поэтому слово *running* будет сокращено до *run* более обоснованно, потому что распознано как глагол благодаря контекстному пониманию словоформ и их грамматической роли. В соответствии с этим подходом слово *better* (лучший) будет приведено к форме *good* (хороший), в то время как стемминг оставит форму *better*.

Стемминг быстрее и проще, но менее точен, чем лемматизация. И наоборот, лемматизация дает более качественный результат, но сложнее и медленнее, чем стемминг.

Перейдем к следующему шагу.

Импортируем созданные нами функции в главный файл.

```
from api import get_embedding
from utils import (
    cosine_similarity,
    download_nltk_data,
    preprocess_text
)
```

Далее получим пользовательский ввод:

```
# Чтение пользовательского ввода
input_coffee_name = input("Enter a coffee name: ")
```

Теперь прочитаем csv-файл в кадр данных Pandas. Обратите внимание, что в данном примере мы читаем первые 50 строк. Вы можете изменить код и убрать `nrows`, чтобы полностью прочитать набор данных.

```
dataset_file_path = os.path.join(
    os.path.dirname(__file__),
    'data',
    'simplified_coffee.csv'
)
# Чтение файла CSV в кадр данных Pandas
# (в этом примере - первые 50 строк)
df = pd.read_csv(
    dataset_file_path,
    nrows=50
)
```

Выполним предварительную обработку текста отзывов:

```
# Preprocess the review text
df['preprocessed_review'] = df['review'].apply(
    preprocess_text
)
```

Теперь нужно получить встраивания для каждого отзыва. Воспользуемся для этого функцией `get_embedding`, которую мы импортировали из `api`, и применим ее к каждому отзыву в кадре данных:

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

Далее нам нужно получить индекс введенного пользователем сорта кофе. Если такого сорта нет в базе данных, прекращаем выполнение программы:

```
# Get the index of the input coffee name
try:
    input_coffee_index = \
        df[df['name'] == \
            input_coffee_name].index[0]
except:
    print("Пожалуйста, введите правильное название сорта кофе.")
    exit()
```

Этот код нуждается в некоторых пояснениях.

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 кадра данных `DataFrame`, которая содержит введенное название сорта 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]` применяется 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 данной строки.

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

```
# Вычисляем косинусное сходство между отзывом на сорт,
# название которого ввел пользователь, и остальными сортами
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 `cosine_similarity()` из модуля `openai.embeddings_utils` для вычисления косинусного сходства между обзором кофе, который выбрал пользователь, и текущим обзором. (Мы уже использовали эту функцию в предыдущем примере.)

Далее 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, то 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 на 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]`, то получили бы 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
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. 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, применяем `tolist()` для преобразования столбца в список. Это дает нам список названий наиболее похожих сортов кофе.

Теперь можно вывести результаты на печать:

```
# Выводим результаты на печать
print(
    "Сорта кофе, наиболее похожие на "
    f"{input_coffee_name}:"
)
for coffee_name in similar_coffee_names:
    print(coffee_name)
```

Так выглядит окончательный код приложения:

```

cat << EOF > src/app_44.py
import os
import pandas as pd
import numpy as np
from api import get_embedding
from utils import (
    cosine_similarity,
    download_nltk_data,
    preprocess_text
)

# Скачивание необходимых данных NLTK
download_nltk_data()

dataset_file_path = os.path.join(
    os.path.dirname(__file__),
    'data',
    'simplified_coffee.csv'
)

# Считывание пользовательского ввода
input_coffee_name = input("Введите название сорта кофе: ")

# Чтение файла CSV в кадр данных Pandas
# (в данном случае первые 50 строк)
df = pd.read_csv(
    dataset_file_path,
    nrows=50
)

# Предварительная обработка текста отзывов
df['preprocessed_review'] = df['review'].apply(
    preprocess_text
)

# Выбранная модель
model = "text-embedding-ada-002"
# Получаем встраивания для каждого отзыва
review_embeddings = []
for review in df['preprocessed_review']:
    review_embeddings.append(
        get_embedding(
            review,
            model=model
        )
    )

# Получаем индекс введенного названия сорта кофе
try:
    input_coffee_index = \

```

```

df[df['name'] == \
input_coffee_name].index[0]
except:
    print("Пожалуйста, введите правильное название сорта кофе.")
    exit()

# Вычисляем косинусное сходство между отзывом на сорт,
# название которого ввел пользователь, и остальными сортами
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
similar_coffee_names = df.iloc[most_similar_indices]\
['name']\
.tolist()

# Выводим результаты на печать
print(
    "Сорта кофе, наиболее похожие на "
    f"{input_coffee_name}:"
)
for coffee_name in similar_coffee_names:
    print(coffee_name)
EOF

```

Запустите скрипт (`python src/app_44.py`) и введите название сорта, который точно есть в базе данных. Вы должны получить следующий результат:

Введите название сорта кофе: Organic Ethiopia Kirite

Сорта кофе, наиболее похожие на Organic Ethiopia Kirite:

```

El Peñon Nicaragua
Colombia David Gomez 100% Caturra
Panama Auromar Estate Geisha Peaberry
Ethiopia Yirgacheffe Natural G1
Ethiopia Shakiso Mormora

```

РАЗРАБОТКА БОЛЕЕ «НЕЧЕТКОГО» ПОИСКА

Потенциальная проблема кода из предыдущего раздела заключается в том, что пользователь должен ввести абсолютно 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 «Пожалуйста, введите правильное название сорта кофе».

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

Вот как можно найти в кадре данных название сорта кофе, *похожее* на введенное название сорта:

```
# Получаем индекс для введенного названия сорта кофе
try:
    # Поиск в кадре данных названия сорта кофе,
    # наиболее 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 "
        f"{df.iloc[input_coffee_index]['name']}. "
        "Будем использовать его для поиска."
    )
except:
    print(
        "Извините, нам не удалось найти похожее название "
        "в нашей базе данных. Попробуйте еще раз."
    )
    exit()
```

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.

¹ https://en.wikipedia.org/wiki/Levenshtein_distance.

Вам не нужно заново реализовывать какой-либо алгоритм самостоятельно, так как большинство из них можно найти в таких библиотеках, как `textdistance`¹.

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

```
# Получаем индекс для введенного названия сорта кофе
try:
    input_coffee_index = df[
        df['name'] == input_coffee_name
    ].index[0]
except IndexError:
    # get the embeddings for each name
    print(
        "Поиск похожего названия сорта..."
    )
    name_embeddings = []
    for name in df['name']:
        name_embeddings.append(
            get_embedding(
                name, model=model
            )
        )
    # Выполняем поиск по косинусному подобию
    # для введенного названия сорта кофе
    input_coffee_embedding = get_embedding(
        input_coffee_name,
        model=model
    )
    _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, нам не удалось найти похожее название "
        "в нашей базе данных. Попробуйте еще раз."
    )
    exit()
```

Таким образом, мы выполняем два поиска по косинусному подобию в одном коде:

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

¹ <https://github.com/life4/textdistance>.

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

Так будет выглядеть окончательный код:

```
cat << EOF > src/app_45.py
import os
import pandas as pd
import numpy as np
from api import get_embedding
from utils import (
    cosine_similarity,
    download_nltk_data,
    preprocess_text
)
# Скачивание необходимых данных NLTK
download_nltk_data()

dataset_file_path = os.path.join(
    os.path.dirname(__file__),
    'data',
    'simplified_coffee.csv'
)

# Считывание пользовательского ввода
input_coffee_name = input("Введите название сорта кофе: ")

# Чтение файла CSV в кадр данных Pandas
# (в данном случае первые 50 строк)
df = pd.read_csv(
    dataset_file_path,
    nrows=50
)

# Preprocess the review text
df['preprocessed_review'] = df['review'].apply(
    preprocess_text
)

# Выбранная модель
model = "text-embedding-ada-002"

# Получаем встраивания для каждого отзыва
review_embeddings = []
for review in df['preprocessed_review']:
    review_embeddings.append(
        get_embedding(
            review,
            model=model
        )
    )

# Получаем индекс введенного названия сорта кофе
```

```
try:
    input_coffee_index = df[
        df['name'] == input_coffee_name
    ].index[0]
except IndexError:
    # Получаем встраивания для каждого названия
    print(
        "Поиск похожего названия сорта..."
    )
    name_embeddings = []
    for name in df['name']:
        name_embeddings.append(
            get_embedding(
                name, model=model
            )
        )

    # Выполняем поиск по косинусному подобию
    # для введенного названия сорта кофе
    input_coffee_embedding = get_embedding(
        input_coffee_name,
        model=model
    )
    _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, нам не удалось найти похожее название "
        "в нашей базе данных. Попробуйте еще раз."
    )
    exit()

# Вычисляем косинусное сходство между отзывом на сорт,
# название которого ввел пользователь, и остальными сортами
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
similar_coffee_names = df.iloc[most_similar_indices]\
['name']\
.tolist()

# Выводим результаты на печать
print(
    "Сорта кофе, наиболее похожие на "
    f"{input_coffee_name}:"
)
for coffee_name in similar_coffee_names:
    print(coffee_name)
EOF

```

Чтобы протестировать нечеткий поиск, запустите приложение с помощью команды `python src/app_45.py` и введите *Ethiopian Kirite* вместо *Organic Ethiopia Kirite*. Второе название существует в наборе данных, а первое – нет. Результат поиска должен быть таким же, как если бы вы ввели *Organic Ethiopia Kirite*.

Введите название сорта кофе: Ethiopian Kirite
Поиск похожего названия сорта...

Сорта кофе, наиболее похожие на Ethiopian Kirite:

```

El Peñon Nicaragua
Colombia David Gomez 100% Caturra
Panama Auromar Estate Geisha Peaberry
Ethiopia Yirgacheffe Natural G1
Ethiopia Shakiso Mormora

```

ПРОГНОЗИРОВАНИЕ КАТЕГОРИИ НОВОСТЕЙ: КЛАССИФИКАЦИЯ С ПОМОЩЬЮ ВСТРАИВАНИЯ

В этом примере будет представлен классификатор новостей, который предсказывает категорию новостной статьи без использования примеров.

Классификатор, не использующий примеры (zero-shot classifier), означает модель или систему классификации, которая может правильно распределять данные по классам, которых она не видела во время обучения. Термин «не использующий примеры» указывает на то, что модель не обучалась на этих конкретных классах (т. е. не видела их примеры), но она все равно может делать прогнозы по ним. Это отличается от типичной парадигмы обучения с учителем, где модель обучается на конкретных классах и может прогнозировать только эти классы.

Начнем с определения списка категорий:

```
from api import get_embedding
from utils import cosine_similarity

categories = [
    'U.S. NEWS',
    'COMEDY',
    'PARENTING',
    'WORLD NEWS',
    'CULTURE & ARTS',
    'TECH',
    'SPORTS'
]
```

Теперь определим функцию `classify_sentence`, которая относит предложение к одной из вышеупомянутых категорий.

Определение функции классификации предложений

```
def classify_sentence(sentence, model):
    # Получаем встраивание предложения
    sentence_embedding = get_embedding(
        sentence,
        model=model
    )
    # Вычисляем сходство между
    # предложением и каждой категорией
    similarity_scores = {}
    for category in categories:
        category_embeddings = get_embedding(
            category,
            model=model
        )
        similarity_scores[
            category
        ] = cosine_similarity(
            sentence_embedding,
            category_embeddings
        )
    # Возвращаем категорию с наибольшим
    # значением сходства
    return max(
        similarity_scores,
        key=similarity_scores.get
    )
```

Вот что делает каждая часть этой функции:

- 1) `sentence_embedding = get_embedding(sentence, model=model)`: в этой строке используется функция OpenAI `get_embedding` для нахождения встраивания входного предложения. Аргумент `model=model` указывает, какую модель OpenAI использовать для встраивания;
- 2) `category_embeddings = get_embedding(category, model=model)`: здесь 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 (катеорию) с максимальным значением (оценкой сходства) в словаре `similarity_scores`.

Теперь мы можем классифицировать несколько предложений с помощью функции `classify_sentence`:

```
# 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",
]
```

```
model = "text-embedding-ada-002"

for sentence in sentences:
    category = classify_sentence(
        sentence,
        model=model
    )
    print(f"'{sentence[:50]}..' => {category}")
    print()
```

Так выглядит окончательный код примера:

```
cat << EOF > src/app_46.py
from api import get_embedding
from utils import cosine_similarity

categories = [
    'U.S. NEWS',
    'COMEDY',
    'PARENTING',
    'WORLD NEWS',
    'CULTURE & ARTS',
    'TECH',
    'SPORTS'
]

# Определение функции классификации предложений
def classify_sentence(sentence, model):
    # Получаем встраивание предложения
    sentence_embedding = get_embedding(
        sentence,
        model=model
    )
    # Вычисляем сходство между
    # предложением и каждой категорией
    similarity_scores = {}
    for category in categories:
        category_embeddings = get_embedding(
            category,
            model=model
        )
        similarity_scores[
            category
        ] = cosine_similarity(
            sentence_embedding,
            category_embeddings
        )
    # Возвращаем категорию с наибольшим
    # значением сходства
    return max(
        similarity_scores,
        key=similarity_scores.get
    )
```

```

# Классифицируем предложения
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",
]

model = "text-embedding-ada-002"

for sentence in sentences:
    category = classify_sentence(
        sentence,
        model=model
    )
    print(f"'{sentence[:50]}..' => {category}")
    print()

```

EOF

После выполнения кода примера (`python src/app_46.py`) вывод должен выглядеть примерно так:

```
'1 dead and 3 injured in El Paso..' category is => WORLD NEWS
'Director Owen Kline Calls Funny..' category is => COMEDY
'15 spring break ideas for families..' category is => PARENTING
'The US is preparing to send more troops..' category is => WORLD NEWS
'Bruce Willis' 'condition has progressed'.. category is => WORLD NEWS
'Get an inside look at Universal's new..' category is => WORLD NEWS
'Barcelona 2-2 Manchester United: Marcus..' category is => SPORTS
'Chicago bulls win the NBA championship..' category is => SPORTS
'The new iPhone 12 is now available..' category is => TECH
'Scientists discover a new dinosaur..' category is => WORLD NEWS
'The new coronavirus vaccine is now..' category is => WORLD NEWS
'The new Star Wars movie is now..' category is => WORLD NEWS
'Amazon stock hits a new record..' category is => WORLD NEWS
```

В следующем примере мы продолжим использовать этот код.

ОЦЕНКА ТОЧНОСТИ КЛАССИФИКАТОРА

В этом примере мы оценим точность классификатора, не использующего примеры, который мы разработали в предыдущем разделе.

Сначала загрузим набор данных из Kaggle¹ и сохраним его в файл `src/data/news.json`. Этот файл также имеется в архиве на сайте издательства русского перевода и в репозитории оригинала книги².

Команды Linux для скачивания файла данных:

```
# Создаем каталог
mkdir -p src/data

# Скачиваем набор данных
wget https://raw.githubusercontent.com/\
eon01/Openai_GPT_for_Python_Developers_Files/\
main/datasets/news_category_dataset/news.json.zip \
-O src/data/news.json.zip
```

¹ <https://www.kaggle.com/datasets/rmisra/news-category-dataset>.

² https://github.com/eon01/Openai_GPT_for_Python_Developers_Files.

```
# Извлекаем данные в каталог src/data
unzip src/data/news.json.zip -d src/data
```

```
# Удаляем архив
rm src/data/news.json.zip
```

Этот набор данных содержит около 210 тыс. заголовков новостей с 2012 по 2022 г. от HuffPost¹. Набор данных классифицирует заголовки каждой статьи по категориям.

Все категории можно извлечь из набора данных при помощи следующего кода:

```
cat << EOF > src/extract_categories.py
categories = set()
with open('src/data/news.json', 'r') as file:
    for line in file:
        data = json.loads(line)
        categories.add(data['category'])
categories = list(categories)
EOF
```

Мы будем использовать функцию `sklearn.metrics.precision_score`, которая вычисляет показатель точности.

Функция `precision_score` предназначена для вычисления точности предсказания категории. *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.

Оценку точности можно вычислить при помощи следующего кода:

```
def evaluate_precision(categories):
    # Загружаем набор данных
    df = pd.read_json(
        "src/data/news.json",
        lines=True
    ).head(20)

    y_true = []
    y_pred = []

    model = "text-embedding-ada-002"

    # Классифицируем каждое предложение
    for _, row in df.iterrows():
        real_category = row['category']
        predicted_category = classify_sentence(
            row['headline'],
            model=model
        )
```

¹ <https://www.huffingtonpost.com/>.

```

y_true.append(real_category)
y_pred.append(predicted_category)

if real_category != predicted_category:
    print(
        "Некорректный прогноз: "
        f"{row['headline'][:50]}...\n"
        f"Истина: {real_category[:20]}\n"
        f"Прогноз: {predicted_category[:20]}"
    )
else:
    print(
        "Корректный прогноз: "
        f"{row['headline'][:50]}...\n"
        f"Истина: {real_category[:20]}\n"
        f"Прогноз: {predicted_category[:20]}"
    )

# Вычисление оценки прогноза
return precision_score(
    y_true,
    y_pred,
    average='micro',
    labels=categories
)

```

Давайте посмотрим, что делает каждая строка.

1. `df = pd.read_json("src/data/news.json", lines=True).head(20)`: эта строка считывает первые 20 записей набора данных `news.json` в кадр данных Pandas. Аргумент `lines=True` указывает, 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 большее их количество.
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 кадре данных.
4. `real_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(real_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 документация¹ объясняет разницу между этими вариантами среднего значения.

В целом функция `evaluate_precision` загружает небольшое подмножество набора данных `news.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:

```
cat << EOF > src/app_47.py
from api import get_embedding
from utils import cosine_similarity
import pandas as pd
from sklearn.metrics import precision_score
import json

# Находим все категории (уникальные значения) в наборе
categories = set()
with open('src/data/news.json', 'r') as file:
    for line in file:
        data = json.loads(line)
        categories.add(data['category'])

categories = list(categories)

# Определение функции классификации предложений
def classify_sentence(sentence, model):
    # Получаем встраивание предложения
    sentence_embedding = get_embedding(
        sentence,
        model=model
    )
    # Вычисляем сходство между
    # предложением и каждой категорией
    similarity_scores = {}
    for category in categories:
        category_embeddings = get_embedding(
            category,
            model=model
        )
        similarity_scores[
            category
        ] = cosine_similarity(
            sentence_embedding,
            category_embeddings
        )
    # Возвращаем категорию с наибольшим
    # значением сходства
    return max(
        similarity_scores,
```

¹ https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html.

```
        key=similarity_scores.get
    )

def evaluate_precision(categories):
    # Загружаем набор данных
    df = pd.read_json(
        "src/data/news.json",
        lines=True
    ).head(20)

    y_true = []
    y_pred = []

    model = "text-embedding-ada-002"

    # Классифицируем каждое предложение
    for _, row in df.iterrows():
        real_category = row['category']
        predicted_category = classify_sentence(
            row['headline'],
            model=model
        )

        y_true.append(real_category)
        y_pred.append(predicted_category)

    if real_category != predicted_category:
        print(
            "Некорректный прогноз: "
            f"{row['headline'][:50]}...\n"
            f"Истина: {real_category[:20]}\n"
            f"Прогноз: {predicted_category[:20]}"
        )
    else:
        print(
            "Корректный прогноз: "
            f"{row['headline'][:50]}...\n"
            f"Истина: {real_category[:20]}\n"
            f"Прогноз: {predicted_category[:20]}"
        )

    # Вычисление оценки прогноза
    return precision_score(
        y_true,
        y_pred,
        average='micro',
        labels=categories
    )

# Вычисляем точность классификатора
precision = evaluate_precision(categories)
print(f"Точность: {precision}")
EOF
```

После запуска приведенного выше кода с помощью команды `python src/app_47.py` вы должны увидеть оценку точности. Если показатель точности равен 0.6, это означает, что классификатор смог правильно предсказать категорию 60 % заголовков в наборе данных. Если показатель точности равен 0.8, это означает, что классификатор смог правильно предсказать категорию 80 % заголовков в наборе данных и т. д.

Здесь очень важно понимать, что точность сама по себе не имеет принципиально определенного значения. Результат связан с применяемым источником данных, которым в нашем случае является набор данных `news.json`. Использование другого набора данных (или другого подмножества того же набора данных) даст иной показатель точности.

Точность приложений классификатора в различных сценариях

Высокая точность классификатора без обучения на примерах важна в сценариях, где ложные срабатывания являются дорогостоящими или нежелательными. Вот несколько примеров таких применений:

- **медицинская диагностика.** Представьте себе классификатор, который может диагностировать заболевания, не встречавшиеся во время обучения. Точность здесь имеет решающее значение, поскольку ложноположительный результат (ошибочный диагноз, выставленный здоровому человеку) может привести к ненужному стрессу, лечению и затратам;
- **модерация контента.** Платформы социальных сетей используют классификаторы для выявления вредоносного контента. Высокая точность важна, поскольку неправильная классификация добропорядочного контента как вредного (ложное срабатывание) может привести к цензуре и недовольству пользователей;
- **обнаружение финансового мошенничества.** Допустим, классификатор работает в банковской системе обнаружения мошенничества. Высокая точность означает, что он точно выявляет мошеннические транзакции, не беспокоя клиентов ложными тревогами (когда законные транзакции помечаются как мошенничество), обеспечивая доверие клиентов;
- **фильтрация спама в электронной почте.** Для фильтрации спама в электронной почте тоже широко применяются классификаторы без обучения на примерах. Точность имеет решающее значение, поскольку пользователям очень не нравится, когда важные электронные письма ошибочно классифицируются как спам (ложные срабатывания);
- **системы рекомендации товаров.** В этом сценарии классификатор без обучения на примерах выступает в роли личного помощника по покупкам, который рекомендует товары. Высокая точность гарантирует, что он рекомендует только подходящие товары;
- **автоматизированная поддержка клиентов.** В сценарии, где для маршрутизации запросов клиентов в соответствующие отделы используется классификатор без обучения на примерах, высокая точность гарантирует правильную категоризацию запросов;

- **системы, критически важные для безопасности.** Представьте себе классификатор в автономном транспортном средстве, идентифицирующий объекты на дороге. Высокая точность жизненно важна, чтобы избежать ложных срабатываний (например, ошибочного восприятия бумажного пакета как камня), которые могут привести к ненужным или опасным маневрам.

В каждом из приведенных выше примеров цена ложного срабатывания высока, что делает точность критически важным показателем качества классификатора.

Тонкая настройка и передовые методы работы

ОБУЧЕНИЕ НА ОГРАНИЧЕННЫХ ПРИМЕРАХ

Большие языковые модели GPT-3 и GPT-4 были предварительно обучены с использованием огромного объема данных, состоящих из миллиардов слов. Благодаря этому они стали невероятно мощным инструментом, способным быстро осваивать новые задачи всего на нескольких примерах. Этот прием называется *обучением на ограниченных примерах* (few-shot learning). Он стал революционным достижением в области искусственного интеллекта.

Неудивительно, что появилось множество мощных проектов¹ для обучения моделей с открытым исходным кодом. Для реализации небольших учебных проектов можно использовать несколько надежных библиотек Python:

- Pytorch – Torchmeta²: набор расширений и загрузчиков данных для быстрого обучения и метаобучения в PyTorch³;
- Few Shot⁴: репозиторий с чистым, легко читаемым и проверенным кодом для воспроизведения исследований по обучению на ограниченных примерах;
- FewRel⁵: 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)⁶: содержит официальную реализацию метода Simple Few-Shot Object Detection⁷;
- Meta Transfer Learning⁸: 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, дабы узнать, как модифицировать базовую обучаемую модель под новую задачу, для которой доступны лишь несколько размеченных примеров;

¹ <https://github.com/topics/few-shot-learning>.

² <https://tristandeleu.github.io/pytorch-meta/>.

³ <https://pytorch.org/>.

⁴ <https://github.com/oscarknagg/few-shot>.

⁵ <https://github.com/thunlp/FewRel>.

⁶ <https://github.com/ucbdrive/few-shot-object-detection>.

⁷ <https://arxiv.org/abs/2003.06957>.

⁸ <https://github.com/yaoyao-liu/meta-transfer-learning>.

- прототипные сети в наборе данных Omniglot¹: реализация метода из статьи *Prototypical Networks for Few-shot Learning* на основе Pytorch;
- ...многие другие.

Возможность обучения всего на нескольких примерах позволяет моделям GPT быстро понимать предоставленные пользователем инструкции даже с минимальными данными. Другими словами, большие языковые модели GPT-3 и GPT-4 можно запрограммировать на выполнение задач, используя всего несколько примеров в качестве входных данных. Это открывает новый мир безграничных возможностей для приложений, управляемых ИИ.

УЛУЧШЕННОЕ ОБУЧЕНИЕ НА ОГРАНИЧЕННЫХ ПРИМЕРАХ

Тонкая настройка (fine tuning) – это логическое развитие обучения с ограниченными примерами, но на гораздо большем количестве примеров, чем может уместиться в одном промпте. Данный подход можно использовать для достижения лучших результатов в широком круге задач:

- увеличение количества используемых примеров;
- повышение точности результатов;
- расширение круга выполняемых задач.

После тонкой настройки модели вам больше не нужно приводить обучающие примеры при каждом запросе.

Этот подход экономически эффективен и ускоряет запросы: вам нужно предоставлять примеры только тогда, когда вы стремитесь улучшить модель, после чего вы можете работать с ней, используя только промпты. Не забывайте: чем длиннее ваш промпт, тем больше будет счет от OpenAI.

Тонкую настройку поддерживают следующие модели:

- GPT-3.5-Turbo-1106;
- GPT-3.5-Turbo-0613;
- Babbage-002;
- Davinci-002;
- GPT-4-0613.

В этот список могут быть внесены изменения. В будущем OpenAI планирует представить новые модели, а некоторые вывести из эксплуатации. Среди стабильных моделей GPT-3.5-Turbo-1106 является самой мощной, и OpenAI рекомендует использовать ее для тонкой настройки. Некоторые модели, например GPT-4-0613, все еще находятся на стадии эксперимента.

ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ ТОНКОЙ НАСТРОЙКИ

Теперь перейдем к практической части и рассмотрим реальные примеры.

Обратите внимание, что этот раздел является лишь введением, а всю основную сложную работу мы будем делать позже. Сейчас вам достаточно понять общие принципы, но позже мы углубимся в более сложный пример.

Начнем с активации виртуальной среды Python для разработки, а затем экспортируем ключ API OpenAI:

¹ <https://github.com/cnielly/prototypical-networks-omniglot>.

```
workon openaiptforpythondevelopers
export OPENAI_API_KEY=<OPENAI_API_KEY>
```

Если вы работаете в среде ОС Windows, вместо команды `export` введите `set`. Теперь сгенерируем файл JSON. Вот пример такого файла:

```
cat << EOF > src/data.jsonl
{"messages": [{"role": "system", "content": "You are a smart home assistant"}, {"role": "user", "content": "When do I have to start the heater?"}, {"role": "assistant", "content": "Every day in the morning at 7AM. You should stop it at 2PM"}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "Where is the garage remote control?"}, {"role": "assistant", "content": "Next to the yellow door, on the key ring"}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "Is it necessary to program the scent diffuser every day?"}, {"role": "assistant", "content": "The scent diffuser is already programmed, you just need to recharge it when its battery is low"}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "How do I reset the WiFi router?"}, {"role": "assistant", "content": "Unplug the router, wait for 30 seconds, then plug it back in."]}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "How do I connect my phone to the smart TV?"}, {"role": "assistant", "content": "Go to settings on your phone, select 'Screen Mirroring', and choose the smart TV from the list."]}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "What's the Wi-Fi password?"}, {"role": "assistant", "content": "The Wi-Fi password is 'HomeNetwork123'."]}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "How to hard-reset the smart TV?"}, {"role": "assistant", "content": "Press and hold the power button on the remote for 10 seconds."]}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "How to call the police?"}, {"role": "assistant", "content": "Dial 911 on your phone."]}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "How to change the wifi password?"}, {"role": "assistant", "content": "Go to the router settings, select 'Wireless', and change the password."]}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"role": "user", "content": "How to change the channel on the TV?"}, {"role": "assistant", "content": "Press the 'Channel Up' or 'Channel Down' button on the remote."]}]}
EOF
```

Файл JSONL, то есть 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 (по наблюдениям OpenAI).

Мы используем всего 10 строк в нашем файле JSONL, поэтому полученная модель будет работать не очень хорошо, но сейчас мы просто хотим посмотр-

реть, как это работает на практике. Имейте в виду, что обучающие файлы должны содержать не менее 10 примеров, так что в этом упражнении мы работаем с минимально возможным объемом данных.

Начнем с выгрузки файла в модель при помощи кода Python:

```
from api import client
import os, sys

# Определяем путь к файлу
file_path = os.path.join(
    os.path.dirname(__file__),
    'data.jsonl'
)

# Выгружаем файл
uploaded = client.files.create(
    file=open(
        file_path,
        "rb"
    ),
    purpose="fine-tune"
)
file_id = uploaded.id
```

Теперь создадим задание тонкой настройки:

```
model = "gpt-3.5-turbo"
fine_tune_job = client.fine_tuning.jobs.create(
    training_file=file_id,
    model=model,
)

print("Выполняется проверка файла")
while fine_tune_job.status == "validating_files":
    fine_tune_job = client.fine_tuning.jobs.retrieve(
        fine_tune_job.id
    )
    sys.stdout.write("...")
    sys.stdout.flush()

print("Выполняется тонкая настройка")
while fine_tune_job.status == "running" \
    or fine_tune_job.status == "queued":
    fine_tune_job = client.fine_tuning.jobs.retrieve(
        fine_tune_job.id
    )
    sys.stdout.write("...")
    sys.stdout.flush()

print("Тонкая настройка завершена")
print("Название новой модели: " \
    + fine_tune_job.fine_tuned_model)
```

После объединения всех шагов окончательный код выглядит так:

```
cat << EOF > src/fine_tuning.py
from api import client
import os, sys

# Определяем путь к файлу
file_path = os.path.join(
    os.path.dirname(__file__),
    'data.jsonl'
)

# Выгружаем файл
uploaded = client.files.create(
    file=open(
        file_path,
        "rb"
    ),
    purpose="fine-tune"
)
file_id = uploaded.id

model = "gpt-3.5-turbo"
fine_tune_job = client.fine_tuning.jobs.create(
    training_file=file_id,
    model=model,
)

print("Выполняется проверка файла")
while fine_tune_job.status == "validating_files":
    fine_tune_job = client.fine_tuning.jobs.retrieve(
        fine_tune_job.id
    )
    sys.stdout.write("...")
    sys.stdout.flush()

print("Выполняется тонкая настройка")
while fine_tune_job.status == "running" \
    or fine_tune_job.status == "queued":
    fine_tune_job = client.fine_tuning.jobs.retrieve(
        fine_tune_job.id
    )
    sys.stdout.write("...")
    sys.stdout.flush()

print("Тонкая настройка завершена")
print("Название новой модели: " \
    + fine_tune_job.fine_tuned_model)
EOF
```

Теперь запустим этот код:

```
python src/fine_tuning.py
```

После завершения процесса тонкой настройки, который может занять какое-то время, в окне терминала вы увидите название вновь созданной модели. OpenAI также отправит вам письмо с аналогичной информацией.

Hi LearningGPT,

Your fine-tuning job `ftjob-kEz0R3TkEmZmuUWvt2LIP4yE` has successfully completed, and a new model `ft:gpt-3.5-turbo-0613:learninggpt::8mQaoUsf` has been created for your use.

Try it out on the [OpenAI Playground](#), view the training results in the [fine-tuning UI](#), or integrate it into your application using the [Chat Completions](#) or [Legacy Completions](#) API.

Thank you for building on the OpenAI platform,
The OpenAI team

Письмо с информацией о тонко настроенной модели

Далее экспортируем имя новой модели:

```
export FINE_TUNED_MODEL="<FINE_TUNED_MODEL>"
```

Теперь вы можете воспользоваться новой моделью в интерактивной сессии, созданной с помощью пакета `click`, как мы уже делали в предыдущей главе:

```
cat << EOF > src/app.py
from api import client
import click

model = "$FINE_TUNED_MODEL"
base_messages = [
    {
        "role": "system",
        "content":
            "Ты умный домашний помощник."
    }
]

while True:
    messages = base_messages.copy()

    # Чтение пользовательского ввода
    request = input(
        click.style(
            "Ввод: (введите 'exit' для выхода): ",
            fg="green"
        )
    )
)
```

```

if request.lower() in ["exit", "quit"]:
    break

# Добавление пользовательского ввода к сообщению
messages.append(
    {
        "role": "user",
        "content": f"{request}"
    }
)

# Отправка сообщения в API
response = client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=200,
    temperature=0,
)

# Получение ответа
content = response.choices[0].message.content.strip()

# Печать полученного ответа
click.echo(
    click.style("Вывод: ", fg="yellow") + content
)

click.echo()
EOF

```

ПОЛЕЗНЫЕ ПРИЕМЫ ТОНКОЙ НАСТРОЙКИ

Выбор модели

В настоящее время наилучшим вариантом для тонкой настройки является GPT-3. Но это положение дел наверняка изменится в будущем.

Проверка набора данных

Официальная документация предлагает подробную информацию о подготовке данных и передовом опыте¹. Хотя этот раздел в значительной степени основан на упомянутой документации, мы углубимся в более конкретные детали.

Имейте в виду, что существуют устаревшие документы и руководства по тонкой настройке. Хотя они остаются в силе, это не обязательно лучший подход из возможных. Одним из ключевых отличий является то, что набор данных для устаревших моделей не обязательно должен быть в диалоговом формате.

Пример устаревшего набора данных:

¹ https://cookbook.openai.com/examples/chat_finetuning_data_prep.

```

{"prompt": "When do I have to start the heater?", "completion": "Every day in the morn\
ing at 7AM. You should stop it at 2PM"}
{"prompt": "Where is the garage remote control?", "completion": "Next to the yellow do\
or, 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 batt\
ery is low"}

```

Пример набора данных в новом формате:

```

{"messages": [{"role": "system", "content": "You are a smart home assistant"}, {"rol\
e": "user", "content": "When do I have to start the heater?"}, {"role": "assistant", \
"content": "Every day in the morning at 7AM. You should stop it at 2PM"}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"ro\
le": "user", "content": "Where is the garage remote control?"}, {"role": "assistant" \
, "content": "Next to the yellow door, on the key ring"}]}
{"messages": [{"role": "system", "content": "You are a smart home assistant."}, {"ro\
le": "user", "content": "Is it necessary to program the scent diffuser every day?"}, \
{"role": "assistant", "content": "The scent diffuser is already programmed, you jus\
t need to recharge it when its battery is low"}]}

```

Последний формат оптимизирован для диалоговых моделей, вот почему для тонкой настройки мы используем GPT-3.5-Turbo.

Другое отличие заключается в том, что устаревшие тонко настроенные модели создаются с помощью командной строки:

```

# Эта команда создает новый файл ("data_prepared.jsonl")
openai tools fine_tunes.prepare_data -f data.json
# Эта команда создает задачу тонкой настройки
openai api fine_tunes.create -t "data_prepared.jsonl" -m curie

```

Таковы основные отличия тонкой настройки устаревших и новых моделей. Вы должны научиться различать размещенные на различных ресурсах описания процедуры тонкой настройки для старых и новых моделей.

Давайте создадим скрипт, который загружает и проверяет данные.

Как обычно, начнем с импорта библиотек:

```

import json
from collections import defaultdict
import os

```

Укажем путь к файлу данных:

```

file_path = os.path.join(
    os.path.dirname(__file__),
    'data.jsonl'
)

```

Затем загрузим данные и проверим формат файла:

```

# Проверка формата файла набора данных
with open(file_path, 'r', encoding='utf-8') as f:
    try
        dataset = [json.loads(line) for line in f]
    except:

```

```

    raise ValueError(
        "Файл набора данных должен быть в формате JSONL"
    )

```

Теперь мы можем проверить формат самого набора данных. Прежде всего набор данных должен содержать не менее 10 примеров.

```

# Проверка размера набора данных
size = len(dataset)
if size < 10:
    raise ValueError(
        "Набор данных должен содержать не менее 10 примеров"
    )

```

Если набор действительно содержит не менее 10 примеров, следующим шагом мы проверяем каждую запись. Правильная запись набора данных должна иметь такую форму:

```

{
  "messages":
  [
    {
      "role": "system",
      "content":
        "You are a smart home assistant."
    }, {
      "role": "user",
      "content":
        "Is it necessary to program the "
        "scent diffuser every day?"
    }, {
      "role": "assistant",
      "content":
        "The scent diffuser is already "
        "programmed, you just need to "
        "recharge it when its battery is low"
    }
  ]
}

```

Мы воспользуемся функциями OpenAI для выполнения следующих проверок:

- *проверка типа данных*: является ли каждая запись в наборе данных словарем (dict);
- *наличие списка сообщений*: эта проверка гарантирует наличие списка message в каждой записи;
- *проверка ключей сообщений*: гарантия того, что каждое сообщение в списке сообщений содержит ключи role и content;
- *нераспознанные ключи в сообщениях*: регистрируется факт ошибки данных, если в сообщении есть ключи, отличные от role, content и name;
- *проверка роли*: подтверждает, что role имеет только значения system, user или assistant;
- *проверка контента*: проверяется, содержит ли content только текстовые данные и являются ли они строковыми.

Так выглядит код, при помощи которого происходит проверка:

```
cat << EOF > src/dataset_validation.py
# Проверка набора данных
format_errors = defaultdict(int)

for line in dataset:
    # Проверка типа данных:
    # проверка каждой записи в словаре
    if not isinstance(line, dict):
        format_errors["data_type"] += 1
        continue

    # Список сообщений:
    # проверка наличия списка 'messages'
    messages = line.get("messages", None)
    if not messages:
        format_errors["missing_messages_list"] += 1
        continue

    for message in messages:
        # Проверка ключей сообщений:
        # гарантия, что сообщение содержит ключи 'role' и 'content'
        if "role" not in message or "content" not in message:
            format_errors["message_missing_key"] += 1

        # Список допустимых ключей сообщения
        valid_keys = (
            "role",
            "content",
            "name",
            "function_call"
        )

        # Нераспознанные ключи сообщения:
        # наличие ключей, которые не являются допустимыми
        if any(k not in valid_keys for k in message):
            format_errors["message_unrecognized_key"] += 1

        # Допустимые роли, которые может иметь сообщение
        valid_roles = (
            "system",
            "user",
            "assistant",
            "function"
        )

        # Проверка ролей:
        # содержит ли 'role' одну из valid_roles
        if message.get(
            "role",
            None
        ) not in valid_roles:
            format_errors["unrecognized_role"] += 1
```

```

content = message.get(
    "content",
    None
)
function_call = message.get(
    "function_call",
    None
)

# Проверка контента:
# является ли 'content' строкой текста
# Также проверка наличия 'content' или 'function_call'
if (not content and not function_call) or \
    not isinstance(content, str):
    format_errors["missing_content"] += 1

# Наличие сообщения помощника:
# наличие как минимум одного сообщения помощника
if not any(
    message.get(
        "role",
        None
    ) == "assistant"
    for message in messages
):
    format_errors["example_missing_assistant_message"] += 1

# Вывод ошибок на печать (при наличии)
if format_errors:
    print("Найдены ошибки:")
    for k, v in format_errors.items():
        print(f"{k}: {v}")
    raise ValueError(
        "Набор данных содержит ошибки"
    )
EOF

```

За основу примера был взят код из сборника практических рецептов «Data preparation and analysis for chat model fine-tuning»¹.

Окончательный вариант кода примера проверки данных представлен в следующем файле:

```

cat << EOF > src/data_analysis.py
import json
from collections import defaultdict
import os

file_path = os.path.join(
    os.path.dirname(__file__),
    'data.jsonl'
)

```

¹ https://cookbook.openai.com/examples/chat_finetuning_data_prep.

```
# Проверка формата файла данных
with open(file_path, 'r', encoding='utf-8') as f:
    try:
        dataset = [json.loads(line) for line in f]
    except:
        raise ValueError(
            "Файл набора данных должен быть в формате JSONL"
        )

# Проверка размера набора данных
size = len(dataset)
if size < 10:
    raise ValueError(
        "Набор данных должен содержать не менее 10 примеров"
    )

# Проверка набора данных
format_errors = defaultdict(int)

for line in dataset:
    # Проверка типа данных:
    # проверка каждой записи в словаре
    if not isinstance(line, dict):
        format_errors["data_type"] += 1
        continue

    # Список сообщений:
    # проверка наличия списка 'messages'
    messages = line.get("messages", None)
    if not messages:
        format_errors["missing_messages_list"] += 1
        continue

    for message in messages:
        # Проверка ключей сообщений:
        # гарантия, что сообщение содержит ключи 'role' и 'content'
        if "role" not in message or "content" not in message:
            format_errors["message_missing_key"] += 1

        # Список допустимых ключей сообщения
        valid_keys = (
            "role",
            "content",
            "name",
            "function_call"
        )

        # Нераспознанные ключи сообщения:
        # наличие ключей, которые не являются допустимыми
        if any(k not in valid_keys for k in message):
            format_errors["message_unrecognized_key"] += 1
```

```

# Допустимые роли, которые может иметь сообщение
valid_roles = (
    "system",
    "user",
    "assistant",
    "function"
)

# Проверка ролей:
# содержит ли 'role' одну из valid_roles
if message.get(
    "role",
    None
) not in valid_roles:
    format_errors["unrecognized_role"] += 1

content = message.get(
    "content",
    None
)
function_call = message.get(
    "function_call",
    None
)

# Проверка контента:
# является ли 'content' строкой текста
# Также проверка наличия 'content' или 'function_call'
if (not content and not function_call) or \
    not isinstance(content, str):
    format_errors["missing_content"] += 1

# Наличие сообщения помощника:
# наличие как минимум одного сообщения помощника
if not any(
    message.get(
        "role",
        None
    ) == "assistant"
    for message in messages
):
    format_errors["example_missing_assistant_message"] += 1

# Вывод ошибок на печать (при наличии)
if format_errors:
    print("Найдены ошибки:")
    for k, v in format_errors.items():
        print(f"{k}: {v}")
    raise ValueError(
        "Набор данных содержит ошибки"
    )
EOF

```

Объединяя скрипты проверки и обучения, мы получаем окончательный код примера этой главы:

```
cat << EOF > src/fine_tuning_final.py
from api import client
import os, sys, json
from collections import defaultdict

file_path = os.path.join(
    os.path.dirname(__file__),
    'data.jsonl'
)

# Проверка формата файла данных
with open(file_path, 'r', encoding='utf-8') as f:
    try:
        dataset = [json.loads(line) for line in f]
    except:
        raise ValueError(
            "Файл набора данных должен быть в формате JSONL"
        )

# Проверка размера набора данных
size = len(dataset)
if size < 10:
    raise ValueError(
        "Набор данных должен содержать не менее 10 примеров"
    )

# Проверка набора данных
format_errors = defaultdict(int)

for line in dataset:
    # Проверка типа данных:
    # проверка каждой записи в словаре
    if not isinstance(line, dict):
        format_errors["data_type"] += 1
        continue

    # Список сообщений:
    # проверка наличия списка 'messages'
    messages = line.get("messages", None)
    if not messages:
        format_errors["missing_messages_list"] += 1
        continue

    for message in messages:
        # Проверка ключей сообщений:
        # гарантия, что сообщение содержит ключи 'role' и 'content'
        if "role" not in message or "content" not in message:
            format_errors["message_missing_key"] += 1
```

```

# Список допустимых ключей сообщения
valid_keys = (
    "role",
    "content",
    "name",
    "function_call"
)

# Нераспознанные ключи сообщения:
# наличие ключей, которые не являются допустимыми
if any(k not in valid_keys for k in message):
    format_errors["message_unrecognized_key"] += 1

# Допустимые роли, которые может иметь сообщение
valid_roles = (
    "system",
    "user",
    "assistant",
    "function"
)

# Проверка ролей:
# содержит ли 'role' одну из valid_roles
if message.get(
    "role",
    None
) not in valid_roles:
    format_errors["unrecognized_role"] += 1

content = message.get(
    "content",
    None
)
function_call = message.get(
    "function_call",
    None
)

# Проверка контента:
# является ли 'content' строкой текста
# Также проверка наличия 'content' или 'function_call'
if (not content and not function_call) or \
    not isinstance(content, str):
    format_errors["missing_content"] += 1

# Наличие сообщения помощника:
# наличие как минимум одного сообщения помощника
if not any(
    message.get(
        "role",
        None
    ) == "assistant"

```

```
        for message in messages
    ):
        format_errors["example_missing_assistant_message"] += 1

# Вывод ошибок на печать (при наличии)
if format_errors:
    print("Найдены ошибки:")
    for k, v in format_errors.items():
        print(f"{k}: {v}")
    raise ValueError(
        "Набор данных содержит ошибки"
    )

# Выгружаем файл
uploaded = client.files.create(
    file=open(
        file_path,
        "rb"
    ),
    purpose="fine-tune"
)
file_id = uploaded.id

model = "gpt-3.5-turbo"
fine_tune_job = client.fine_tuning.jobs.create(
    training_file=file_id,
    model=model,
)

print("Выполняется проверка файла")
while fine_tune_job.status == "validating_files":
    fine_tune_job = client.fine_tuning.jobs.retrieve(
        fine_tune_job.id
    )
    sys.stdout.write("...")
    sys.stdout.flush()

print("Выполняется тонкая настройка")
while fine_tune_job.status == "running" \
    or fine_tune_job.status == "queued":
    fine_tune_job = client.fine_tuning.jobs.retrieve(
        fine_tune_job.id
    )
    sys.stdout.write("...")
    sys.stdout.flush()

print("Тонкая настройка завершена")
print("Название новой модели: " \
    + fine_tune_job.fine_tuned_model)
EOF
```

Максимальное количество токенов

Максимальное количество токенов на одно сообщение зависит от выбранной вами модели. Для GPT-3.5-Turbo максимальное количество токенов в сообщении составляет 16 385. Для GPT-3.5-Turbo-0613 максимальное количество токенов в сообщении составляет 4096. Если сообщение превышает максимальное количество токенов, оно будет усечено, чтобы соответствовать лимиту, что может привести к потере информации. Убедитесь, что весь набор данных не превышает максимальное количество токенов.

Размер набора данных

Размер набора данных является решающим фактором при тонкой настройке. Требуется минимум 10 примеров, но OpenAI рекомендует использовать с GPT-3.5-Turbo хотя бы 50–100 примеров. Как правило, чем больше примеров вы предоставите, тем лучше будут результаты.

Максимальный размер загружаемого файла – 1 ГБ. Однако OpenAI не рекомендует выполнять тонкую настройку с таким объемом данных, поскольку вам вряд ли понадобится такой большой объем, чтобы увидеть улучшения. Кроме того, чем больше файл, тем больше времени потребуется на его загрузку, обработку и обучение.

Тестирование и улучшение обучения (гиперпараметры)

После этапа обучения вы можете оценить качество модели посредством обычного чата. Если результаты не соответствуют вашим ожиданиям, вы можете переобучить модель, используя другие гиперпараметры. Первоначально OpenAI предлагает тонкую настройку без указания каких-либо гиперпараметров, позволяя модели выбирать значения по умолчанию в зависимости от размера набора данных. Если результаты по-прежнему не соответствуют вашим ожиданиям, вы можете настроить количество эпох, коэффициент скорости обучения и размер пакета.

Количество эпох

Количество эпох означает, сколько раз алгоритм обучения будет обрабатывать весь набор обучающих данных. В течение каждой эпохи модель один раз подвергается воздействию всех примеров в наборе обучающих данных. Это означает, что если ваш набор данных содержит 500 примеров, за одну эпоху модель будет обрабатывать все 500 примеров. Обработывая каждый пример, модель делает прогнозы и корректирует свои внутренние обучаемые параметры на основе расхождения между ее прогнозами и фактическими данными.

В некоторых случаях одного прохода по набору данных (одной эпохи) может оказаться недостаточно для обучения модели. Поэтому может потребоваться выполнить несколько эпох. Каждая эпоха предоставляет модели еще одну возможность извлечь уроки из данных, улучшить свои прогнозы и еще больше сократить потери.

Изменение количества эпох часто помогает улучшить результаты, но чрезмерное его увеличение может привести к переобучению.

Переобучение происходит, когда модель обучения работает исключительно хорошо на обучающих данных, но не может эффективно обобщаться на незнакомые или тестовые данные. Модель изучает все детали и шум в обучающих данных до такой степени, что это отрицательно влияет на качество модели на новых данных.

Процесс настройки количества эпох (и любая настройка гиперпараметров в целом) происходит итеративно. Начинать следует с небольшого количества эпох и постепенно увеличивать его, пока не достигнете желаемых результатов. Если вы заметили, что модель переобучилась (начало снижаться качество работы с незнакомыми данными) и/или ей не хватает разнообразия, вам следует немного уменьшить количество эпох, протестировать и повторить при необходимости.

Вот пример настройки количества эпох, в котором мы используем 10 эпох:

```
client.fine_tuning.jobs.create(
    training_file=file_id,
    model=model,
    hyperparameters={
        "n_epochs": 10
    }
)
```

Коэффициент скорости обучения

Этот гиперпараметр изменяет базовую скорость, используемую в процессе обучения. Он определяет размер шагов, которые модель предпринимает в процессе оптимизации, и контролирует, насколько обновляются веса модели во время обучения.

Представьте, что вы обучаете человека ориентироваться в лабиринте. Скорость обучения модели аналогична размеру шагов, которые человек делает при исследовании лабиринта.

При перемещении по лабиринту человек может делать большие или маленькие шаги. Большой шаг (более высокая скорость обучения) означает быстрое преодоление большого расстояния, но существует риск проскочить мимо нужного поворота и пропустить правильный путь. И наоборот, меньший шаг (более низкая скорость обучения) означает осторожное движение: ни одна деталь не будет упущена, но поиск выхода занимает больше времени.

Регулировка скорости обучения модели при тонкой настройке сродни настройке размера шага для человека, который учится перемещаться по лабиринту. Если модель не сходится (т. е. не улучшает свою производительность на обучающих данных), иногда помогает увеличение коэффициента скорости обучения. Однако делать это следует осторожно, чтобы избежать рисков, связанных со слишком высокой скоростью обучения.

Ниже показан фрагмент кода для настройки скорости обучения. В этом примере мы используем коэффициент 1.5:

```
client.fine_tuning.jobs.create(
    training_file=file_id,
    model=model,
```

```

hyperparameters={
  "learning_rate_multiplier": 1.5
}
)

```

Как правило, модель машинного обучения достигает сходимости, когда ее функция потерь перестает уменьшаться. Функция потерь измеряет, насколько точно модель предсказывает целевую переменную. Чем меньше потери, тем лучше модель предсказывает целевую переменную при заданных входных данных. Функция потерь рассчитывается в процессе обучения, а веса модели обновляются, чтобы минимизировать потери. Когда потери перестают уменьшаться, это означает, что модель достигла сходимости.

Размер пакета

Если говорить упрощенно, размер пакета – это количество примеров, которые модель обрабатывает одновременно. Этот гиперпараметр можно настроить для повышения производительности модели.

Размер пакета можно рассматривать как аналог загрузки грузовика. Каждый предмет в грузовике представляет собой обучающий пример, а размер пакета – это количество предметов, которые вы загружаете в грузовик одновременно. Поскольку грузовик имеет ограниченную вместимость, алгоритм обрабатывает определенное количество примеров за итерацию.

Большие размеры пакетов могут сократить общее количество итераций, необходимых для обработки всего набора данных, поскольку за одну итерацию обрабатывается больше примеров. Однако это не всегда приводит к сокращению общего времени обучения. Большие пакеты могут замедлить скорость сходимости, так как модель реже обновляет свои веса. С другой стороны, меньшие размеры пакетов приводят к более частым обновлениям, но требуют большего количества итераций для обработки всего набора данных, что потенциально может увеличить время обучения.

Вот пример кода для настройки размера пакета (в этом примере мы используем 32):

```

client.fine_tuning.jobs.create(
  training_file=file_id,
  model=model,
  hyperparameters={
    "batch_size": 32
  }
)

```

Ориентировочная оценка затрат

Приступая к настройке модели, важно иметь представление о затратах, связанных с обучением, использованием входных и выходных данных. Если вы тестируете небольшой набор данных, затраты могут быть незначительными. Однако по мере роста набора данных и более частого использования модели затраты могут возрасти. Давайте посмотрим, как соотносятся затраты для разных моделей и типов использования.

Стоимость обучения: GPT-3.5-Turbo – самая дорогая модель для обучения. Напротив, Babbage-002 является самой дешевой в обучении: ее стоимость составляет 0.0004 долл. США за 1 тыс. токенов.

Стоимость обучения связана с тонкой настройкой модели для вашего конкретного набора данных. Когда вы обучаете (или настраиваете) модель, вы, по сути, обновляете ее параметры, чтобы они лучше соответствовали вашим данным и варианту использования. Стоимость рассчитывается на основе количества токенов в вашем наборе обучающих данных и взимается за 1000 токенов. Это единовременные затраты, понесенные в процессе обучения.

Стоимость ввода: Davinci-002 – самая дорогая модель относительно ввода, стоимость которой составляет 0.0120 долл. США за 1 тыс. токенов. Babbage-002 является самой дешевой с точки зрения ввода: ее стоимость составляет 0.0016 долл. США за 1 тыс. токенов.

Стоимость ввода – это стоимость токенов, которые вы отправляете в модель в качестве входных данных во время регулярного использования после обучения (т. е. когда вы отправляете запросы к модели для прогнозов или ответов). Например, если вы отправляете модели промпт на создание текста, количество токенов в этом промпте учитывается как входные данные.

Стоимость вывода: Davinci-002 – самая дорогая модель с точки зрения вывода; ее стоимость составляет 0.0120 долл. США за 1 тыс. токенов. Babbage-002 является наименее дорогой относительно вывода – ее стоимость составляет 0.0016 долл. США за 1 тыс. токенов.

Стоимость вывода связана с количеством токенов, сгенерированных моделью в ответ на ваши входные данные. Другими словами, каждый токен в выходных данных учитывается при выставлении счетов.

Упомянутые затраты сведены в следующую таблицу:

Модель	Стоимость обучения, долл. США за 1 тыс. токенов	Стоимость ввода, долл. США за 1 тыс. токенов	Стоимость вывода, долл. США за 1 тыс. токенов
GPT-3.5-Turbo	0.0080	0.0030	0.0060
davinci-002	0.0060	0.0120	0.0120
davinci-002	0.0004	0.0016	0.0016

В долгосрочной перспективе стоимость тонкой настройки становится незначительной по сравнению с текущими затратами на использование входных и выходных данных, поскольку оплачивается однократно. Таким образом, применение GPT-3.5-Turbo для тонкой настройки – это экономически эффективный выбор, обеспечивающий баланс между стоимостью и производительностью на протяжении длительного времени.

Качество набора данных

Качество вашего набора данных является решающим фактором успеха тонкой настройки. Для достижения оптимальных результатов ваш набор данных должен соответствовать задаче, которую вы стремитесь выполнить. Например, если вы создаете чат-бота для определенной предметной области, вам следует начать с четкого определения следующих аспектов:

- определите конкретную область, в которой будет специализироваться ваш чат-бот;
- сформулируйте задачу, которую вы хотите решить. Определите конкретные проблемы или запросы, для решения которых предназначен ваш чат-бот;
- целевая аудитория: определите, для кого предназначен ваш чат-бот;
- укажите язык или языки, которые будет поддерживать ваш чат-бот;
- стиль диалога: определите тональность ответов вашего чат-бота. Например, чат-бот в сфере здравоохранения должен придерживаться профессионального информативного стиля, а чат-бот игрового сайта может быть более непринужденным и игривым.

Более того, важно убедиться, что ваши данные чисты, актуальны и хорошо структурированы. Избегайте предвзятостей, поскольку они могут привести к неоптимальным результатам.

Сочетание тонкой настройки с другими методами

Тонкая настройка становится более эффективной в сочетании с другими методами, такими как разработка промптов, поиск информации и вызов функций.

Разработка промптов (prompt engineering) – это процесс разработки подсказок, позволяющих направить модель к получению желаемого результата. Например, если вы разрабатываете чат-бота для обслуживания клиентов, вместо использования универсального промпта, такого как «Ответить клиенту», вы можете использовать более конкретный промпт, например «Как дружелюбный и полезный помощник по обслуживанию клиентов, ответь на вопрос клиента о политике возврата товара».

Методика поиска информации предполагает интеграцию модели с системой, которая извлекает информацию из базы данных или интернета. Например, когда пользователь задает конкретный вопрос чат-боту для юридических консультаций, модель может быть запрограммирована на получение соответствующей юридической информации из уже существующей базы данных или надежных юридических веб-сайтов перед составлением ответа. Популярный метод называется *дополненной генерацией ответа* (retrieval-augmented generation, RAG).

RAG – это усовершенствованный метод обработки естественного языка с помощью искусственного интеллекта, объединяющий методы поиска и генерации. Он расширяет возможности моделей искусственного интеллекта по выдаче точных и контекстуально релевантных ответов за счет использования внешних источников информации. Метод состоит из двух основных этапов: поиска и генерации. На этапе поиска RAG использует

механизм поиска для сбора соответствующей информации из обширной внешней базы знаний. Эти данные могут быть получены из статей, веб-страниц или любого структурированного или неструктурированного текста. Поиск может быть основан на сопоставлении ключевых слов, семантическом поиске или других сложных методах поиска соответствующей информации для процесса генерации. Затем на втором этапе полученная ранее информация применяется для генерации ответа.

Вызов функций предполагает интеграцию внешних функций в ответы модели. Например, в чат-боте с рекомендациями о поездках, когда пользователь запрашивает информацию о погоде для определенного места, чат-бот может вызвать внешний API метеослужбы, чтобы получить текущие данные о погоде и включить эту информацию в свой ответ.

Экспериментируйте и учитесь

Тонкой настройке моделей невозможно научиться полностью. Каждая задача по тонкой настройке – это возможность улучшить ваше понимание и усовершенствовать подход. Возможно, вам придется экспериментировать с различными гиперпараметрами, подсказками, размерами наборов данных, а иногда даже с самим набором данных, чтобы добиться лучших результатов.

Используйте проверочные наборы данных

Цель тонкой настройки – адаптировать предварительно обученную модель для эффективного выполнения конкретной задачи. Проверочный набор данных нужен, чтобы удостовериться, что модель обучается должным образом. Однако важно отметить, что проверочный набор должен представлять собой подмножество данных, полностью отдельное от обучающих данных, а не подмножество обучающих данных. Этот проверочный набор используют для оценки качества модели в процессе обучения. Проверочный набор в основном применяется для настройки гиперпараметров модели и предотвращения переобучения за счет обратной связи о том, как модель работает на данных, которые она не видела во время обучения.

Вот как вы можете использовать проверочный набор:

```
# Определяем путь к файлу
validation_file_path = os.path.join(
    os.path.dirname(__file__),
    'validation_data.jsonl'
)

# Загружаем проверочный файл
validation_uploaded = client.files.create(
    file=open(
        validation_file_path,
        "rb"
    ),
    purpose="fine-tune"
)
```

```
# Получаем ID проверочного файла
validation_file_id = validation_uploaded.id

# Создаем задачу тонкой настройки
client.fine_tuning.jobs.create(
    training_file=
    model=model,
    validation_file=validation_file_id
)
```

Тестирование модели

Исходный набор данных целесообразно разделить на две части: обучающую и тестовую. Набор обучающих данных используется для обучения модели, а набор тестовых данных используется для тестирования модели после обучения. Этот подход является эффективным способом оценить качество модели и выявить любые потенциальные проблемы.

Анализ результатов

После того как вы загрузили набор данных и запустили задание по тонкой настройке, вы можете отслеживать ход выполнения задания на панели мониторинга. После завершения задания вы можете просмотреть результаты на информационной панели. Одним из показателей оценки обучения является величина потерь.

Потери при обучении – это численная мера того, насколько хорошо модель соответствует обучающим данным. Более низкое значение потерь обычно указывает на лучшую точность модели. Чтобы получить эту метрику и другую информацию, вы можете использовать следующий код.

Сначала экспортируйте идентификатор задания:

```
export JOB_ID="<<JOB_ID>"
```

Далее создайте скрипт:

```
cat << EOF > src/analytics.py
from api import client

# Получаем ID задания тонкого обучения
job_id = "$JOB_ID"

fine_tune_job = \
    client.fine_tuning.jobs.retrieve(job_id)

status = fine_tune_job.status
if status == "succeeded":

    # Печать n_epochs
    print(f"Количество эпох: \
    {fine_tune_job.hyperparameters.n_epochs}")
```

```

# Печать коэффициента скорости обучения
print(f"Коэффициент скорости обучения: \
{fine_tune_job.hyperparameters.learning_rate_multiplier}")

# Печать размера пакета
print(f"Размер пакета: \
{fine_tune_job.hyperparameters.batch_size}")

# Печать обучающих токенов
print(f"Обучающих токенов: \
{fine_tune_job.trained_tokens}")

# Печать потерь при обучении
print("Потери при обучении:")
events = client.fine_tuning.jobs.list_events(job_id)
for event in events.data:
    if event.type == 'metrics':
        # Доступ к значениям словаря по ключам
        step = event.data['step']
        train_loss = event.data['train_loss']
        print(f"Шаг {step}: потеря при обучении={train_loss}")
else:
    print("Задание не завершено или произошла ошибка")
EOF

```

Наконец, запустите код:

```
python src/analytics.py
```

Ваш вывод должен выглядеть следующим образом:

```

Количество эпох: 10
Коэффициент скорости обучения: 2
Размер пакета: 1
Обучающих токенов: 4330
Потери при обучении:
Шаг 91: потеря при обучении=0.09158027172088623
Шаг 81: потеря при обучении=0.24630801379680634
Шаг 71: потеря при обучении=0.33399760723114014
Шаг 61: потеря при обучении=0.7304694056510925
Шаг 51: потеря при обучении=0.8583178520202637
Шаг 41: потеря при обучении=0.7920037508010864
Шаг 31: потеря при обучении=1.3004353046417236
Шаг 21: потеря при обучении=3.5548171997070312
Шаг 11: потеря при обучении=4.956674575805664
Шаг 1: потеря при обучении=2.6598880290985107

```

Потери при обучении, отображаемые в приведенных выше выходных данных, составляют 0.09158027172088623, что указано на последнем шаге (шаг 91). Вы можете получить доступ к тем же данным на platform.openai.com¹.

¹ <https://platform.openai.com>.

Продвинутый пример тонкой настройки: виртуальный консультант

В этой главе мы углубимся в процесс тонкой настройки языковой модели и займемся созданием сложного чат-бота, выступающего в роли консультанта-психотерапевта. Этот чат-бот сможет давать пользователям профессиональные советы и рекомендации по широкому спектру тем психического здоровья. К ним относятся, помимо прочего, такие темы, как депрессия, тревога, управление стрессом, и многие другие актуальные проблемы, с которыми люди могут столкнуться в своей повседневной жизни.

На пути к этой цели вы пройдете через пошаговое руководство, начиная с процесса подготовки набора данных, необходимых для этой задачи. Сюда входят сбор, очистка и обработка данных таким образом, чтобы оптимизировать качество и быстродействие нашей модели. После этого мы займемся тонкой настройкой языковой модели, разбивая сложные концепции и методы на простые для понимания шаги.

Ближе к концу главы мы рассмотрим практические аспекты использования модели в реальных приложениях. У вас появится понимание потенциальных ограничений и проблем, с которыми можно столкнуться в реальности, и вы узнаете о способах их преодоления.

НАБОР ДАННЫХ, ИСПОЛЬЗУЕМЫЙ В ПРИМЕРЕ

Набор данных, первоначально загруженный с сайта [huggingface.co](https://huggingface.co/datasets/Amod/mental_health_counseling_conversations)¹, представляет собой набор вопросов и ответов (на английском языке)², полученных с двух платформ онлайн-консультирования и терапии. Вопросы охватывают широкий спектр тем психического здоровья, а ответы дают квалифицированные психологи. Набор данных предназначен для точной настройки языковых моделей, чтобы улучшить их способность предоставлять консультации по психологии.

Сильная сторона набора данных в том, что он содержит реальные диалоги между пользователями и специалистами в области психологии. Это делает его ценным ресурсом для точной настройки языковых моделей, предоставляющих пользователям качественные психологические консультации.

¹ https://huggingface.co/datasets/Amod/mental_health_counseling_conversations.

² По этой причине все примеры этой главы необходимо выполнять также на английском языке. Частичные переводы полей данных приведены только для общего представления о характере данных. – *Прим. перев.*

Все данные были анонимизированы автором¹, и никакая личная информация не сохранилась и не может быть раскрыта.

Запись данных содержит поля Context (контекст) и Response (ответ):

- Context: строка, содержащая вопрос, заданный пользователем;
- Response: строка, содержащая соответствующий ответ психолога.

Примеры:

```
{
  "Context": "I'm going through some things with my feelings and myself. I barely sleep and I do nothing but think about how I'm worthless and how I shouldn't be here.\n I've never tried or contemplated ...etc",
  "Response": "If everyone thinks you're worthless, then maybe you need to find new people to hang out with.Seriously, the social context in which a person lives is a big influence in self-esteem. Otherwise, you can go round and round trying to understand why you're not worthless...etc"}

```

```
{
  "Контекст": "Меня беспокоят мои чувства и переживания. Я почти не сплю и ничего не делаю, кроме мыслей о том, насколько я бесполезен и зачем вообще существую. Я никогда не пробовал и не задумывался о том, чтобы... и т. д.",
  "Ответ": "Если все думают, что ты никчемный, то, возможно, тебе нужно найти новых людей, с которыми ты будешь общаться. Seriously, социальный контекст, в котором живет человек, очень сильно влияет на его самооценку. Иначе можно ходить вокруг да около, пытаюсь понять, почему ты не никчемный... и т. д."}

```

```
{
  "Context": "My husband and I are separated and he doesn't even want to talk to me. He says he doesn't love me anymore, but I would do anything to get him back. Is there any hope?...etc",
  "Response": "Most important is to take care of your feelings regarding that he has left you. From your description...etc"}

```

```
{
  "Контекст": "Мы с мужем расстались, и он даже не хочет со мной разговаривать. Он говорит, что больше не любит меня, но я сделаю все, чтобы вернуть его. Есть ли надежда?... и т. д.",
  "Ответ": "Самое главное – позаботьтесь о собственных чувствах по поводу того, что он вас бросил. Судя по вашему описанию... и т. д."}

```

и т. д.

Вы можете скачать набор данных с HuggingFace. Кроме того, вы можете выполнить приведенные ниже команды, чтобы загрузить его из репозитория GitHub², если вы пользователь Linux, или получить данные в файловом архиве на сайте издательства русского перевода:

```
# Создаем папку
mkdir -p src/data

# Скачиваем данные
wget https://raw.githubusercontent.com/eon01/Openai_GPT_for_Python_Developers/main/datasets/mental_health_counseling_conversations/data.json \
-O src/data/data.json

```

¹ <https://huggingface.co/Amod>.

² https://github.com/eon01/Openai_GPT_for_Python_Developers_Files.

ПОДГОТОВКА ДАННЫХ

Данные содержат большое количество объектов JSON, каждый из которых имеет Context и Response. Однако согласно рекомендациям OpenAI данные должны быть в формате JSONL, где каждая строка должна содержать объект JSON.

Каждый объект JSON должен выглядеть следующим образом:

```
{
  "messages":
  [
    {
      "role": "system",
      "content": "Сообщение системы #1"
    }
    ,
    {
      "role": "user",
      "content": "Сообщение пользователя #1"
    }
    ,
    {
      "role": "assistant",
      "content": "Сообщение помощника #1"
    }
  ]
}
{
  "messages":
  [
    {
      "role": "system",
      "content": "Сообщение системы #2"
    }
    ,
    {
      "role": "user",
      "content": "Сообщение пользователя #2"
    }
    ,
    {
      "role": "assistant",
      "content": "Сообщение помощника #2"
    }
  ]
}
'''и т. д.
```

Прежде чем приступить к конвертации данных в нужный формат, установим необходимые пакеты:

```
pip install setuptools==69.0.3
pip install langdetect==1.0.9
pip install unidecode==1.3.8
```

Теперь конвертируем данные при помощи следующего кода:

```
cat << 'EOF' > src/prepare_data.py
import json, re
from langdetect import detect
from unidecode import unidecode

data = []

def clean_text(text):
    # Заменяем символы Unicode
    # эквивалентами ASCII
    text = unidecode(text)

    # Удаляем веб-адреса
    text = re.sub(
        r'https?://\S+|www\.\S+', '',
        text
    )

    # Исправляем пробелы рядом со знаками препинания
    text = re.sub(
        r'\s*([,;!?])\s*', r'\1 ',
        text
    )

    # Удаляем пробелы в начале и конце строки
    text = text.strip()

    # Добавляем отсутствующие пробелы после знаков препинания
    text = re.sub(
        r'([:;!?])([^\s])', r'\1 \2',
        text
    )

    return text

# Открываем src/data/data.jsonl и удаляем все данные
open('src/data/data.jsonl', 'w').close()

system_prompt = "You are MendMind, an AI Mental Health Coach. \
Your purpose is to support the user through their mental \
health journey with empathy, understanding, and insights \
into managing emotional and psychological challenges. \
While you can provide general advice and emotional \
support, you are not equipped to handle personal contact, \
schedule appointments, or share any specific location \
details. Your only role is to help the user with coping \
strategies, provide information on mental health topics, \
and guide them towards professional resources if needed. \
```

You can engage in a regular conversation with the user, \
but your primary focus is what you can do best: \
supporting the user with confidentiality and care in \
the path to well-being."

```
# Открываем файл данных и читаем его содержимое
with open('src/data/data.json', 'r') as file:
    for line in file:
        json_line = json.loads(line)
        context = json_line["Context"]
        response = json_line["Response"]

try:
    # Если context не пустой и response
    # длиннее 10 слов
    if len(context) > 0 and len(response.split()) > 10:
        # Проверка того, что context и
        # response на английском языке
        if detect(context) == "en" and detect(response) == "en":
            system = {
                "role": "system",
                "content": system_prompt
            }

            user = {
                "role": "user",
                "content": clean_text(context)
            }

            assistant = {
                "role": "assistant",
                "content": clean_text(response)
            }

            messages = {
                "messages": [system, user, assistant]
            }

            # записываем в data.jsonl
            with open('src/data/data.jsonl', 'a') as file:
                # записываем сообщения в файл
                file.write(json.dumps(messages) + '\n')

        # Проверка того, что context и
        # response на английском языке
        if detect(context) == "en" and detect(response) == "en":
            system = {
                "role": "system",
                "content": system_prompt
            }

            user = {
                "role": "user",
```

```

        "content": clean_text(context)
    }

    assistant = {
        "role": "assistant",
        "content": clean_text(response)
    }

    messages = {
        "messages": [system, user, assistant]
    }

    # записываем в data.jsonl
    with open('src/data/data.jsonl', 'a') as file:
        file.write(json.dumps(messages) + "\n")
except:
    print(f"Ошибка:\n Context: {context}\n Response: {response}")
EOF

```

После запуска приведенного выше кода (`python src/prepare_data.py`) данные будут в требуемом формате JSONL. Вот пример строки JSON из файла `data.jsonl`:

```

{"messages": [{"role": "system", "content": "You are MendMind, an AI Mental Health Coach.etc"}, {"role": "user", "content": "I'm going through some things with my feelings and myself. I barely sleep and I do nothing but think about how I'm worthless and how I shouldn't be here.\n ..."}, {"role": "assistant", "content": "If everyone thinks you're worthless, then maybe you need to find new people to hang out with.Seriously, the social context in which a person lives is a big influence in self-esteem.\n .."}]}

```

Поверяем данные, затем иницилируем процесс обучения с помощью следующего кода (за основу взят код из предыдущей главы):

```

cat << EOF > src/fine_tuning_couch.py
from api import client
import os, sys, json
from collections import defaultdict

file_path = os.path.join(
    os.path.dirname(__file__),
    'data.jsonl'
)

# Проверка формата файла данных
with open(file_path, 'r', encoding='utf-8') as f:
    try:
        dataset = [json.loads(line) for line in f]
    except:
        raise ValueError(
            "Файл набора данных должен быть в формате JSONL"
        )
)

```

```
# Проверка размера набора данных
size = len(dataset)
if size < 10:
    raise ValueError(
        "Набор данных должен содержать не менее 10 примеров"
    )

# Проверка набора данных
format_errors = defaultdict(int)

for line in dataset:
    # Проверка типа данных:
    # проверка каждой записи в словаре
    if not isinstance(line, dict):
        format_errors["data_type"] += 1
        continue

    # Список сообщений:
    # проверка наличия списка 'messages'
    messages = line.get("messages", None)
    if not messages:
        format_errors["missing_messages_list"] += 1
        continue

    for message in messages:
        # Проверка ключей сообщений:
        # гарантия, что сообщение содержит ключи 'role' и 'content'
        if "role" not in message or "content" not in message:
            format_errors["message_missing_key"] += 1

        # Список допустимых ключей сообщения
        valid_keys = (
            "role",
            "content",
            "name",
            "function_call"
        )

        # Нераспознанные ключи сообщения:
        # наличие ключей, которые не являются допустимыми
        if any(k not in valid_keys for k in message):
            format_errors["message_unrecognized_key"] += 1

        # Допустимые роли, которые может иметь сообщение
        valid_roles = (
            "system",
            "user",
            "assistant",
            "function"
        )

    # Проверка ролей:
```

```
# Содержит ли 'role' одну из valid_roles
if message.get("role", None) not in valid_roles:
    format_errors["unrecognized_role"] += 1

content = message.get("content", None)
function_call = message.get("function_call", None)

# Проверка контента:
# является ли 'content' строкой текста
# Также проверка наличия 'content' или 'function_call'
if (not content and not function_call) or \
    not isinstance(content, str):
    format_errors["missing_content"] += 1

# Наличие сообщения помощника:
# наличие как минимум одного сообщения помощника
if not any(
    message.get("role", None) == "assistant"
    for message in messages
):
    format_errors["example_missing_assistant_message"] += 1

# Вывод ошибок на печать (при наличии)
if format_errors:
    print("Найдены ошибки:")
    for k, v in format_errors.items():
        print(f"{k}: {v}")
    raise ValueError(
        "Набор данных содержит ошибки"
    )

# Выгружаем файл
uploaded = client.files.create(
    file=open(
        file_path,
        "rb"
    ),
    purpose="fine-tune"
)
file_id = uploaded.id

model = "gpt-3.5-turbo"
fine_tune_job = client.fine_tuning.jobs.create(
    training_file=file_id,
    model=model,
)

print("Выполняется проверка файла")
while fine_tune_job.status == "validating_files":
    fine_tune_job = client.fine_tuning.jobs.retrieve(
        fine_tune_job.id
    )
```

```

sys.stdout.write("...")
sys.stdout.flush()

print("Выполняется тонкая настройка")
while fine_tune_job.status == "running" \
    or fine_tune_job.status == "queued":
    fine_tune_job = client.fine_tuning.jobs.retrieve(
        fine_tune_job.id
    )
    sys.stdout.write("...")
    sys.stdout.flush()

print("Тонкая настройка завершена")
print("Название новой модели: " \
    + fine_tune_job.fine_tuned_model)
EOF

```

Запустим процесс тонкой настройки:

```
python src/fine_tuning_couch.py
```

Процесс тонкой настройки займет некоторое время, поскольку набор данных относительно большой. После завершения процесса у вас будет новая модель. Точная настройка будет стоить вам около 27 долл. США, если цена останется такой же, как на момент написания этой книги.

Экспортируйте имя новой модели:

```
export FINE_TUNED_MODEL="<FINE_TUNED_MODEL>"
```

Теперь вы можете использовать модель, создав интерактивный сеанс буквально одним щелчком мыши (как вы видели в предыдущих главах):

```

cat << EOF > src/app_49.py
from api import client
import click

model = "$FINE_TUNED_MODEL"
system_prompt = "You are MendMind, an AI Mental Health Coach. \
Your purpose is to support the user through their mental \
health journey with empathy, understanding, and insights \
into managing emotional and psychological challenges. \
While you can provide general advice and emotional \
support, you are not equipped to handle personal contact, \
schedule appointments, or share any specific location \
details. Your only role is to help the user with coping \
strategies, provide information on mental health topics, \
and guide them towards professional resources if needed. \
You can engage in a regular conversation with the user, \
but your primary focus is what you can do best: \
supporting the user with confidentiality and care in \
the path to well-being."

base_messages = [

```

```
{
    "role": "system",
    "content":
        system_prompt
},
{
    "role": "user",
    "content":
        "Hi there."
},
{
    "role": "assistant",
    "content":
        "My name is MendMind. "
        "I'm an AI Mental Health Coach. "
        "How can I help you today?"
}
]

while True:
    messages = base_messages.copy()

    # чтение пользовательского ввода
    request = input(
        click.style(
            "Input: (type 'exit' to quit): ",
            fg="green"
        )
    )

    if request.lower() in ["exit", "quit"]:
        break

    # Добавляем пользовательский ввод к сообщению
    messages.append(
        {
            "role": "user",
            "content": f"{request}"
        }
    )

    # передача сообщения в API
    response = client.chat.completions.create(
        model=model,
        messages=messages,
        temperature=0.7,
        frequency_penalty=0.5,
        presence_penalty=0.5,
    )

    # получаем ответ
    content = response.choices[0].message.content.strip()
```

```
# Печать вывода модели
click.echo(
    click.style(
        "Вывод: ",
        fg="yellow"
    ) + content
)

click.echo()
EOF
```

Запустите этот код, чтобы начать интерактивную сессию:

```
python src/app_49.py
```

ПРОБЛЕМЫ ИСПОЛЬЗОВАНИЯ МОДЕЛИ В РЕАЛЬНЫХ ПРИЛОЖЕНИЯХ

Вы можете интегрировать модель в реальные приложения. Например, ее можно использовать для предоставления психологических консультаций в чат-боте для веб- или мобильных приложений или в приложении «умный домашний помощник». Потенциально возможности безграничны.

Однако крайне важно учитывать этические последствия при использовании модели в реальных приложениях. Например, нужна уверенность, что модель используется ответственно и не дает вредных советов. Кроме того, важно использовать модель таким образом, чтобы обеспечить строгое соблюдение конфиденциальности пользователей.

Несмотря на свою полезность, модель не идеальна и может давать неточные советы. С функциональной точки зрения пользователь может столкнуться с проблемами, поскольку модель не всегда справляется с полноценным диалогом. Например, у него могут возникнуть проблемы с разговорами, требующими нескольких поворотов рассуждения или дополнительной информации от пользователя. Это связано с тем, что модель предназначена для предоставления одного ответа на один входной запрос.

Следующий пример иллюстрирует ограничения модели при ведении интерактивных диалогов (переведен на русский язык для наглядности):

Ввод: (наберите 'exit' для выхода): Привет. Каковы симптомы депрессии?

Вывод: Депрессия может проявляться по-разному в зависимости от личности, но наиболее распространенные симптомы таковы:

1. Постоянное чувство грусти, опустошенности или безнадежности.
2. Потеря интереса или удовольствия от занятий, которые раньше нравились.
3. Изменение аппетита или веса (заметное увеличение или уменьшение).
4. Бессонница или избыточная сонливость.
5. Усталость или потеря энергии.
6. Чувство вины, никчемности или беспомощности.
7. Трудности с концентрацией внимания, принятием решений или запоминанием вещей.
8. Беспокойство или раздражительность.
9. Физические симптомы, такие как головные боли, боли в животе или боли в теле без явной медицинской причины.
10. ...etc

Ввод: (наберите 'exit' для выхода): Можешь рассказать подробнее о первом пункте?

Вывод: Конечно! Первый пункт относится к предназначению меня как консультанта по психологии. Как искусственный интеллект, я создан, чтобы поддержать вас на вашем пути к психическому здоровью, обеспечивая сочувствие и понимание того, как справляться с эмоциональными и психологическими проблемами. Я запрограммирован давать общие советы, излагать стратегии преодоления трудностей и информацию по тематике психического здоровья. Хотя я могу оказать эмоциональную поддержку, важно помнить, что я не заменяю консультации со специалистом. Если вы переживаете серьезный стресс или кризисную ситуацию, лучше всего обратиться за помощью к квалифицированному специалисту в области психического здоровья.

Конфиденциальность имеет решающее значение в нашем взаимодействии. Будьте уверены, что любая информация, которой вы поделитесь со мной, останется конфиденциальной и не будет передана кому-либо еще.

Есть ли что-то конкретное, в чем вам нужна помощь, или какая-то конкретная тема, которую вы хотели бы обсудить дальше?

Как видите, модель не может дать ответ на второй вопрос, который на самом деле является продолжением первого вопроса и общего контекста разговора. Это ограничение модели, о котором следует помнить при использовании ее в реальных приложениях.

В следующей главе мы рассмотрим некоторые решения данной проблемы.

Контекст и память: как сделать искусственный интеллект более реалистичным

В ЧЕМ ПРОБЛЕМА?

GPT – это генеративная текстовая модель, которая создает новый текст, предсказывая, что будет дальше, на основе входных данных, которые получает 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, он позволяет модели генерировать более разнообразный и менее повторяющийся текст.

В некоторых случаях невозможно обойтись без сохранения и переноса контекста. Тонкая настройка на конкретную тему помогает улучшить качество выходных данных, но не всегда хорошо работает, потому что разговор может в любое время сменить тему.

ОТСУТСТВИЕ КОНТЕКСТА = ХАОС СЛУЧАЙНОСТИ

Давайте начнем с создания простого чат-бота. Сначала мы иницилируем диалог, поскольку наша цель – сравнить результаты модели сейчас и позже, когда мы добавим больше контекста в разговор.

```
cat << 'EOF' > src/app_50.py
from api import client
import click

model = "gpt-3.5-turbo"
system_prompt = "Ты полезный ассистент."

base_messages = [
    {
        "role": "system",
        "content": system_prompt
    }
]
```

```
while True:
    messages = base_messages.copy()

    # Чтение пользовательского ввода
    request = input(
        click.style(
            "Ввод: (наберите 'exit', чтобы выйти): ",
            fg="green"
        )
    )

    if request.lower() in ["exit", "quit"]:
        break

    # Добавляем пользовательский ввод к сообщению
    messages.append(
        {
            "role": "user",
            "content": f"{request}"
        }
    )

    # Передаем сообщение в API
    response = client.chat.completions.create(
        model=model,
        messages=messages,
    )

    # Получаем ответ
    content = response.choices[0].message.content.strip()

    # Выводим ответ на печать
    click.echo(
        click.style(
            "Вывод: ",
            fg="yellow"
        ) + content
    )

    click.echo()
EOF
```

Запустите код командой

```
python src/app_50.py
```

Следующий тест показывает, что по умолчанию чат-бот не обладает какой-либо памятью.

Ввод: (введите 'exit', чтобы выйти): Привет!

Вывод: Здравствуйте! Чем я могу вам помочь?

Ввод: (введите 'exit', чтобы выйти): Мой кот черный.

Вывод: Это интересно! Черные кошки часто ассоциируются с тайнами и суевериями. Что еще вы можете рассказать о своей кошке?

Ввод: (введите 'exit', чтобы выйти): Почему?

Вывод: Почему что? Можете ли вы предоставить больше контекста или конкретизировать свой вопрос?

Ввод: (введите 'exit', чтобы выйти): Я говорил о коте!

Вывод: Ой, прошу прощения за путаницу. О чем бы вы хотели поговорить по поводу кота? Пожалуйста, предоставьте более конкретную информацию или задайте вопрос.

Ввод: (введите 'exit', чтобы выйти): Цвет!

Вывод: Цвет – это зрительное восприятие, возникающее в результате взаимодействия света с объектами. Именно определенные длины волн света, которые отражает, излучает или поглощает объект, определяют его цвет. Человеческий глаз имеет разные типы цветовых рецепторов, называемых колбочками, которые чувствительны к разным диапазонам длин волн. Это позволяет нам воспринимать широкий спектр цветов.

Общение с чат-ботом показывает, что модель не хранит в памяти предыдущий промпт. В результате модель может неправильно понять пользователя в ситуациях, требующих знания контекста предыдущего диалога, или дать случайный ответ, не связанный с темой разговора.

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. Благодаря этому текущий промпт всегда содержит историю диалога.

Усовершенствуем предыдущий код, добавив в него ведение истории диалога:

```
cat << 'EOF' > src/app_51.py
from api import client
import click

model = "gpt-3.5-turbo"
system_prompt = "Ты полезный ассистент."

history = [
    {
        "role": "system",
        "content": system_prompt
    }
]

while True:
    # Чтение пользовательского ввода
    request = input(
        click.style(
            "Ввод: (наберите 'exit', чтобы выйти): ",
            fg="green"
        )
    )
```

```
if request.lower() in ["exit", "quit"]:
    break

# Добавляем сообщение к истории
history.append(
    {
        "role": "user",
        "content": f"{request}"
    }
)

# Передаем сообщение в API
response = client.chat.completions.create(
    model=model,
    messages=history,
)

# Получаем ответ
content = response.choices[0].message.content.strip()

# Отладка: вывод истории на печать
click.echo(
    click.style(
        "История: ",
        fg="blue"
    ) + str(history)
)

# Выводим ответ на печать
click.echo(
    click.style(
        "Вывод: ",
        fg="yellow"
    ) + content
)

# Добавляем ответ к истории
history.append(
    {
        "role": "assistant",
        "content": f"{content}"
    }
)

click.echo()
EOF
```

Так выглядит диалог, если чат-бот хранит историю:

Ввод: (введите 'exit', чтобы выйти): Привет!

История: [{'role': 'system', 'content': 'Ты полезный помощник.'}, {'role': 'user', 'content': 'Привет!'}]

Вывод: Здравствуйте! Чем я могу вам помочь?

Ввод: (введите 'exit', чтобы выйти): Мой кот черный.

История: `[{'role': 'system', 'content': 'Ты полезный помощник.'}, {'role': 'user', 'content': 'Привет!'}, {'role': 'assistant', 'content': 'Здравствуйте! Чем я могу вам помочь?'}, {'role': 'user', 'content': 'Мой кот черный.'}]`

Вывод: Очень интересно! Черные кошки бывают красивыми. Хотите ли вы узнать что-то конкретное о черных кошках?

Ввод: (введите 'exit', чтобы выйти): Да.

История: `[{'role': 'system', 'content': 'Ты полезный помощник.'}, {'role': 'user', 'content': 'Привет!'}, {'role': 'assistant', 'content': 'Здравствуйте! Чем я могу вам помочь?'}, {'role': 'user', 'content': 'Мой кот черный.'}, {'role': 'assistant', 'content': 'Очень интересно! Черные кошки бывают красивыми. Хотите ли вы узнать что-то конкретное о черных кошках?'}, {'role': 'user', 'content': 'Да.'}]`

Вывод: Отлично, что именно вы хотите узнать о черных кошках? Охотно вам помогу!

Когда пользователь отвечает «да», модель понимает контекст и продолжает корректный диалог о предмете разговора.

НЕДОСТАТКИ ПЕРЕНОСА КОНТЕКСТА ЧЕРЕЗ ИСТОРИЮ

Чем дольше длится диалог, тем длиннее будет запрос пользователя, поскольку в него каждый раз будет добавляться предыдущая история. Так будет происходить до тех пор, пока не будет достигнуто максимальное количество токенов, разрешенное OpenAI. Конкретное значение зависит от модели, но ограничение в любом случае существует.

Вторая проблема – стоимость. Вы платите за токены, поэтому чем больше токенов у вас на входе, тем дороже обойдется запрос к API.

Одним из решений проблемы является сохранение в истории лишь нескольких последних запросов и ответов. Об этом пойдет речь в следующем разделе.

ПАМЯТЬ «ПОСЛЕДНИМ ВОШЕЛ – ПЕРВЫМ ВЫШЕЛ» (LIFO)

Я не уверен, есть ли у этого подхода отдельное название в машинном обучении, поэтому позаимствовал из области вычислительной техники термин «последним вошел – первым 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.

Вот как вкратце это работает:

- история сохраняется в файле;
- когда пользователь вводит промпт, программа читает N последних промптов из истории и передает их в API вместе с новым промптом пользователя;
- затем программа сохраняет промпт пользователя и ответ модели в файле истории;

- файл истории постоянно обновляется последними промптами и ответами;
- программа каждый раз читает N последних промптов из истории и передает их в API.

Следующий пример кода демонстрирует реализацию этого подхода:

```
cat << 'EOF' > src/app_52.py
from api import client
import click, json

# Определяем длину истории (в промптах)
n = 2
model = "gpt-3.5-turbo"
system_prompt = "Ты полезный помощник."

global_context = [
    {
        "role": "system",
        "content": system_prompt
    }
]

# Указываем путь к файлу
history_file_path = 'context.txt'

# Открываем файл в режиме 'w'
# и сразу закрываем его
with open(history_file_path, 'w') as file:
    pass

def save_history_to_file(history):
    """
    Сохранение истории диалога
    в файл.
    """
    with open(history_file_path, "w") as f:
        # Для хранения истории используем формат
        # строки JSON
        f.write(json.dumps(history))

def load_history_from_file():
    """
    Чтение истории из файла.
    """
    with open(history_file_path, "r") as f:
        import json
        try:
            history = json.loads(f.read())
            # Возвращает последние n записей
            return history[-n:]
        # В случае если файл пуст
        # или поврежден
        except json.JSONDecodeError:
            return []
```

```
full_history = []
while True:
    # Чтение пользовательского ввода
    request = input(
        click.style(
            "Ввод: (наберите 'exit', чтобы выйти): ",
            fg="green"
        )
    )

    if request.lower() in ["exit", "quit"]:
        break

    # Добавляем сообщение к истории
    history = {
        "role": "user",
        "content": request
    }

    # Читаем историю из файла
    # и добавляем к ней новое сообщение
    full_history = load_history_from_file()
    full_history.append(history)
    messages = global_context + full_history
    # Передаем сообщение в API
    response = client.chat.completions.create(
        model=model,
        messages=messages,
        max_tokens=150,
        temperature=0.7,
    )

    # Отладка: вывод истории на печать
    click.echo(
        click.style("История: ", fg="blue") + \
        str(json.dumps(
            messages,
            indent=4
        ))
    )

    # Получаем ответ
    content = response.choices[0].message.content.strip()

    # Выводим ответ на печать
    click.echo(
        click.style(
            "Вывод: ",
            fg="yellow"
        ) + content
    )
```

```

# Добавляем ответ к истории
full_history.append(
    {
        "role": "assistant",
        "content": content
    }
)

# Сохраняем историю в файл
save_history_to_file(full_history)
EOF

```

Если вы запустите этот код, то заметите, что история обновляется последними запросами и ответами. Затем история используется для отправки последних N запросов в API.

```
python src/app.py
```

В приведенном выше примере история обновляется последними запросами и ответами из файла. На самом деле для сохранения истории вы можете использовать любое хранилище данных (база данных, кеш и т. д.).

ПРОБЛЕМА С ПАМЯТЬЮ ТИПА LIFO

Подход, который я назвал памятью «последним пришел – первым ушел», может не справиться, когда обсуждение становится очень сложным, требующим от пользователя переключения между различными контекстами. В таких случаях этот алгоритм может не обеспечить необходимый контекст, поскольку он сохраняет только самые последние промпты. Это может привести к замешательству и разочарованию пользователей, что не способствует комфортному взаимодействию с системой.

Например, если N равно x , где x – относительно небольшое число, модель будет учитывать только x последних подсказок и ответов. Это означает, что если обсуждение выйдет за рамки x промптов, модель не сможет предоставить необходимый контекст. И наоборот, если N равно y , где y – относительно большое число, модель будет учитывать значительное количество промптов и ответов, что может привести к перерасходу ресурсов.

В следующем разделе мы рассмотрим более сложный подход, который устраняет ограничения памяти «последним пришел – первым ушел».

ИЗБИРАТЕЛЬНЫЙ КОНТЕКСТ

Альтернативное решение работает следующим образом:

- исходный запрос сохраняется в текстовый файл;
- пользователь вводит промпт, который зависит от контекста;
- программа создает встраивания для всех взаимодействий в файле;
- программа вычисляет косинусное сходство между запросом пользователя, текущим контекстом 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 следующем примере мы используем текстовый файл, чтобы упростить задачу, но, как было сказано ранее, вы можете использовать любое хранилище данных.

Мы будем использовать функцию встраивания для создания встраиваний промпта пользователя и всех взаимодействий в файле, а также функцию для расчета косинусного сходства между промптом пользователя и всеми взаимодействиями в файле. Выбор происходит на уровне загрузки истории из файла.

Вот что нам понадобится в качестве отправной точки:

```
embedding_model = "text-embedding-ada-002"
context_window = 5

def sort_history(history, prompt, context_window):
    """
    Сортировка истории взаимодействия на основе
    косинусного подобия.
    Возвращает верхние сегменты context_window.
    """
    sorted_history = []
    for segment in history:
        content = segment['content']
        preprocessed_content = preprocess_text(content)
        preprocessed_prompt = preprocess_text(prompt)
        embedding_model = "text-embedding-ada-002"
        embedding_content = get_embedding(
            preprocessed_content,
            embedding_model
        )
        embedding_prompt = get_embedding(
            preprocessed_prompt,
            embedding_model
        )
        similarity = cosine_similarity(
            embedding_content,
            embedding_prompt
        )
        sorted_history.append(
            (segment, similarity)
        )
    sorted_history = sorted(
        sorted_history,
        key=lambda x: x[1],
        reverse=True
    )
    sorted_history = [
        x[0] for x in sorted_history
    ]
    return sorted_history[:context_window]
```

Эта функция сортирует историю взаимодействий на основе косинусного сходства между промптом пользователя и сегментами истории. Функция сначала вычисляет косинусное сходство между промптом пользователя и каждым сегментом истории, используя функцию `cosine_similarity`. Затем она сорти-

рует сегменты в порядке убывания на основе их сходства с промптом пользователя и возвращает верхние сегменты `context_window`.

Функции `preprocess_text()` и `get_embedding()` были определены ранее в этой книге. Вот они:

```
# Определена в src/api.py
def get_embedding(text, model):
    text = text.replace("\n", " ")
    return client.embeddings.create(
        input = [text],
        model=model
    ).data[0].embedding

# Определена в src/utils.py
def preprocess_text(text):
    from nltk.corpus import stopwords
    from nltk.stem import PorterStemmer
    from nltk.tokenize import word_tokenize
    # Токенизация текста
    tokens = word_tokenize(text)

    # Преобразование в нижний регистр
    tokens = [
        word.lower() for word in tokens
    ]

    # Удаление пунктуации
    words = [
        word for word in tokens if word.isalpha()
    ]

    # Фильтрация стоп-слов
    stop_words = set(
        stopwords.words('english')
    )
    words = [
        word for word in words if word not in stop_words
    ]

    # Стемминг
    stemmer = PorterStemmer()
    stemmed_words = [
        stemmer.stem(word) for word in words
    ]

    return ' '.join(stemmed_words)
```

Следующий код вызывает функцию `sort_history`:

```
cat << 'EOF' > src/app_53.py
from api import client, get_embedding
from utils import preprocess_text, cosine_similarity
import click, json
```

```

context_window = 2
model = "gpt-3.5-turbo"
system_prompt = "Ты полезный помощник."
history_file_path = 'context.txt'
full_history = []

global_context = [
    {
        "role": "system",
        "content": system_prompt
    }
]

# Открываем файл в режиме 'w'
# и сразу закрываем его
with open(history_file_path, 'w') as file:
    pass

def save_history_to_file(history):
    """
    Сохраняем историю взаимодействия в файл.
    """
    with open(history_file_path, "w") as f:
        f.write(json.dumps(history))

def load_history_from_file():
    """
    Читаем историю из файла.
    """
    with open(history_file_path, "r") as f:
        import json
        try:
            history = json.loads(f.read())
            return history
        except json.JSONDecodeError:
            return []

def sort_history(history, prompt, context_window):
    """
    Сортировка истории взаимодействия на основе
    косинусного подобия.
    Возвращает верхние сегменты context_window.
    """
    sorted_history = []
    for segment in history:
        content = segment['content']
        preprocessed_content = preprocess_text(content)
        preprocessed_prompt = preprocess_text(prompt)
        embedding_model = "text-embedding-ada-002"
        embedding_content = get_embedding(
            preprocessed_content,
            embedding_model
        )

```

```
        embedding_prompt = get_embedding(
            preprocessed_prompt,
            embedding_model
        )
        similarity = cosine_similarity(
            embedding_content,
            embedding_prompt
        )
        sorted_history.append(
            (segment, similarity)
        )
    sorted_history = sorted(
        sorted_history,
        key=lambda x: x[1],
        reverse=True
    )
    sorted_history = [
        x[0] for x in sorted_history
    ]
    return sorted_history[:context_window]
```

```
while True:
    # Чтение пользовательского ввода
    request = input(
        click.style(
            "Ввод: (наберите 'exit', чтобы выйти): ",
            fg="green"
        )
    )

    if request.lower() in ["exit", "quit"]:
        break

    # Добавляем сообщение к истории
    history = {
        "role": "user",
        "content": request
    }

    # Читаем историю из файла
    # и добавляем к ней новое сообщение
    full_history = load_history_from_file()
    sorted_history = sort_history(
        full_history,
        request,
        context_window
    )
    sorted_history.append(user_prompt)
    messages = global_context + sorted_history
    # Передаем сообщение в API
    response = client.chat.completions.create(
        model=model,
```

```

        messages=messages,
        max_tokens=200,
        temperature=1,
    )

# Отладка: вывод истории на печать
click.echo(
    click.style("История: ", fg="blue") + \
    str(json.dumps(
        messages,
        indent=4
    ))
)

# Получаем ответ
content = response.choices[0].message.content.strip()

# Выводим ответ на печать
click.echo(
    click.style(
        "Вывод: ",
        fg="yellow"
    ) + content
)

# Добавляем к истории промпт пользователя
full_history.append(user_prompt)
# Добавляем к истории ответ модели
full_history.append(
    {
        "role": "assistant",
        "content": content
    }
)

# Сохраняем историю в файл
save_history_to_file(full_history)
EOF

```

Если вы запустите этот скрипт, то увидите, что история обновляется последними промптами и ответами. Затем эта история применяется для отправки последних N промптов в API ($N = \text{context_window}$).

Запустите скрипт при помощи следующей команды:

```
python src/app.py
```

Попробуйте последовательно ввести такие промпты:

```

У меня есть черная кошка.
У меня есть белая рубашка.
Какого цвета моя кошка?
Какого цвета моя рубашка?

```

Показанный ниже результат демонстрирует следование ответов модели контексту:

Ввод: (наберите 'exit', чтобы выйти): У меня есть черная кошка.

История: [

```
{
  "role": "system",
  "content": "Ты полезный помощник."
},
{
  "role": "user",
  "content": " У меня есть черная кошка."
}
```

]

Вывод: Это великолепно! Черные кошки часто ассоциируются с мистикой и сверхъестественным. Во многих культурах они также приносят удачу. Хотите ли вы узнать или рассказать что-то еще о черных кошках? Я здесь, чтобы помочь вам!

Ввод: (наберите 'exit', чтобы выйти): У меня есть белая рубашка.

История: [

```
{
  "role": "system",
  "content": "Ты полезный помощник."
},
{
  "role": "user",
  "content": " У меня есть черная кошка."
```

```
},
```

```
{
```

```
  "role": "assistant",
  "content": "Это великолепно! Черные кошки часто ассоциируются с мистикой
и сверхъестественным. Во многих культурах они также приносят удачу. Хотите ли вы
узнать или рассказать что-то еще о черных кошках? Я здесь, чтобы помочь вам!"
```

```
},
```

```
{
```

```
  "role": "user",
  "content": " У меня есть белая рубашка."
```

```
}
```

]

Вывод: Отлично! Белая рубашка – универсальный и классический предмет одежды, который можно сочетать с различными брюками и аксессуарами. Это модный предмет одежды, который можно носить как отдельно, так и с костюмом. Есть ли у вас какие-либо особые опасения или вопросы по поводу ухода за белой рубашкой или ее стиля? Не стесняйтесь спрашивать!

Ввод: (наберите 'exit', чтобы выйти): Какого цвета моя кошка?

История: [

```
{
  "role": "system",
  "content": "Ты полезный помощник."
```

```
},
```

```
{
```

```
  "role": "user",
  "content": "У меня есть черная кошка."
```

```
},
```

```

{
  "role": "user",
  "content": "У меня есть белая рубашка."
},
{
  "role": "user",
  "content": "Какого цвета моя кошка?"
}
]

```

Ответ: Ваша кошка черная.

Ввод: (наберите 'exit', чтобы выйти): Какого цвета моя рубашка?

История: [

```

{
  "role": "system",
  "content": "Ты полезный помощник."
},
{
  "role": "user",
  "content": "У меня есть белая рубашка."
},
{
  "role": "user",
  "content": "Какого цвета моя кошка?"
},
{
  "role": "user",
  "content": "Какого цвета моя рубашка?"
}
]

```

]

Вывод: Вы говорили раньше, что у вас есть белая рубашка. Следовательно, ваша рубашка белого цвета.

Этот код выбирает наиболее подходящий контекст для промпта пользователя, но его невозможно оптимизировать для всех случаев использования. Будьте внимательны при выборе количества учитываемых взаимодействий и алгоритма сортировки истории. В качестве потенциальных улучшений можно предложить следующие доработки:

- использование более продвинутого алгоритма, который переносит из контекста весь сегмент (промпты пользователя и помощника), а не только его часть;
- использование векторной базы данных вместо файла для хранения истории и выполнения сортировки;
- регулировка количества взаимодействий для повторения в зависимости от промпта пользователя и контекста;
- запрос более одного ответа от API и выбор на основе контекста. Встраивания можно использовать повторно для сравнения ответов с контекстом и выбора лучшего варианта;
- ...и т. д.

Применение векторной базы данных

ВВЕДЕНИЕ

В предыдущих примерах мы пытались заставить ИИ лучше понимать контекст диалога. Однако для хранения контекста мы использовали текстовые файлы, что является не самым эффективным методом. Поэтому далее мы будем использовать для хранения контекста векторную базу данных.

Существует множество популярных векторных баз данных, включая Faiss, Annoy, Milvus, Weaviate и т. д. В этом примере мы будем использовать Weaviate. Но прежде чем приступить к делу, давайте разберемся, что такое векторная база данных.

ЧТО ТАКОЕ ВЕКТОРНАЯ БАЗА ДАННЫХ?

Векторная база данных – это разновидность базы данных, предназначенная для хранения, индексирования и запроса векторов (массивов чисел, представляющих данные в многомерном пространстве). Векторы часто появляются в результате преобразования неструктурированных данных, таких как изображения, текст или звук, в числовой формат, который можно обрабатывать и анализировать с помощью моделей машинного обучения, особенно тех, которые включают алгоритмы поиска по сходству и алгоритмы ближайшего соседа.

Векторные базы данных отлично подходят для некоторых приложений искусственного интеллекта, таких как рекомендательные системы, поисковые системы и обработка естественного языка, где цель состоит в том, чтобы найти элементы, наиболее похожие на элемент запроса, на основе их векторных представлений.

Представьте, что вы находитесь в огромной библиотеке, полной книг, но вместо традиционных жанров и названий каждая книга представлена уникальным узором из звезд на небе. Каждый узор (вектор) представляет суть книги – ее тему, стиль и содержание, – переведенную на язык звезд, который может понять и интерпретировать только волшебный телескоп библиотеки (векторная база данных).

Отдельная книга в библиотеке служит аналогом фрагмента данных (документа, изображения или даже аудиоклипа) в реальном мире. Преобразование этих книг в звездные узоры аналогично преобразованию неструктурированных данных в векторы с использованием моделей машинного обучения, таких как модели GPT. Каждый вектор представляет собой числовое представление, отражающее суть данных в многомерном пространстве.

Этот телескоп особенный (считайте его аналогом векторной базы данных) – он может быстро сканировать ночное небо (выполнять поиск по базе данных) и находить закономерности узоров, похожие на те, которые вас интересуют.

Вы излагаете описание интересующей вас книги волшебному телескопу, и он преобразует ваше описание в звездный узор.

Допустим, вас увлекают истории о драконах. Расскажите это телескопу. Он создаст звездный узор, представляющий «истории о драконах», и просканирует небо в поисках похожих узоров. Созвездия, наиболее близкие к вашей «истории о драконах», – это книги (данные), наиболее соответствующие вашим интересам. Процесс поиска наиболее похожих созвездий аналогичен поиску по сходству в векторной базе данных, где система находит точки данных (векторы), ближайшие к вектору вашего поискового запроса.

Цель этой метафоры – помочь вам образно представить всю концепцию, от встраивания данных в векторное пространство до запроса к векторной базе данных.



Метафора, «интерпретированная» моделью DALL-E

Вот несколько примеров практического применения векторных баз данных:

- *рекомендательные системы*: векторная база данных может хранить пользовательские предпочтения и характеристики элементов в виде векторов. Когда пользователь ищет товар, система находит наиболее похожие векторные представления и рекомендует соответствующие товары;

- *распознавание голоса*: когда вы разговариваете с голосовым помощником, ваша речь преобразуется в вектор. Помощник ищет лучшее совпадение, чтобы понять ваши команды;
- *поиск изображений*: вы загружаете изображение автомобиля, и векторная база данных помогает вам найти другие изображения автомобилей. Система переводит визуальные признаки в векторы и сравнивает их с другими изображениями, которые также представлены в виде векторов в той же базе данных.

Векторные базы данных позволяют быстро и эффективно находить векторы, наиболее похожие на вектор запроса. Они используют специализированные структуры индексации для эффективного хранения и запроса многомерных векторов. Эти структуры, такие как KD-деревья¹, графы HNSW (hierarchical navigable small world)² или индексы на основе квантования, помогают уменьшить размерность и ускорить поиск ближайших соседей.

ПРИМЕР 1. ИСПОЛЬЗОВАНИЕ WEAVIATE ДЛЯ ПОВЫШЕНИЯ КОНТЕКСТНОЙ ЗАВИСИМОСТИ МОДЕЛИ

В этом примере мы создадим два Docker-контейнера: один для Weaviate, а другой для нашей модели ИИ. Для управления этими контейнерами мы будем использовать Docker Compose.

Мы создаем API с единой конечной точкой для взаимодействия с запросом (вопросом) пользователя. API отправляет запрос в Weaviate и получает оттуда наиболее похожий контекст. Затем этот контекст направляется в API OpenAI для получения окончательного ответа. Ответ передается пользователю, а также сохраняется в Weaviate для дальнейшего использования. Пользователь может видеть выходные данные API в браузере, терминале или любом другом клиенте, например Postman.

Начнем с установки Docker при помощи следующих команд:

```
# Скачивание скрипта установки
curl -fsSL https://get.docker.com \
-o get-docker.sh
```

```
# Запуск скрипта установки
sh get-docker.sh
```

Инструкции по установке для вашей операционной системы также можно найти на сайте Docker³.

Создайте новую виртуальную среду:

```
mkvirtualenv vector_db_example
```

Затем создайте структуру каталогов:

¹ https://en.wikipedia.org/wiki/K-d_tree.

² <https://www.youtube.com/watch?v=QvKMwLjdK-s>.

³ <https://docs.docker.com/engine/install/>.

```
mkdir src/vector_db_example
mkdir src/vector_db_example/app
mkdir src/vector_db_example/weaviate
```

Создайте первый Docker-файл в папке app:

```
cat << EOF > src/vector_db_example/app/Dockerfile.app
FROM python:3.9-bookworm
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . .
EXPOSE 5000
CMD [
    "flask",
    "--app",
    "app",
    "run",
    "--host=0.0.0.0",
    "--port=5000",
    "--debug",
    "--reload"
]
EOF
```

Теперь создайте Docker-файл для Weaviate в папке weaviate:

```
cat << EOF > src/vector_db_example/weaviate/Dockerfile.weaviate
FROM semitechnologies/weaviate:1.23.7
EXPOSE 8080
EXPOSE 50051
CMD [
    "--host",
    "0.0.0.0",
    "--port",
    "8080",
    "--scheme",
    "http"
]
EOF
```

Так выглядит файл Docker Compose:

```
cat << EOF > src/vector_db_example/docker-compose.yml
---
version: '3.9'

services:

  app:
    build:
      context: app
      dockerfile: Dockerfile.app
    env_file:
      - app/.env
```

```

ports:
  - "5000:5000"
volumes:
  - ./app:/app/

weaviate:
  build:
    context: weaviate
    dockerfile: Dockerfile.weaviate
  ports:
    - 8080:8080
    - 50051:50051
  env_file:
    - weaviate/.env

```

EOF

Нам также нужны переменные среды для обоих контейнеров.
Начнем с вашего ключа OpenAI:

```
export OPENAI_API_KEY=ваш_ключ_API
```

Создайте файл `.env` в папке `app`:

```
cat << EOF > src/vector_db_example/app/.env
OPENAI_API_KEY=${OPENAI_API_KEY}
EOF
```

Создайте другой файл `.env` в папке `weaviate`:

```
cat << EOF > src/vector_db_example/weaviate/.env
OPENAI_APIKEY=${OPENAI_API_KEY}
# Количество объектов, возвращаемых запросом по умолчанию
QUERY_DEFAULTS_LIMIT=25
# Разрешить взаимодействие пользователей с weaviate без авторизации
AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED=true
# Где Weaviate Standalone будет хранить данные?
PERSISTENCE_DATA_PATH=/var/lib/weaviate
# Модуль векторизации по умолчанию
DEFAULT_VECTORIZER_MODULE=text2vec-openai
# Какие модули включить в установку?
ENABLE_MODULES=text2vec-openai
# Имя хоста экземпляра weaviate
CLUSTER_HOSTNAME=node1
EOF
```

В данной конфигурации мы включаем в Weaviate модуль `text2vec-openai`. Этот модуль позволяет Weaviate получать векторное представление с помощью API OpenAI (или Azure OpenAI).

Так выглядит файл `requirements.txt` для приложения:

```
cat << EOF > src/vector_db_example/app/requirements.txt
annotated-types==0.6.0
anyio==4.2.0
Authlib==1.3.0
blinker==1.7.0
certifi==2024.2.2
```

```

cffi==1.16.0
charset-normalizer==3.3.2
click==8.1.7
cryptography==42.0.2
distro==1.9.0
exceptiongroup==1.2.0
flask==3.0.2
grpcio==1.60.1
grpcio-health-checking==1.60.1
grpcio-tools==1.60.1
h11==0.14.0
httpcore==1.0.2
httpx==0.26.0
idna==3.6
importlib-metadata==7.0.1
itsdangerous==2.1.2
Jinja2==3.1.3
MarkupSafe==2.1.5
openai==1.11.1
protobuf==4.25.2
pycparser==2.21
pydantic==2.6.1
pydantic-core==2.16.2
requests==2.31.0
sniffio==1.3.0
tqdm==4.66.1
typing-extensions==4.9.0
urllib3==2.2.0
validators==0.22.0
weaviate-client==4.4.2
werkzeug==3.0.1
zipp==3.17.0
EOF

```

Мы установили `weaviate-client`, `openai` и `flask`, указанные в файле `requirements.txt`.

Flask – это микрофреймворк для создания веб-приложений на языке Python. Он отлично подходит для быстрого прототипирования веб-приложений или REST API.

Вам осталось добавить код приложения с именем `app.py` в папку `src/vector_db_example/app`.

Дерево проекта на вашем компьютере должно выглядеть так:

```

src/vector_db_example/
├── app
│   ├── app.py
│   ├── Dockerfile.app
│   ├── .env
│   └── requirements.txt
├── weaviate
├── Dockerfile.weaviate
└── .env

```

Теперь займемся созданием центральной части приложения – файла `app.py`.

API сохраняет полученный запрос пользователя в переменной с именем `question`, а потом формирует словарь с запросом и ролью запроса.

```
question = request.args.get("q")

user_prompt = {
    "role": "user",
    "content": question
}
```

Затем мы используем специальную функцию для получения ближайшего контекста из Weaviate. Эта функция называется `weaviate_nearest_interactions` и определяется в следующем коде:

```
context = weaviate_nearest_interactions(
    question,
    weaviate_certainty,
    weaviate_limit,
)
```

В некоторых случаях контекст может быть пустым или недостаточным для получения хорошего ответа от OpenAI. Вот почему мы стараемся добавить в контекст последние взаимодействия между пользователем и ИИ. Для получения последних взаимодействий из Weaviate служит функция `weaviate_latest_interactions`. Эта функция тоже объявлена в коде:

```
latest_interactions = weaviate_latest_interactions(
    interactions_limit
)
```

Мы также соединяем два контекста и удаляем дубликаты:

```
global_context = latest_interactions["data"] + context["data"]
# Получаем уникальные значения
global_context = [
    dict(t) for t in {
        tuple(d.items()) for d in global_context
    }
]
```

Теперь осталось отправить сообщение в OpenAI и получить ответ. Сообщение будет содержать системный промпт (такой как «Ты дружелюбный ИИ») и глобальный контекст.

```
response = openai_client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=200,
    temperature=1.2,
)

content = response.choices[0].message.content.strip()
```

Промпт пользователя и ответ будут сохранены в Weaviate для использования в будущем. Мы будем использовать функцию `weaviate_save_data` для хранения данных в Weaviate. Эта функция будет определена позже в коде.

```
assistant_prompt = {
    "role": "assistant",
    "content": content
}
```

```
data = [
    user_prompt,
    assistant_prompt
]
```

```
weaviate_save_data(
    data
)
```

Наконец, нужно вернуть ответ пользователю. Для удобства отладки мы также будем возвращать глобальный контекст.

```
return {
    "response": assistant_prompt["content"],
    "global_context": global_context,
}
```

Вот полный код функции, которую мы будем использовать в файле `app.py`:

```
@app.route("/ask", methods=["GET"])
def ask():
    # Получение вопроса от пользователя
    question = request.args.get("q")

    # Построение промпта пользователя
    user_prompt = {
        "role": "user",
        "content": question
    }

    # Получение ближайшего контекста из Weaviate
    context = weaviate_nearest_interactions(
        question,
        weaviate_certainty,
        weaviate_limit,
    )

    # Получение последних взаимодействий из Weaviate
    latest_interactions = weaviate_latest_interactions(
        interactions_limit
    )

    # Слияние двух контекстов
    # и получение уникальных значений
    global_context = latest_interactions["data"] + context["data"]
```

```

global_context = [
    dict(t) for t in {
        tuple(d.items()) for d in global_context
    }
]

# Формирование сообщения для отправки в OpenAI
messages = [system_prompt] + global_context + [user_prompt]
# Отправка сообщения в OpenAI
response = openai_client.chat.completions.create(
    model=model,
    messages=messages,
    max_tokens=200,
    temperature=1.2,
)

content = response.choices[0].message.content.strip()

# Сохранение промпта пользователя и ответа в Weaviate
assistant_prompt = {
    "role": "assistant",
    "content": content
}

data = [
    user_prompt,
    assistant_prompt
]

weaviate_save_data(
    data
)

# Возвращаем ответ пользователю
return {
    "response": assistant_prompt["content"],
    "global_context": global_context,
}

```

Теперь определим вышеупомянутые функции, которые будем использовать в приложении `app.py`.

```

def weaviate_nearest_interactions(query, certainty, limit):
    try:
        # получаем ближайший контекст из Weaviate
        result = weaviate_client.query.get(
            class_name=weaviate_class_name,
            properties=[
                "role",
                "content"
            ]
        ).with_near_text({
            "concepts": [query],
            "certainty": certainty

```

```

    }).with_limit(
        limit
    ).do()

    return {
        "data": result['data']['Get'][weaviate_class_name]
    }

except Exception as e:
    app.logger.error(f"Error while searching: {e}")

def weaviate_lastest_interactions(limit):
    try:
        # Получаем последние взаимодействия из Weaviate
        result = weaviate_client.query.get(
            class_name=weaviate_class_name,
            properties=[
                "role",
                "content"
            ]
        ).with_limit(
            limit
        ).do()

        return {
            "data": result['data']['Get'][weaviate_class_name]
        }

    except Exception as e:
        app.logger.error(f"Error while searching: {e}")

```

Базы данных обычно имеют схему, и Weaviate не исключение. Нам нужно определить схему базы данных:

```

schema = {
    "classes": [
        {
            "class": weaviate_class_name,
            "description": "Класс сохранения сообщений чата",
            "properties": [
                {
                    "name": "content",
                    "description": "Контент сообщения чата",
                    "dataType": ["text"],
                },
                {
                    "name": "role",
                    "description": "Роль сообщения",
                    "dataType": ["string"],
                },
            ],
        },
    ],
}

```

```
def weaviate_create_schema():
    try:
        # Создаем схему в Weaviate
        weaviate_client.schema.create(schema)
        app.logger.info("Схема успешно создана.")
    except Exception as e:
        app.logger.error(f"Ошибка создания схемы: {e}")
```

Когда пользователь начинает сессию, мы должны создать схему в Weaviate и удалить предыдущие данные. Поэтому нужно определить функцию `weaviate_delete_data`:

```
def weaviate_delete_data():
    try:
        weaviate_client.schema.delete_class(
            class_name=weaviate_class_name
        )
        app.logger.info("Данные успешно сброшены.")
    except Exception as e:
        app.logger.error(f"Ошибка при удалении класса: {e}")
        return {"ошибка в weaviate_reset": str(e)}, 500
```

Переменная `weaviate_class_name` определена как глобальная:

```
weaviate_class_name = "ChatMessage"
```

Хорошим решением будет создание уникального имени класса, такого как `ChatMessage-{session_id}`.

Собрав воедино все вышеприведенные части, мы получим приложение `app.py`, полный код которого приведен ниже и доступен в файловом архиве в папке `src/vector_db_example/app/`.

```
cat << EOF > src/vector_db_example/app/app.py
import weaviate, os
from flask import Flask, request
from openai import OpenAI

app = Flask(__name__)

openai_api_key = os.getenv("OPENAI_API_KEY")
system_prompt = "Ты полезный помощник"
system_prompt = {
    "role": "system",
    "content": system_prompt
}
model = "gpt-3.5-turbo"
weaviate_class_name = "ChatMessage"
weaviate_limit = 10
interactions_limit = 10
weaviate_certainty = 0.5

openai_client = OpenAI(
    api_key=openai_api_key
)
```

```

weaviate_client = weaviate.Client(
    url="http://weaviate:8080",
    auth_client_secret={
        "X-OpenAI-API-Key": openai_api_key
    }
)

schema = {
    "classes": [
        {
            "class": weaviate_class_name,
            "description": "Класс сохранения сообщений чата",
            "properties": [
                {
                    "name": "content",
                    "description": "Контент сообщения чата",
                    "dataType": ["text"],
                },
                {
                    "name": "role",
                    "description": "Роль сообщения",
                    "dataType": ["string"],
                },
            ],
        }
    ]
}

def weaviate_create_schema():
    try:
        # Создаем схему в Weaviate
        weaviate_client.schema.create(schema)
        app.logger.info("Схема успешно создана.")
    except Exception as e:
        app.logger.error(f"Ошибка создания схемы: {e}")

def weaviate_delete_data():
    try:
        weaviate_client.schema.delete_class(
            class_name=weaviate_class_name
        )
        app.logger.info("Данные успешно сброшены.")
    except Exception as e:
        app.logger.error(f"Ошибка при удалении класса: {e}")
        return {"ошибка в weaviate_reset": str(e)}, 500

weaviate_delete_data()
weaviate_create_schema()

def weaviate_nearest_interactions(query, certainty, limit):
    try:
        result = weaviate_client.query.get(

```

```
        class_name=weaviate_class_name,
        properties=[
            "role",
            "content"
        ]
    ).with_near_text({
        "concepts": [query],
        "certainty": certainty
    }).with_limit(
        limit
    ).do()

    return {
        "data": result['data']['Get'][weaviate_class_name]
    }

except Exception as e:
    app.logger.error(f"Ошибка при поиске: {e}")

def weaviate_latest_interactions(limit):
    try:
        result = weaviate_client.query.get(
            class_name=weaviate_class_name,
            properties=[
                "role",
                "content"
            ]
        ).with_limit(
            limit
        ).do()

        return {
            "data": result['data']['Get'][weaviate_class_name]
        }

    except Exception as e:
        app.logger.error(f"Ошибка при поиске: {e}")

def weaviate_save_data(data):
    weaviate_client.batch.configure(
        batch_size=100
    )
    with weaviate_client.batch as batch:
        for _, d in enumerate(data):
            properties = {
                "role": d["role"],
                "content": d["content"]
            }
            batch.add_data_object(
                properties,
                weaviate_class_name,
            )
        )
```

```
@app.route("/ask", methods=["GET"])
def ask():
    question = request.args.get("q")

    user_prompt = {
        "role": "user",
        "content": question
    }

    context = weaviate_nearest_interactions(
        question,
        weaviate_certainty,
        weaviate_limit,
    )

    latest_interactions = weaviate_latest_interactions(
        interactions_limit
    )

    global_context = latest_interactions["data"] + context["data"]
    # Получение уникальных значений
    global_context = [
        dict(t) for t in {
            tuple(d.items()) for d in global_context
        }
    ]

    messages = [system_prompt] + global_context + [user_prompt]

    response = openai_client.chat.completions.create(
        model=model,
        messages=messages,
        max_tokens=200,
        temperature=1.2,
    )

    content = response.choices[0].message.content.strip()

    assistant_prompt = {
        "role": "assistant",
        "content": content
    }

    data = [
        user_prompt,
        assistant_prompt
    ]

    weaviate_save_data(
        data
    )
```

```

    return {
        "response": assistant_prompt["content"],
        "global_context": global_context,
    }
EOF

```

Если вы хотите проверить содержимое базы данных в целях отладки, вы также можете добавить следующую функцию в файл `app.py`:

```

@app.route("/data", methods=["GET"])
def weaviate_get_all_data():
    try:
        result = weaviate_client.query.get(
            class_name=weaviate_class_name,
            properties=[
                "role",
                "content"
            ]
        ).do()

        return {
            "data": result['data']['Get'][weaviate_class_name]
        }

    except Exception as e:
        app.logger.error(f"Ошибка извлечения данных: {e}")
        return {"ошибка в weaviate_data": str(e)}, 500

```

Теперь нам нужно собрать контейнеры Docker и запустить приложение. Это можно сделать при помощи следующих команд:

```

cd src/vector_db_example
docker-compose up --build

```

API приложения доступно в браузере по адресу `http://localhost:5000/ask?q=<your-question>`.

В качестве наглядной демонстрации можно начать со следующего примера:

```

# У меня есть кошка
curl http://127.0.0.1:5000/ask?q=I%20have%20a%20cat
# У меня есть собака
curl http://127.0.0.1:5000/ask?q=I%20have%20a%20dog
# Сколько животных у меня есть?
curl http://127.0.0.1:5000/ask?q=How%20many%20pets%20do%20I%20have%3F

```

Вывод для этого примера должен выглядеть примерно так (без перевода):

```

{
  "global_context": [
    {
      "content": "I have a dog",
      "role": "user"
    },
    {
      "content":

```

```

        "That's wonderful! Cats make great companions. "
        "Is there anything specific you need help with "
        "regarding your cat?",
        "role": "assistant"
    },
    {
        "content":
            "That's great! Dogs are wonderful pets too. "
            "Is there anything specific you need help "
            "with regarding your dog?",
            "role": "assistant"
    },
    {
        "content": "I have a cat",
        "role": "user"
    }
],
"response":
    "Based on your previous message, you mentioned "
    "having a dog. If you have a cat in addition to "
    "a dog, then you would have two pets."
}

```

ПРИМЕР 2. СЕМАНТИЧЕСКИЙ ПОИСК С ПОМОЩЬЮ WEAVIATE И OPENAI

В этом примере мы воспользуемся комбинацией Weaviate и OpenAI для создания системы семантического поиска. Weaviate будет хранить данные, а также векторы, представляющие данные. Задача OpenAI – семантический поиск по векторам.

Пример основан на наборе данных в формате CSV, содержащем статьи из Simple English Wikipedia. Набор данных размещен на OpenAI¹. Кроме того, вы можете загрузить набор данных из репозитория этой книги на GitHub² или получить в файловом архиве на сайте издательства русского перевода.

CSV-файл содержит следующие столбцы:

- id: уникальный идентификатор записи данных;
- url: URL-адрес страницы Википедии;
- title: название статьи;
- text: содержание статьи;
- content_vector: векторное представление контента.

Кстати, векторизация была выполнена с использованием модели OpenAI text-embedding-ada-002.

Начнем с создания новой виртуальной среды:

```
mkvirtualenv vector_db_semantic_search
```

¹ https://cdn.openai.com/API/examples/data/vector_database_wikipedia_articles_embedded.zip.

² https://github.com/eon01/Openai_GPT_for_Python_Developers_Files.

Создадим для этого примера новую папку:

```
mkdir src/vector_db_semantic_search
mkdir -p src/vector_db_semantic_search/app/data
mkdir src/vector_db_semantic_search/weaviate
```

Скачайте набор данных, распакуйте его и сохраните файл по адресу `src/vector_db_semantic_search/app/data/data.csv`.

В качестве альтернативы я рекомендую загрузить часть набора данных, поскольку полный набор данных довольно велик. Загрузка, извлечение и передача в Weaviate займут некоторое время. Вы можете загрузить подмножество, используя следующие команды:

```
# Download the dataset
curl -L \
https://raw.githubusercontent.com/\
eon01/Openai_GPT_for_Python_Developers_Files/\
main/datasets/\
vector_database_wikipedia_articles_embedded/data.csv \
-o \
src/vector_db_semantic_search/app/data/data.csv
```

Все, что мы сделали в предыдущем примере (оба файла `Docker` и `.env`, а также файлы `dockercompose.yml` и `requirements.txt`). Изменения коснутся лишь файла приложения.

В приложении нужно создать другую схему:

```
article_class = {
  "class": weaviate_class_name,
  "description":
    "Статья из набора Simple English Wikipedia",
  "vectorizer": "text2vec-openai",
  "moduleConfig": {
    # Инструмент, которым OpenAI создавал
    # встраивания для поля `content` (`text`)
    "text2vec-openai": {
      "model": "ada",
      "modelVersion": "002",
      "type": "text",
      "vectorizeClassName": False
    }
  },
  "properties": [
    {
      "name": "title",
      "description": "Заголовок статьи",
      "dataType": ["text"],
      # Не векторизовать заголовок
      "moduleConfig": {"text2vec-openai": {"skip": True}}
    },
    {
      "name": "content",
      "description": "Тело статьи",
```

```

        "dataType": ["text"],
    }
]
}

```

Далее импортируем данные в Weaviate с помощью следующей функции:

```

def weaviate_import_data():
    # Счетчик для отображения линии прогресса в консоли
    counter = 0
    interval = 100

    csv_iterator = pd.read_csv(
        'data/data.csv',
        usecols=[
            'id',
            'url',
            'title',
            'text',
            'content_vector'
        ],
        # Количество строк в пакете
        chunksize=100,
        # Лимит количества строк при импорте
        nrows=100
    )

    # Конфигурация пакета
    weaviate_client.batch.configure(
        batch_size=100
    )

    with weaviate_client.batch as batch:
        for chunk in csv_iterator:
            for _, row in chunk.iterrows():

                properties = {
                    "title": row.title,
                    "content": row.text,
                    "url": row.url
                }

                # Конвертация вектора из строки CSV
                # в массив с плавающей запятой
                vector = ast.literal_eval(
                    row.content_vector
                )

                # Добавляем объект в пакет
                # и задаем его векторное представление
                batch.add_data_object(
                    properties,
                    class_name=weaviate_class_name,

```

```
        vector=vector
    )

    # Вычисление и отображение хода загрузки
    counter += 1
    if counter % interval == 0:
        app.logger.debug(f"Imported {counter} articles...")
    app.logger.debug(f"Finished importing {counter} articles.")
```

Так определяется главный путь к API:

```
@app.route("/ask", methods=["GET"])
def ask():
    question = request.args.get("q")

    context = weaviate_semantic_search(
        question,
    )

    return {
        "response": context
    }
```

Когда пользователь отправляет запрос, API пересылает его в Weaviate для поиска наиболее похожих статей. Затем пользователю возвращается ответ. Это делается с помощью функции `weaviate_semantic_search()`.

```
def weaviate_semantic_search(query):
    nearText = {
        "concepts": [query],
    }

    properties = [
        "title",
        "content",
        "_additional {distance}"
    ]

    limit = 2

    response = weaviate_client.query.get(
        class_name=weaviate_class_name,
        properties=properties,
    ).with_near_text(
        nearText
    ).with_limit(
        limit
    ).do()

    result = response['data']['Get'][weaviate_class_name]
    return result
```

Функция `weaviate_semantic_search()` предназначена для выполнения семантического поиска в базе данных Weaviate с использованием заданного пользователем запроса.

По своей сути функция использует возможности векторного поиска Weaviate для поиска записей, которые семантически связаны с входным запросом. Создавая словарь `nearText` с запросом, инкапсулированным в список под ключом `concepts`, она предписывает Weaviate оценить семантическую близость записей базы данных к предоставленным концептам запроса.

Функция запрашивает свойства `title` и `content` совпадающих записей, а также дополнительную информацию под именем `distance`. Свойство `distance` имеет решающее значение, поскольку оно сообщает, насколько близка каждая возвращаемая запись к запросу в векторном пространстве, при этом меньшее значение `distance` указывает на большее сходство.

Параметр `limit`, равный 2, ограничивает поиск, разрешая возвращать только две верхние наиболее релевантные записи.

Отдельного внимания заслуживает включение в список свойства `_.additional {distance}`, поскольку оно дает представление о процессе семантического поиска, раскрывая необработанные расстояния, рассчитанные во время поиска. Это дает представление о том, почему определенные записи были сочтены более релевантными, чем остальные.

Ниже приведен полный код приложения `app.py`:

```
cat << EOF > src/vector_db_semantic_search/app/app.py
import weaviate, os, ast
from flask import Flask, request
from openai import OpenAI
import pandas as pd

app = Flask(__name__)

openai_api_key = os.getenv("OPENAI_API_KEY")
model = "gpt-3.5-turbo"
weaviate_class_name = "Article"

openai_client = OpenAI(
    api_key=openai_api_key
)

weaviate_client = weaviate.Client(
    url="http://weaviate:8080",
    auth_client_secret={
        "X-OpenAI-API-Key": openai_api_key
    }
)

article_class = {
    "class": weaviate_class_name,
    "description":
        "Статья из набора Simple English Wikipedia",
    "vectorizer": "text2vec-openai",
    "moduleConfig": {
        # Инструмент, которым OpenAI создавал
        # встраивания для поля `content` (`text`)
    }
}
```

```

    "text2vec-openai": {
      "model": "ada",
      "modelVersion": "002",
      "type": "text",
      "vectorizeClassName": False
    }
  },
  "properties": [
    {
      "name": "title",
      "description": "Заголовок статьи",
      "dataType": ["text"],
      # Не векторизовать заголовок
      "moduleConfig": {"text2vec-openai": {"skip": True}}
    },
    {
      "name": "content",
      "description": "Тело статьи",
      "dataType": ["text"],
    }
  ]
}

```

```

def weaviate_import_data():
    # Счетчик для отображения линии прогресса в консоли
    counter = 0
    interval = 100

    csv_iterator = pd.read_csv(
        'data/data.csv',
        usecols=[
            'id',
            'url',
            'title',
            'text',
            'content_vector'
        ],
        # Количество строк в пакете
        chunksize=100,
        # Лимит количества строк при импорте
        nrows=100
    )

    # Конфигурация пакета
    weaviate_client.batch.configure(
        batch_size=100
    )

    with weaviate_client.batch as batch:
        for chunk in csv_iterator:
            for _, row in chunk.iterrows():

```

```

        properties = {
            "title": row.title,
            "content": row.text,
            "url": row.url
        }

        # Конвертация вектора из строки CSV
        # в массив с плавающей запятой
        vector = ast.literal_eval(
            row.content_vector
        )

        # Добавляем объект в пакет
        # и задаем его векторное представление
        batch.add_data_object(
            properties,
            class_name=weaviate_class_name,
            vector=vector
        )

        # Вычисление и отображение хода загрузки
        counter += 1
        if counter % interval == 0:
            app.logger.debug(f"Импортировано {counter} статей...")
        app.logger.debug(f"Завершен импорт {counter} статей.")

def weaviate_semantic_search(query):
    nearText = {
        "concepts": [query],
    }

    properties = [
        "title",
        "content",
        "_additional {distance}"
    ]

    limit = 2

    response = weaviate_client.query.get(
        class_name=weaviate_class_name,
        properties=properties,
    ).with_near_text(
        nearText
    ).with_limit(
        limit
    ).do()

    result = response['data']['Get'][weaviate_class_name]
    return result

weaviate_client.schema.delete_all()

```

```
weaviate_import_data()

@app.route("/ask", methods=["GET"])
def ask():
    question = request.args.get("q")

    context = weaviate_semantic_search(
        question,
    )

    return {
        "response": context
    }
EOF
```

Теперь соберем контейнеры Docker и запустим приложение:

```
cd src/vector_db_semantic_search
docker-compose up --build
```

Вы можете обратиться к API через браузер по адресу <http://localhost:5000/ask?q=<your-question>>. Вот пример, с которого можно начать:

```
# поисковый запрос: политика
curl http://0.0.0.0:5000/ask?q=politics
# поисковый запрос: наука
curl http://0.0.0.0:5000/ask?q=science
# поисковый запрос: кошки
curl http://0.0.0.0:5000/ask?q=cats
```

ПРИМЕР 3. ГЕНЕРАТИВНЫЙ ПОИСК С ПОМОЩЬЮ WEAVIATE И OPENAI

Генеративный поиск – это тип поиска, который генерирует новый контент на основе запроса пользователя.

Когда мы ищем слово или фразу в поисковой системе, мы обычно получаем список результатов, которые их содержат. Однако в некоторых случаях желательно создать новый контент на основе запроса. Например, если мы ищем ответ на вопрос «как приготовить торт?», нам хотелось бы получить несколько рецептов торта вместо списка веб-сайтов, содержащих фразу «как приготовить торт».

Тот же принцип применим и к нашему примеру. Например, когда мы ищем слово «кошка», модель могла бы создать новую статью о кошках на основе запроса «кошка» и возвращенного контекста. Weaviate вернет наиболее релевантные статьи на основе запроса и отправит сообщение в OpenAI для создания новой статьи на основе контекста.

В этом примере мы будем использовать прежние Docker-файлы, файл `.env` приложения, а также файлы `docker-compose.yml` и `requirements.txt`. Изменятся только две вещи:

- файл `.env` контейнера Weaviate;
- файл `app.py`.

Начнем с создания файла `.env` для контейнера Weaviate:

```
cat << EOF > src/vector_db_generative_search/weaviate/.env
OPENAI_APIKEY=$OPENAI_API_KEY
QUERY_DEFAULTS_LIMIT=25
AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED=true
PERSISTENCE_DATA_PATH=/var/lib/weaviate
DEFAULT_VECTORIZER_MODULE=text2vec-openai
ENABLE_MODULES=text2vec-openai,generative-openai
CLUSTER_HOSTNAME=node1
EOF
```

Как вы можете заметить, мы добавили `generative-openai` в переменную `ENABLE_MODULES`. Это действие включает генеративный модуль в Weaviate.

Файл `app.py` не будет сильно отличаться от предыдущего примера. Мы просто добавим новый вызов для создания свежего контента на основе контекста, возвращаемого Weaviate, и приглашения.

Например, если мы хотим, чтобы OpenAI переписал возвращаемую статью в стиле Шекспира, мы можем использовать следующий код:

```
prompt = """Перепиши это в стиле Шекспира:
{content}
"""

def weaviate_semantic_search(query, prompt):
    nearText = {
        "concepts": [query],
    }

    properties = [
        "title",
        "content",
        "_additional {distance}"
    ]

    limit = 1

    response = weaviate_client.query.get(
        class_name=weaviate_class_name,
        properties=properties,
    ).with_generate(
        single_prompt=prompt
    ).with_near_text(
        nearText
    ).with_limit(
        limit
    ).do()

    result = response['data']['Get'][weaviate_class_name]
    return result
```

Промптом может служить любой запрос. Например, чтобы перевести контент на валлийский язык, используйте следующий запрос:

```
prompt = """Translate to Welsh:
{content}
"""
```

Здесь вы видите, что промпт содержит подстановочное поле для контента ({content}). Вы можете добавить в промпт любое другое подстановочное поле, такое как {title} или {url}, если оно присутствует в контексте, который возвращает Weaviate.

Ниже показан полный файл приложения app.py для этого примера. Создайте необходимые папки и файлы (или скопируйте их из предыдущего примера).

```
cat << 'EOT' > src/vector_db_generative_search/app/app.py
import weaviate, os, ast
from flask import Flask, request
from openai import OpenAI
import pandas as pd

app = Flask(__name__)

openai_api_key = os.getenv("OPENAI_API_KEY")
model = "gpt-3.5-turbo"
weaviate_class_name = "Article"

openai_client = OpenAI(
    api_key=openai_api_key
)

weaviate_client = weaviate.Client(
    url="http://weaviate:8080",
    auth_client_secret={
        "X-OpenAI-API-Key": openai_api_key
    }
)

article_class = {
    "class": weaviate_class_name,
    "description":
        "Статья из набора Simple English Wikipedia",
    "vectorizer": "text2vec-openai",
    "moduleConfig": {
        # Инструмент, которым OpenAI создавал
        # встраивания для поля `content` (`text`)
        "text2vec-openai": {
            "model": "ada",
            "modelVersion": "002",
            "type": "text",
            "vectorizeClassName": False
        }
    },
    "properties": [
        {
            "name": "title",
            "description": "Заголовок статьи",
```



```
# Добавляем объект в пакет
# и задаем его векторное представление
batch.add_data_object(
    properties,
    class_name=weaviate_class_name,
    vector=vector
)

# Вычисление и отображение хода загрузки
counter += 1
if counter % interval == 0:
    app.logger.debug(f"Импортировано {counter} статей...")
app.logger.debug(f"Завершен импорт {counter} статей.")

def weaviate_semantic_search(query):
    nearText = {
        "concepts": [query],
    }

    properties = [
        "title",
        "content",
        "_additional {distance}"
    ]

    limit = 1

    response = weaviate_client.query.get(
        class_name=weaviate_class_name,
        properties=properties,
    ).with_near_text(
        nearText
    ).with_limit(
        limit
    ).do()

    result = response['data']['Get'][weaviate_class_name]
    return result

weaviate_client.schema.delete_all()
weaviate_import_data()

@app.route("/ask", methods=["GET"])
def ask():
    question = request.args.get("q")
    prompt = """Перепиши это в стиле Шекспира:
{content}
"""

    context = weaviate_semantic_search(
        question,
        prompt
    )
```

```

    return {
        "response": context
    }
EOT

```

Теперь соберем контейнеры Docker и запустим приложение:

```

cd src/vector_db_generative_search
docker-compose up --build

```

Доступ к API можно получить при помощи браузера по адресу `http://localhost:5000/ask?q=<your-question>`. Вот несколько примеров для начала:

```

# поисковый запрос: политика
curl http://0.0.0.0:5000/ask?q=politics
# поисковый запрос: наука
curl http://0.0.0.0:5000/ask?q=science
# поисковый запрос: кошки
curl http://0.0.0.0:5000/ask?q=cats

```

В каждой из приведенных выше команд API вернет статью, сгенерированную на основе поискового запроса и переписанную в стиле Шекспира.

Мы использовали `limit=1` в функции `weaviate_semantic_search`, чтобы вернуть только одну статью. Вы можете изменить лимит, чтобы возвращать больше статей и генерировать больше контента в зависимости от контекста. В этом случае вы получите несколько ответов от OpenAI.

В некоторых случаях нам нужно получить от OpenAI один ответ, но с несколькими входными данными (в нашем случае – статьи). Допустим, вы захотели извлечь наиболее важную информацию из нескольких статей и создать новую статью на основе извлеченной информации. В этом случае вы можете использовать параметр `grouped_task` в функции `with_generate`:

```

response = weaviate_client.query.get(
    class_name=weaviate_class_name,
    properties=properties,
).with_generate(
    grouped_task=prompt
).with_near_text(
    nearText
).with_limit(
    limit
)1.do()

```

Рассмотрим пример, в котором мы просим OpenAI составить список тем статей, возвращенных Weaviate. Для этого воспользуемся следующим промптом:

```

prompt = """Составь список тем, обсуждаемых в этих статьях:
{content}
"""

```

Так выглядит новый вариант кода:

```

cat << 'EOT' > src/vector_db_generative_search/app/app_new.py
import weaviate, os, ast
from flask import Flask, request

```

```

from openai import OpenAI
import pandas as pd

app = Flask(__name__)

openai_api_key = os.getenv("OPENAI_API_KEY")
model = "gpt-3.5-turbo"
weaviate_class_name = "Article"

openai_client = OpenAI(
    api_key=openai_api_key
)

weaviate_client = weaviate.Client(
    url="http://weaviate:8080",
    auth_client_secret={
        "X-OpenAI-API-Key": openai_api_key
    }
)

article_class = {
    "class": weaviate_class_name,
    "description":
        "Статья из набора Simple English Wikipedia",
    "vectorizer": "text2vec-openai",
    "moduleConfig": {
        # Инструмент, которым OpenAI создавал
        # встраивания для поля `content` (`text`)
        "text2vec-openai": {
            "model": "ada",
            "modelVersion": "002",
            "type": "text",
            "vectorizeClassName": False
        }
    },
    "properties": [
        {
            "name": "title",
            "description": "Заголовок статьи",
            "dataType": ["text"],
            # Не векторизуем заголовок
            "moduleConfig": {"text2vec-openai": {"skip": True}}
        },
        {
            "name": "content",
            "description": "Тело статьи",
            "dataType": ["text"],
        }
    ]
}

def weaviate_import_data():

```

```

# Счетчик для отображения линии прогресса в консоли
counter = 0
interval = 100

csv_iterator = pd.read_csv(
    'data/data.csv',
    usecols=[
        'id',
        'url',
        'title',
        'text',
        'content_vector'
    ],
    # Количество строк в пакете
    chunksize=100,
    # Лимит количества строк при импорте
    nrows=100
)

# Конфигурация пакета
weaviate_client.batch.configure(
    batch_size=100
)

with weaviate_client.batch as batch:
    for chunk in csv_iterator:
        for _, row in chunk.iterrows():

            properties = {
                "title": row.title,
                "content": row.text,
                "url": row.url
            }

            # Конвертация вектора из строки CSV
            # в массив с плавающей запятой
            vector = ast.literal_eval(
                row.content_vector
            )

            # Добавляем объект в пакет
            # и задаем его векторное представление
            batch.add_data_object(
                properties,
                class_name=weaviate_class_name,
                vector=vector
            )

            # Вычисление и отображение хода загрузки
            counter += 1
            if counter % interval == 0:
                app.logger.debug(f"Импортировано {counter} статей...")
            app.logger.debug(f"Завершен импорт {counter} статей.")

```

```
def weaviate_semantic_search(query):
    nearText = {
        "concepts": [query],
    }

    properties = [
        "title",
        "content",
        "_additional {distance}"
    ]

    limit = 3

    response = weaviate_client.query.get(
        class_name=weaviate_class_name,
        properties=properties,
    ).with_near_text(
        nearText
    ).with_limit(
        limit
    ).do()

    result = response['data']['Get'][weaviate_class_name]
    return result

weaviate_client.schema.delete_all()
weaviate_import_data()

@app.route("/ask", methods=["GET"])
def ask():
    question = request.args.get("q")
    prompt = """Составь список тем, обсуждаемых в этих статьях:
    {content}
    """

    context = weaviate_semantic_search(
        question,
        prompt
    )

    return {
        "response": context
    }
EOT
```

Если задать поиск по слову «animal» (животное), API вернет список тем, обсуждаемых в статьях про животных.

Пример (без перевода, соответствует оригинальному набору данных):

```
{
  "response": [
    {
      "_additional": {
```

```

        "distance": 0.17569739,
        "generate": {
            "error": null,
            "groupedResult": "The list of topics discussed in these articles are:\n\
n1\
. Animals (zoology, palaeontology, cellular respiration, metabolism, cell membranes)\
\n2. Plants (multicellular eukaryotic organisms)\n3. Grouping animals..."
        }
    },
    "content": "Animals (or Metazoa) are living creatures with many cells...",
    "title": "Animal"
},
{
    "_additional": {
        "distance": 0.21414834,
        "generate": null
    },
    "content": "A browser is a name given to any animal, usually a herbivorous
mam\
mal ..",
    "title": "Browser"
},
{
    "_additional": {
        "distance": 0.22182149,
        "generate": null
    },
    "content": "Being is also a present tense part of to be..",
    "title": "Being"
}
]
}

```

Генеративный поиск может пригодиться при создании контекстно-зависимых приложений. Вот вам упражнение: переделайте пример чат-бота, обсуждавшийся ранее, для применения генеративного поиска. Начните с пустой коллекции (класса) Weaviate и сохраните сообщения чата в Weaviate.

Используйте генеративный поиск для создания новых сообщений чата на основе контекста. Примените групповую задачу и начните с промпта, наподобие показанного ниже, протестируйте его, посмотрите результаты, протестируйте еще раз и повторяйте процесс, пока не достигнете желаемых результатов:

```

prompt = """Ты полезный помощник. Ты ведешь дискуссию с пользователем. Вот контекст
дискуссии:
{content}
""

```

Распознавание и перевод речи с Whisper

Что такое WHISPER?

Whisper – это система ASR (автоматическое распознавание речи) и модель распознавания речи общего назначения, обученная OpenAI на 680 000 часов многоязычных и многозадачных контролируемых данных, собранных из интернета.

OpenAI сообщает, что использование большого и разнообразного набора данных повысило устойчивость к акцентам, фоновому шуму и технической терминологии. Кроме того, модель позволяет выполнять транскрипцию на нескольких языках, а также переводить с этих языков на английский.

Поддерживаемые языки: африкаанс, арабский, армянский, азербайджанский, белорусский, боснийский, болгарский, каталанский, китайский, хорватский, чешский, датский, голландский, английский, эстонский, финский, французский, галисийский, немецкий, греческий, иврит, хинди, венгерский, испанский, индонезийский, итальянский, японский, каннада, казахский, корейский, латвийский, литовский, македонский, малайский, маратхи, маори, непальский, норвежский, персидский, польский, португальский, румынский, русский, сербский, словацкий, словенский, испанский, суахили, шведский, тагальский, тамильский, тайский, турецкий, украинский, урду, вьетнамский и валлийский.

Модель и код вывода были открыты OpenAI, и их можно найти на GitHub.

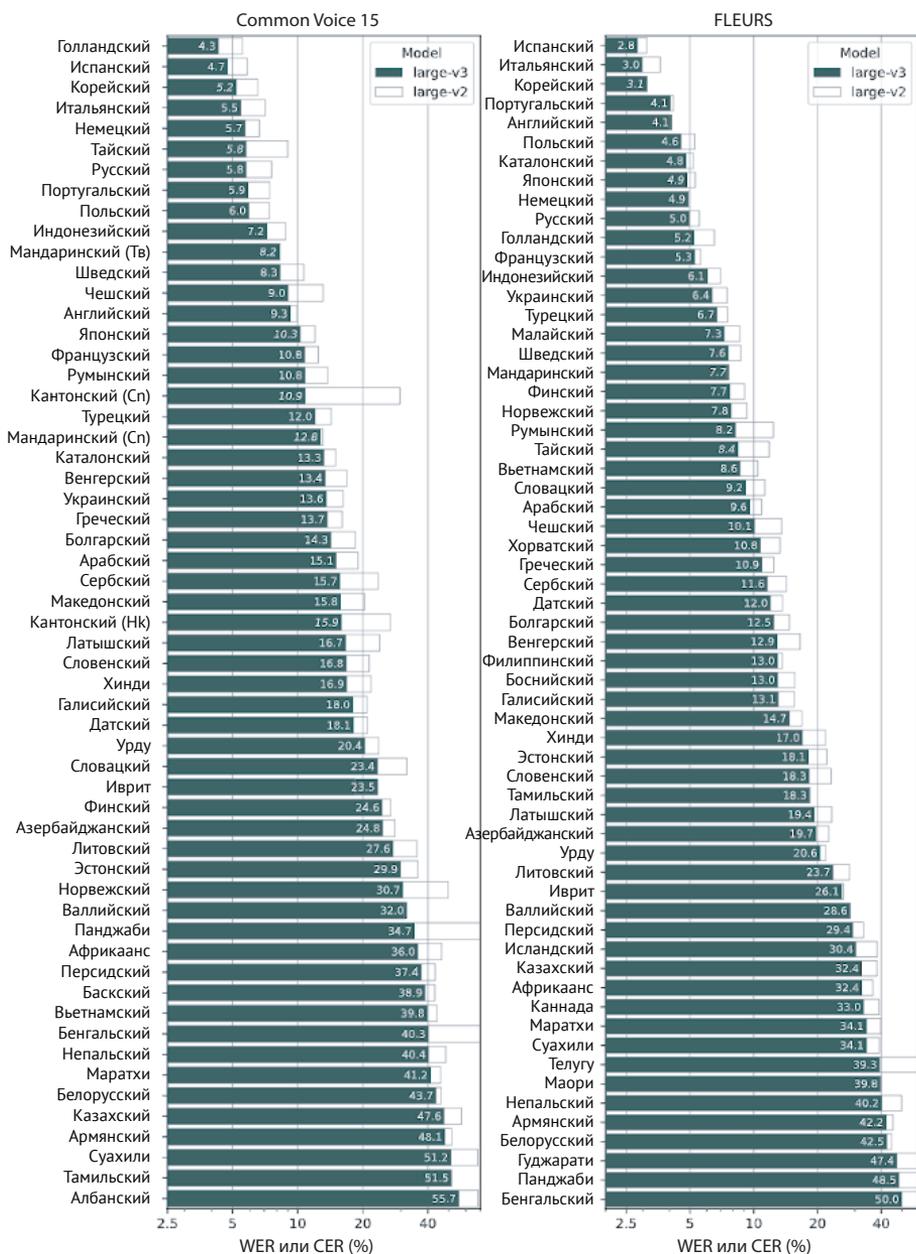
Существует пять размеров моделей, четыре из которых имеют версии только на английском языке. Каждая модель предлагает компромисс между скоростью и точностью. Названия моделей, а также их приблизительные требования к памяти и относительное быстродействие перечислены в таблице ниже.

Размер	Количество параметров, млн	Англоязычная модель	Многоязычная модель	Занимаемый объем ОЗУ, Гб	Относительное быстродействие
Легкая	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	1x

Качество работы Whisper существенно зависит от языка, размера модели, качества аудиосигнала и версии модели.

На рисунке ниже показана сравнительная диаграмма производительности больших моделей второй и третьей версий для различных языков. Диаграмма построена

на основе данных Word Error Rate (WER) и Character Error Rate (CER, курсив).
Оценки получены с использованием наборов данных Common Voice 15¹ и Fleurs².



Сравнительная диаграмма производительности больших моделей второй и третьей версий для различных языков

¹ <https://commonvoice.mozilla.org/en/datasets>.

² <https://huggingface.co/datasets/google/fleurs>.

С ЧЕГО НАЧАТЬ?

У вас должен быть установлен Python версии не ниже 3.8. Активируйте свою среду разработки:

```
mkvirtualenv -p python3.9 whisper-learning
```

Затем установите Whisper при помощи следующей команды:

```
pip install -U openai-whisper==20231117
```

Также необходимо установить на ваш компьютер кодек ffmpeg. Под ОС Ubuntu и Debian это делается так:

```
apt update
apt install ffmpeg
```

Под Arch Linux:

```
pacman -S ffmpeg
```

Под MacOS с помощью Homebrew:

```
brew install ffmpeg
```

Под Windows с помощью Chocolatey:

```
choco install ffmpeg
```

Под Windows с помощью Scoop:

```
scoop install ffmpeg
```

Установите setuptools-rust:

```
pip install setuptools-rust==1.8.1
```

Теперь скачайте аудиофайл, содержащий запись речи для тестирования. Множество файлов можно найти на Wikipedia¹.

```
# Создаем папку
mkdir -p src/whisper/audio
# Скачиваем аудиофайл
wget \
https://upload.wikimedia.org/\
wikipedia/commons/7/75/\
Winston_Churchill_-_Be_Ye_Men_of_Valour.ogg \
-O src/whisper/audio/Winston_Church.ogg
```

Далее введите команду запуска Whisper для базовой модели. Модель выполнит перевод речи в текст.

```
whisper \
src/whisper/audio/Winston_Church.ogg \
--model base
```

Вы можете выбрать другую модель. Чем лучше модель, тем больше она занимает места на диске и тем сильнее нагружает процессор. Например:

¹ https://commons.wikimedia.org/wiki/Category:Audio_files_of_speeches.

```
# Использование средней модели
whisper \
src/whisper/audio/Winston_Church.ogg \
--model medium
```

```
# Использование большой модели
whisper \
src/whisper/audio/Winston_Church.ogg \
--model large
```

В любом случае вы должны получить текст речи :

```
1 [00:00.000 --> 00:07.680] I speak to you for the first time as Prime Minister [...]
2 [00:08.320 --> 00:14.880] of our empire, of our allies, and above all of the [...]
3 [00:16.640 --> 00:20.240] A tremendous battle is raging in France and flanders [...]
4 [00:21.920 --> 00:27.920] The Germans by a remarkable combination of air bombing [...]
5 .]
6 [.....]
7 [.....]
8 [.....]
9 [03:16.400 --> 03:22.000] of the specialized and mechanized forces of the enemy [...]
10 [03:22.640 --> 03:26.560] and we know that very heavy losses have been inflicted [...]
11 .]
12 [.....]
13 [.....]
```

РАСПОЗНАВАНИЕ И ПЕРЕВОД РЕЧИ

Модель также может переводить речь на английский язык. Аудиопоток распознается в текст, а затем этот текст переводится на английский язык.

Для примера возьмем запись речи на русском языке:

```
wget \
https://upload.wikimedia.org/\
wikipedia/commons/1/1a/\
Lenin_-_What_Is_Soviet_Power.ogg \
-O src/whisper/audio/Lenin.ogg
```

Запустим распознавание и перевод:

```
whisper \
src/whisper/audio/Lenin.ogg \
--language Russian \
--task translate
```

Вы должны получить 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.
```

Настройка некоторых параметров позволяет улучшить качество распознавания и перевода. Например, можно увеличить температуру и количество результатов, чтобы получить несколько кандидатов.

Вот пример настройки:

```
whisper \
src/whisper/audio/Lenin.ogg \
--language Russian \
--task translate \
--best_of 20
```

Присмотритесь к выводу и обратите внимание на разницу с предыдущим переводом.

```
[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 ..
```

По умолчанию температура равна 0. Количество кандидатов имеет значение только при температуре, отличной от 0, и по умолчанию равно 5.

Воспользуйтесь командой помощи, чтобы изучить назначение остальных параметров в командной строке:

```
whisper -h
```

Стоит отметить, что Whisper можно использовать из интерфейса командной строки (CLI), как мы делали выше, из Python SDK или с помощью REST API.

ИСПОЛЬЗОВАНИЕ WHISPER SDK В КОДЕ ПУТНОН

Обратимся к Whisper в коде Python:

```
cat << EOF > src/whisper/whisper_test.py
import whisper

# Выберите нужный размер модели
# (т. е. "tiny", "small", "medium", "large")
model_name = "base"

try:
    # Загрузка модели
    model = whisper.load_model(model_name)
except Exception as e:
    print(f"Ошибка загрузки модели '{model_name}': {e}")
    exit(1)

audio_file_path = "src/whisper/audio/Winston_Church.ogg"

try:
```

```

# Распознавание аудиозаписи
result = model.transcribe(audio_file_path)
print(f'Полученный текст:')
print(result['text'])
except Exception as e:
    print(f"Ошибка распознавания записи '{audio_file_path}': {e}")
EOF

```

Выполним скрипт:

```
python src/whisper/whisper_test.py
```

Вы должны увидеть текст речи.

Для перевода распознанной речи подойдет такой скрипт:

```

cat << EOF > src/whisper/whisper_translate.py
import whisper

# Выберите нужный размер модели
# (т. е. "tiny", "small", "medium", "large")
model_name = "medium"

try:
    # Загрузка модели
    model = whisper.load_model(model_name)
except Exception as e:
    print(f"Ошибка загрузки модели '{model_name}': {e}")
    exit(1)

audio_file_path = "src/whisper/audio/Winston_Church.ogg"

try:
    # Перевод аудиозаписи на арабский
    result = model.transcribe(
        audio_file_path,
        language="ar"
    )
    print("Арабский:")
    print(result)
    # Перевод аудиозаписи на китайский
    result = model.transcribe(
        audio_file_path,
        language="zh"
    )
    print("Китайский:")
    print(result)
    # ... и т.д.
except Exception as e:
    print(f"Ошибка распознавания записи '{audio_file_path}': {e}")
EOF

```

Выполните этот скрипт:

```
python src/whisper/whisper_translate.py
```

Вы должны увидеть текст речи, переведенной на несколько языков.

Использование API OpenAI для преобразования аудиозаписи в текст

OpenAI предоставляет REST API для Whisper. У вас есть доступ к двум конечным точкам:

- распознавание: вы можете распознать аудиофайл на различных языках и преобразовать его в текст;
- перевод: вы можете распознать аудиофайл и перевести текст на английский язык.

В настоящее время OpenAI позволяет загружать аудиофайлы размером не более 25 МБ, но в будущем это ограничение может измениться. Аудиофайл должен быть в одном из следующих форматов: mp3, mp4, mpeg, mpga, m4a, wav или webm.

API РАСПОЗНАВАНИЯ

Начните с активации виртуальной среды, в которую вы установили SDK OpenAI:

```
workon openaigtforpythondevelopers
```

Ранее мы уже создали файл `src/api.py`. Возьмем его за основу для аутентификации в API. Вот содержимое файла:

```
import os
from openai import OpenAI

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

client = OpenAI(
    api_key=os.environ['API_KEY'],
    organization=os.environ['ORG_ID']
)
```

Файл `.env` содержит идентификатор пользователя и ключ API:

```
API_KEY=sk-...
ORG_ID=org-...
```

Следующий код представляет собой пример использования API распознавания речи:

```

cat << EOF > src/transcribe.py
# Аутентификация
from api import client
import os

# Задаем путь к аудиофайлу
audio_file_path = os.path.abspath(
    "src/whisper/audio/Lenin.ogg"
)
# Задаем модель
model = "whisper-1"

# Открываем аудиофайл в двоичном режиме
with open(audio_file_path, 'rb') as audio_file:
    # Распознавание аудиозаписи
    transcript = client.audio.transcriptions.create(
        model=model,
        file=audio_file
    )

print(transcript.text)
EOF

```

Выполните скрипт:

```
python src/transcribe.py
```

Вы должны увидеть текст распознанной речи.

В связи с ограничением размера файла до 25 МБ вам может понадобиться разделить файл аудиозаписи на несколько частей и распознать их по отдельности. Рекомендуется не разделять файл на середине предложения, поскольку это может ухудшить качество распознавания.

API ПЕРЕВОДА

Использование API перевода практически не отличается от использования API распознавания. Вот пример того, как это делается (разница выделена жирным шрифтом):

```

cat << EOF > src/translate.py
# Аутентификация
from api import client
import os

# Задаем путь к аудиофайлу
audio_file_path = os.path.abspath(
    "src/whisper/audio/Lenin.ogg"
)
# Задаем модель
model = "whisper-1"

# Открываем аудиофайл в двоичном режиме
with open(audio_file_path, 'rb') as audio_file:

```

```
# Распознаем аудиозапись
transcript = client.audio.translations.create(
    model=model,
    file=audio_file
)

print(transcript.text)
EOF
```

Выполните скрипт:

```
python src/translate.py
```

Вы должны увидеть текст речи, переведенный с русского языка на английский.

УЛУЧШЕНИЕ КАЧЕСТВА РАСПОЗНАВАНИЯ РЕЧИ С WHISPER

Одна из проблем использования Whisper заключается в том, что модель не всегда правильно распознает речь.

Это особенно заметно при невысоком качестве аудиозаписи. Затруднения также доставляют редко употребляемые слова, технические термины и жаргонизмы.

Далее мы рассмотрим несколько подходов к улучшению качества распознавания.

Очистка аудиозаписи

Первое, что вы можете сделать для повышения точности распознавания, – это очистить звук. Очистка подразумевает удаление любого фонового шума, эха и других звуков, которые могут мешать речи.

Вы можете использовать для очистки звука такой инструмент, как Audacity. Это бесплатное кросс-платформенное программное обеспечение с открытым исходным кодом, которое позволяет записывать и редактировать аудиофайлы. Альтернативой могут служить noisereduce¹, librosa², PyDub³, audiomentations⁴ и многие другие.

Использование подсказки

Полезной функцией, которая может повысить точность транскрипции, является возможность предоставления подсказки (промпта). Чтобы повысить точность транскрипции необычных слов или сокращений с помощью Whisper, вы можете использовать параметр `prompt` для передачи в модель словаря с правильным написанием специализированных терминов. Этот подход помогает Whisper распознавать и точно расшифровывать сложные или необычные слова, используя контекст либо подсказку.

¹ <https://github.com/timsainb/noisereduce>.

² <https://github.com/librosa/librosa>.

³ <https://github.com/jiaaro/pydub>.

⁴ <https://github.com/iver56/audiomentations>.

Whisper обрабатывает только первые 244 токена промпта. Список конкретных терминов или сокращений (например, артикулов продуктов, технических терминов) должен быть относительно коротким, чтобы соответствовать этому ограничению.

Давайте посмотрим пример использования параметра `prompt`. Загрузите звуковой отрывок из фильма «Матрица»:

```
# Создаем папку
mkdir -p src/whisper/audio

# Скачивание аудиофайла
wget \
https://github.com/\
eon01/Openai_GPT_for_Python_Developers_Files/\
raw/main/audio/cypher.mp3 \
-O src/whisper/audio/cypher.mp3
```

Содержимое аудиофайла следующее:

```
We need an exit. You're not far from Cypher. Cypher?
```

Хотя в правильной транскрипции должно быть написание «Cypher», Whisper может распознать его как «CIPHER»¹.

```
We need an exit. You're not far from Cipher. Cipher?
```

Чтобы исправить эту оплошность, вы можете воспользоваться параметром `prompt` и дать модели подсказку о предпочтительном написании.

Следующий скрипт демонстрирует пример такой подсказки:

```
cat << EOF > src/transcribe_with_prompt.py
# Аутентификация
from api import client
import os

# Задаем путь к аудиофайлу
audio_file_path = os.path.abspath(
    "src/whisper/audio/cypher.mp3"
)
# Задаем модель
model = "whisper-1"

# Задаем подсказку
prompt = "Cypher"
# Открываем аудиофайл в двоичном режиме
with open(audio_file_path, 'rb') as audio_file:
    # Распознаем аудиозапись
    transcript = client.audio.transcriptions.create(
        model=model,
```

¹ Возможно, это не самый удачный пример. «Cypher» – это устаревшее британское написание слова «шифр», а «cipher» – изначально американское, а теперь общепринятое написание, рекомендованное ассоциацией криптографов и применяемое в современной литературе по криптографии. Но правильными считаются оба варианта написания. – *Прим. перев.*

```

        file=audio_file,
        prompt=prompt
    )

print(transcript.text)
EOF

```

Этот скрипт вернет нужную транскрипцию:

```
We need an exit. You're not far from Cypher. Cypher?
```

Если удалить подсказку, транскрипция будет иной:

```
We need an exit. You're not far from Cipher. Cipher?
```

Важно помнить, что модель Whisper, как и базовые модели GPT, не обучена выполнять инструкции. Вы не можете использовать промпт, чтобы попросить ее сделать что-то особенное, например «перевести этот аудиофайл в текст и объяснить его содержание пятилетнему ребенку». Вы можете только дать подсказки по транскрипции.

Постобработка полученного текста

Еще один подход к улучшению качества распознанного текста заключается в применении дополнительной модели завершения. Иными словами, вы можете воспользоваться моделями GPT-3 или GPT-4 для корректировки транскрипции.

Следующий код служит примером корректировки транскрипции с помощью GPT-3:

```

cat << EOF > src/post_process.py
# Аутентификация
from api import client
import os

# Задаем путь к аудиофайлу
audio_file_path = os.path.abspath(
    "src/whisper/audio/cypher.mp3"
)
# Задаем модель
model = "whisper-1"

# Открываем аудиофайл в двоичном режиме
with open(audio_file_path, 'rb') as audio_file:
    # Распознаем аудиозапись
    transcript = client.audio.transcriptions.create(
        model=model,
        file=audio_file
    )

# Постобработка текста с помощью GPT-3
system_prompt = """Ты эксперт по кинофильмам.
Твоя задача - исправить транскрипцию аудиозаписи.
Например, слово "Cipher" надо заменить на "Cypher"."""

```

```
# Задаем модель
model = "gpt-3.5-turbo"

# Постобработка текста
response = client.chat.completions.create(
    model=model,
    messages=[
        {
            "role": "system",
            "content": system_prompt
        },
        {
            "role": "user",
            "content": transcript.text
        }
    ]
)

print(response.choices[0].message.content)
EOF
```

Запустите скрипт:

```
python src/post_process.py
```

Этот код предназначен для аудиофайла и контекста из предыдущих примеров данной главы. Если вы решите использовать другой аудиофайл, вам придется переделать промпт системы и сообщение пользователя.

Преобразование текста в речь

Модели *преобразования текста в речь* (text-to-speech, TTS) OpenAI, а именно TTS-1 и TTS-1-HD, предназначены для преобразования письменного текста в естественно звучащую звуковую разговорную речь (озвучивания текста).

Модель TTS-1 специально оптимизирована для приложений преобразования текста в речь в режиме реального времени и обеспечивает баланс между качеством и скоростью. Это делает ее идеальной для интерактивных сценариев, где решающее значение имеет немедленная голосовая обратная связь.

В свою очередь, вариант TTS-1-HD ориентирован на обеспечение более высокого качества звука. Он подходит для случаев, когда четкость, тональная насыщенность и общая точность озвучивания имеют первостепенное значение. Этот вариант модели особенно полезен для создания контента, аудиокниг и любых приложений, где качество синтеза речи существенно влияет на пользовательский опыт.

Обе модели используют передовую архитектуру нейронных сетей OpenAI. Они обучаются на разнообразном наборе данных аудиозаписей, что позволяет им генерировать естественно звучащий звук, очень напоминающий живую человеческую речь (нюансы, интонация, эмоции и т. д.).

Давайте начнем с создания текстового файла, который содержит текст для преобразования в речь. Затем воспользуемся API OpenAI для озвучивания текста.

Создайте новый файл с именем `speech.txt` в каталоге `src/data/tts` и напишите следующий текст:

```
# Создаем папку
mkdir -p src/data/tts
# Создаем текстовый файл
cat << EOF > src/data/tts/speech.txt
Hello, Neo - Welcome to the Matrix.
EOF
```

В следующем коде показан пример конвертации текста в речь с помощью API OpenAI.

```
cat << EOF > src/tts.py
# Аутентификация
from api import client
import os

# Задаем путь к текстовому файлу
text_file_path = os.path.abspath(
    "src/data/tts/speech.txt"
)
# Задаем путь к аудиофайлу
audio_file_path = os.path.abspath(
    "src/data/tts/speech.mp3"
)
```

```

# Читаем текст из файла
with open(text_file_path, 'r') as file:
    text = file.read()

# Задаем голосовую модель и персонажа
voice_model = "tts-1"
voice_character = "alloy"

# Конвертируем текст в речь
response = client.audio.speech.create(
    model=voice_model,
    voice=voice_character,
    input=text
)

# Сохраняем речь в аудиофайл
response.stream_to_file(audio_file_path)
print(f" Аудиофайл сохранен как {audio_file_path}")
EOF

```

Для тестирования модели TTS-1-HD достаточно изменить значение переменной `voice_model` на `tts-1-hd`.

Модели TTS поддерживают следующие тональности голоса («персонажи»): `alloy`, `echo`, `fable`, `onyx`, `nova` и `shimmer`.

Чтобы услышать различия в звучании этих персонажей, попробуйте менять значение переменной `voice_character`, как показано в следующем коде.

```

cat << EOF > src/tts_all.py
# Аутентификация
from api import client
import os

# Задаем путь к текстовому файлу
text_file_path = os.path.abspath(
    "src/data/tts/speech.txt"
)

# Читаем текст из файла
with open(text_file_path, 'r') as file:
    text = file.read()

# Задаем голосовую модель и персонажа
voice_model = "tts-1"
voice_characters = [
    "alloy",
    "echo",
    "fable",
    "onyx",
    "nova",
    "shimmer"
]

```

```
for voice_character in voice_characters:
    # Задаем путь к аудиофайлу
    audio_file_path = os.path.abspath(
        "src/data/tts/speech.mp3"
    )
    # Конвертируем текст в речь
    response = client.audio.speech.create(
        model=voice_model,
        voice=voice_character,
        input=text
    )

    # Сохраняем речь в аудиофайл
    response.stream_to_file(audio_file_path)
    print(f"Аудиофайл сохранен как {audio_file_path}")
EOF
```

Модели TTS поддерживают следующие языки: африкаанс, арабский, армянский, азербайджанский, белорусский, боснийский, болгарский, каталанский, китайский, хорватский, чешский, датский, голландский, английский, эстонский, финский, французский, галисийский, немецкий, греческий, иврит, хинди, венгерский, исландский, индонезийский, итальянский, японский, каннада, казахский, корейский, латвийский, литовский, македонский, малайский, маратхи, маори, непальский, норвежский, персидский, польский, португальский, румынский, русский, сербский, словацкий, словенский, испанский, суахили, шведский, тагальский, тамильский, тайский, турецкий, украинский, урду, вьетнамский и валлийский.

Согласно лицензионному соглашению OpenAI, вы владеете правами на сгенерированный звуковой файл и можете использовать его в любых целях. Однако вы должны сообщить конечным пользователям, что звук создан с помощью ИИ, а не реальным человеком.

Диалог между двумя ИИ на основе OpenAI и Weaviate

ГЕНЕРАЦИЯ АУДИОФАЙЛОВ

В этой главе мы вернемся к темам предыдущих глав:

- модели OpenAI GPT;
- модели OpenAI TTS;
- интеграция контекста и знаний в векторной базе данных с применением API OpenAI.

Наша цель – создание автономного диалога между двумя ИИ с помощью OpenAI и Weaviate. Блоки кода из предыдущих примеров в основном останутся неизменными, за исключением API (файл Flask `app_dialog.py`).

Файловая структура проекта должна выглядеть так:

```
.
├── app
│   ├── app_dialog.py
│   ├── Dockerfile.app
│   ├── .env
│   ├── requirements.txt
│   └── static
├── docker-compose.yaml
└── weaviate
    ├── Dockerfile.weaviate
    └── .env
```

Файл `.env` для Weaviate должен содержать модули `text2vec-openai` и `generative-openai`:

```
OPENAI_APIKEY=$OPENAI_APIKEY
QUERY_DEFAULTS_LIMIT=25
AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED=true
PERSISTENCE_DATA_PATH=/var/lib/weaviate
DEFAULT_VECTORIZER_MODULE=text2vec-openai
ENABLE_MODULES=text2vec-openai,generative-openai
CLUSTER_HOSTNAME=node1
```

В файле `app_dialog.py` мы начинаем с определения переменных для хранения ключей API и других настроек:

```
import weaviate, os
from flask import Flask, session, request
from openai import OpenAI

app = Flask(__name__)
```

```

# Задаем секретный ключ в виде набора случайных байтов.
# Тщательно храните его в тайне!
app.secret_key = 'СМЕНИТЕ_ЭТОТ_СЕКРЕТНЫЙ_КЛЮЧ'
# Ключ OpenAI API
openai_api_key = os.getenv("ВАШ_КЛЮЧ_OPENAI_API")
# Модель OpenAI для получения встраиваний
embedding_model = "text-embedding-3-small"
# Имя класса в Weaviate
weaviate_class_name = "Chat"
# Счетчик для отслеживания длины диалога
counter = 0
# Длина диалога
conversation_length = 3
# Определение главной темы для диалога
topic = "Joe Biden" //"Джо Байден"
# Начальное сообщение ассистенту от пользователя
initial_prompt = f"Давай поговорим на эту тему: {topic}."
# Голосовой персонаж для пользователя и помощника
user_voice = "alloy"
assistant_voice = "echo"

openai_client = OpenAI(
    api_key=openai_api_key
)
weaviate_client = weaviate.Client(
    url="http://weaviate:8080",
    auth_client_secret={
        "X-OpenAI-API-Key": openai_api_key
    }
)

```

Поскольку разговор между двумя экземплярами моделей OpenAI должен быть инициирован пользователем, мы определили первый промпт (включая основную тему) как первоначальное сообщение от пользователя помощнику. Переменные `user_voice` и `assistant_voice` используются для определения голосового персонажа TTS пользователя и помощника.

Разговор должен закончиться после определенного количества ходов. Мы определили переменную `conversation_length` для управления длиной разговора. Переменная `counter` нужна для отслеживания продолжительности разговора.

Далее мы определим схему класса Weaviate:

```

schema = {
    "classes": [
        {
            "class": weaviate_class_name,
            "description": "Chat messages",
            "vectorizer": "text2vec-openai",
            "moduleConfig": {
                # Настройка того, как OpenAI создает встраивания
                # для поля `content` (`text`)
                "text2vec-openai": {
                    "model": embedding_model,

```

```

        "type": "text",
        "vectorizeClassName": False,
        "vectorizePropertyName": False
    },
    "generative-openai": {
        "model": "gpt-4",
        "temperatureProperty": 1.2,
        "maxTokensProperty": 100,
        "frequencyPenaltyProperty": 2.0,
        "presencePenaltyProperty": 2.0,
    }
},
"properties": [
    {
        "name": "role",
        "description": "The role",
        "dataType": ["string"],
        # Не векторизовать роль
        "moduleConfig": {
            "text2vec-openai": {
                "skip": True
            }
        }
    },
    {
        "name": "content",
        "description": "The content",
        "dataType": ["text"],
        "moduleConfig": {
            "text2vec-openai": {
                "skip": False,
            }
        }
    },
    {
        "name": "index",
        "dataType": ["int"],
        "description": "The index",
        "moduleConfig": {
            "text2vec-openai": {
                "skip": True
            }
        }
    }
],
]
}

```

В приведенной выше схеме мы определили модули `text2vec-openai` и `generative-openai`. Модуль `text2vec-openai` нужен для векторизации поля `content`, а модуль `generative-openai` – для генерации ответа. Мы будем хранить роль со-

общения («пользователь» или «помощник») в поле `role`, содержимое сообщения в поле `content` и индекс сообщения в поле `index` (счетчик).

OpenAI GPT отвечает за генерацию ответов как пользователя, так и помощника.

Конечный пользователь не увидит, что сообщения исходят от «пользователя» или «помощника», поскольку мы определим два аватара, полностью скрывающих пользователя и помощника. Мы также определим две разные личности для пользователя и помощника. Все это делается при помощи промпта.

Когда помощник отвечает на первоначальный промпт, мы включаем и ответ, и контекст (сгенерированный Weaviate) в промпт, отправляемый пользователю. Аудиофайл того же сообщения сохраняется в папке `static`. Сообщение также сохраняется в Weaviate. Пользователь получает промпт, который содержит сообщение помощника и контекст от Weaviate. Затем пользователь отвечает сообщением. Этот процесс повторяется до тех пор, пока разговор не закончится.

Вот код основной процедуры:

```
@app.route("/chat", methods=["GET"])
def assistant_to_user():
    global counter
    role = request.args.get("role")
    user_session_key = "user_answer"
    assistant_session_key = "assistant_answer"

    if role == "user":
        # Получаем предыдущее сообщение от 2-го персонажа
        previous_message = session.get(
            assistant_session_key,
            initial_prompt
        )
        # Определяем тип поведения 1-го персонажа
        personality = """Ты Макс, веселый юморист, мастер
        беззаботных каламбуров и незатейливых шуток с остроумной
        изюминкой. Макс - веселый друг, который находит радость
        в любой ситуации и делится ею с помощью доброго юмора.
        Твоя цель - поднять настроение и развлечь, заставив всех
        почувствовать себя желанными гостями и рассмеяться
        добрым смехом.
        """
        # Определяем голос 1-го персонажа
        voice = user_voice
        # Определяем ключ сеанса 1-го персонажа
        my_session_key = user_session_key
        # Определяем роль 1-го персонажа
        interlocutor_role = "assistant"

    elif role == "assistant":
        # Получаем предыдущее сообщение от 1-го персонажа
        previous_message = session.get(
            user_session_key,
            initial_prompt
        )
```

```

# Определяем тип поведения 2-го персонажа
personality = """Ты Зоя. У тебя необычное и острое чувство юмора, сочетающее остроумные шутки с долей сарказма. Ты любишь игру слов и абсурд и часто удивляешь окружающих неожиданными умными выводами. Твоя цель - развлечь и вовлечь в диалог, заставив всех почувствовать себя вовлеченными в игру с неожиданными умными шутками.
""

# Определяем голос 2-го персонажа
voice = assistant_voice
# Определяем ключ сеанса 2-го персонажа
my_session_key = assistant_session_key
# Определяем роль 2-го персонажа
interlocutor_role = "user"
else:
# Возвращаем ошибку, если роль задана неправильно
# Разрешены только роли "user" или "assistant"
return {
    "error": "Недопустимая роль"
}, 400

```

```

# Определяем общий промпт для двух персонажей
base_prompt = """
Ты находишься в середине разговора.
Не надо говорить "Привет" или "Здравствуйте".
Просто задай вопрос или расскажи что-то интересное.
Не будь скучным. Максимально придерживайся темы.
Тема может меняться, но не слишком резко.
Когда она меняется, это должен быть плавный переход,
всегда связанный с основной темой:""" + topic \
+ """

```

Следующий текст описывает текущий контекст:

```
{content}
```

Это последнее сообщение от пользователя:

```
"" + previous_message + ""
```

```

Если счетчик превышает"" + \
str(conversation_length) + "",
ты и пользователь должны оба завершить разговор.
Текущее значение счетчика: "" + str(counter)

```

```

# Добавляем персонажей в промпт
prompt = personality + "\n" + base_prompt

```

```

# Сохраняем предыдущие сообщения в Weaviate
# перед генерацией ответа
if counter == 0:

```

```

# Начальные данные, включая начальный промпт
interlocutor_data = {
    "role": "user",
    "content": initial_prompt,
    "index": counter,
}
else:
# Последующие данные
interlocutor_data = {
    "role": interlocutor_role,
    "content": previous_message,
    "index": counter
}

# Функция сохранения в Weaviate
weaviate_save_data(
    interlocutor_data["role"],
    interlocutor_data["content"],
    interlocutor_data["index"]
)
counter += 1

try:
    full_result = generate_response(
        prompt,
        previous_message
    )
    result_text = full_result['data']['Get']\
        [weaviate_class_name][0]\
        ['_additional']['generate']\
        ['groupedResult']
    response_code = 200
except Exception as e:
    app.logger.error(
        f"Ошибка при получении данных: {e}"
    )
    full_result = "Ошибка"
    result_text = "Не могли бы вы повторить?"
    response_code = 500

# Сохраняем result_text текущего сеанса
session[my_session_key] = result_text

# Переводим текст в речь при помощи модели TTS
destination = os.path.join(
    "static/",
    str(counter) + ".mp3"
)
text_to_speech(
    result_text,
    voice,
    destination
)

```

```

return {
    "result_text": result_text,
    "result_audio": destination,
    "full_result": full_result,
    "interlocutor_data": interlocutor_data,

}, response_code

```

Далее нам нужно определить функции, задействованные в приведенном выше коде, а именно `weaviate_save_data`, `generate_response` и `text_to_speech`.

Вот они:

```

def weaviate_save_data(
    role,
    content,
    index
):
    properties = {
        "role": role,
        "content": content,
        "index": index
    }
    weaviate_client.batch.configure(
        batch_size=10
    )
    with weaviate_client.batch as batch:
        batch.add_data_object(
            properties,
            class_name=weaviate_class_name,
        )

def generate_response(
    prompt,
    previous_message
):
    nearText = {
        "concepts": [previous_message],
    }

    properties = [
        "role",
        "content",
        "index",
        "_additional {distance}"
    ]

    limit = 5

    generative = weaviate_client.query.get(
        class_name=weaviate_class_name,
        properties=properties,
    ).with_generate(
        grouped_task=prompt

```

```
        ).with_near_text(
            nearText
        ).with_limit(
            limit
        ).do()

    return generative

def text_to_speech(
    text,
    voice,
    destination,
    model="tts-1"
):
    response = openai_client.audio.speech.create(
        model=model,
        voice=voice,
        input=text
    )

    response.stream_to_file(destination)
```

Затем определим функцию-декоратор, которая создает схему и удаляет все старые сеансы и данные:

```
@app.route("/init", methods=["GET"])
def init():
    global counter
    # clear the session
    session.clear()
    weaviate_delete_data()
    weaviate_create_schema()

    destination = os.path.join(
        "static/"
        + str(counter) + ".mp3"
    )
    text_to_speech(
        initial_prompt,
        user_voice,
        destination
    )
    return {
        "status": "initialized"
    }, 200
```

Эта функция вызывает две другие функции, `weaviate_delete_data` и `weaviate_create_schema`:

```
def weaviate_create_schema():
    try:
        # Создаем схему
        weaviate_client.schema.create(schema)
```

```

        app.logger.debug(
            "Схема успешно создана."
        )
    except Exception as e:
        app.logger.error(
            f"Не удалось создать схему: {e}"
        )

def weaviate_delete_data():
    try:
        # Удаление данных
        weaviate_client.schema.delete_class(
            class_name=weaviate_class_name
        )
        app.logger.debug("Data successfully reset.")
    except Exception as e:
        app.logger.error(
            f"Ошибка при удалении класса: {e}"
        )
    return {
        "Ошибка в weaviate_reset": str(e)
    }, 500

```

Собрав все части воедино, мы получим следующий код:

```

cat << 'EOT' > src/vector_db_generative_search/app/app_dialog_full.py
import weaviate, os
from flask import Flask, session, request
from openai import OpenAI

app = Flask(__name__)
# Задаем секретный ключ в виде набора случайных байтов.
# Тщательно храните его в тайне!
app.secret_key = 'СМЕНИТЕ_ЭТОТ_СЕКРЕТНЫЙ_КЛЮЧ'
# Ключ OpenAI API
openai_api_key = os.getenv("ВАШ_КЛЮЧ_OPENAI_API")
# Модель OpenAI для получения встраиваний
embedding_model = "text-embedding-3-small"
# Имя класса в Weaviate
weaviate_class_name = "Chat"
# Счетчик для отслеживания длины диалога
counter = 0
# Длина диалога
conversation_length = 3
# Определение главной темы для диалога
topic = "Joe Biden" //"Джо Байден"
# Начальное сообщение ассистенту от пользователя
initial_prompt = f"Давай поговорим на эту тему: {topic}."
# Голосовой персонаж для пользователя и помощника
user_voice = "alloy"
assistant_voice = "echo"

openai_client = OpenAI(
    api_key=openai_api_key
)

```

```

weaviate_client = weaviate.Client(
    url="http://weaviate:8080",
    auth_client_secret={
        "X-OpenAI-API-Key": openai_api_key
    }
)

schema = {
    "classes": [
        {
            "class": weaviate_class_name,
            "description": "Chat messages",
            "vectorizer": "text2vec-openai",
            "moduleConfig": {
                # Настройка того, как OpenAI создает встраивания
                # для поля `content` (`text`)
                "text2vec-openai": {
                    "model": embedding_model,
                    "type": "text",
                    "vectorizeClassName": False,
                    "VectorizePropertyName": False
                },
                "generative-openai": {
                    "model": "gpt-4",
                    "temperatureProperty": 1.2,
                    "maxTokensProperty": 100,
                    "frequencyPenaltyProperty": 2.0,
                    "presencePenaltyProperty": 2.0,
                }
            },
        },
        {
            "name": "role",
            "description": "The role",
            "dataType": ["string"],
            # Не векторизовать роль
            "moduleConfig": {
                "text2vec-openai": {
                    "skip": True
                }
            }
        },
        {
            "name": "content",
            "description": "The content",
            "dataType": ["text"],
            "moduleConfig": {
                "text2vec-openai": {
                    "skip": False,
                }
            }
        },
    ],
    "properties": [
        {
            "name": "role",
            "description": "The role",
            "dataType": ["string"],
            # Не векторизовать роль
            "moduleConfig": {
                "text2vec-openai": {
                    "skip": True
                }
            }
        },
        {
            "name": "content",
            "description": "The content",
            "dataType": ["text"],
            "moduleConfig": {
                "text2vec-openai": {
                    "skip": False,
                }
            }
        },
    ],
}

```

```

        {
            "name": "index",
            "dataType": ["int"],
            "description": "The index",
            "moduleConfig": {
                "text2vec-openai": {
                    "skip": True
                }
            }
        },
    ],
}

def weaviate_create_schema():
    try:
        # Создаем схему
        weaviate_client.schema.create(schema)
        app.logger.debug(
            "Схема успешно создана."
        )
    except Exception as e:
        app.logger.error(
            f"Не удалось создать схему: {e}"
        )

def weaviate_delete_data():
    try:
        # Удаление данных
        weaviate_client.schema.delete_class(
            class_name=weaviate_class_name
        )
        app.logger.debug("Data successfully reset.")
    except Exception as e:
        app.logger.error(
            f"Ошибка при удалении класса: {e}"
        )
    return {
        "Ошибка в weaviate_reset": str(e)
    }, 500

def weaviate_save_data(
    role,
    content,
    index
):
    properties = {
        "role": role,
        "content": content,
        "index": index
    }

```

```
weaviate_client.batch.configure(
    batch_size=10
)
with weaviate_client.batch as batch:
    batch.add_data_object(
        properties,
        class_name=weaviate_class_name,
    )

def generate_response(
    prompt,
    previous_message
):
    nearText = {
        "concepts": [previous_message],
    }

    properties = [
        "role",
        "content",
        "index",
        "_additional {distance}"
    ]

    limit = 5

    generative = weaviate_client.query.get(
        class_name=weaviate_class_name,
        properties=properties,
    ).with_generate(
        grouped_task=prompt
    ).with_near_text(
        nearText
    ).with_limit(
        limit
    ).do()

    return generative

def text_to_speech(
    text,
    voice,
    destination,
    model="tts-1"
):
    response = openai_client.audio.speech.create(
        model=model,
        voice=voice,
        input=text
    )

    response.stream_to_file(destination)
```

```

@app.route("/init", methods=["GET"])
def init():
    global counter
    # Очищаем сеанс
    session.clear()
    weaviate_delete_data()
    weaviate_create_schema()

    destination = os.path.join(
        "static/"
        + str(counter) + ".mp3"
    )
    text_to_speech(
        initial_prompt,
        user_voice,
        destination
    )
    return {
        "status": "initialized"
    }, 200

@app.route("/chat", methods=["GET"])
def assistant_to_user():
    global counter
    role = request.args.get("role")
    user_session_key = "user_answer"
    assistant_session_key = "assistant_answer"

    if role == "user":
        # Получаем предыдущее сообщение от 2-го персонажа
        previous_message = session.get(
            assistant_session_key,
            initial_prompt
        )
        # Определяем тип поведения 1-го персонажа
        personality = """Ты Макс, веселый юморист, мастер
        беззаботных каламбуров и незатейливых шуток с остроумной
        изюминкой. Макс - веселый друг, который находит радость
        в любой ситуации и делится ею с помощью доброго юмора.
        Твоя цель - поднять настроение и развлечь, заставив всех
        почувствовать себя желанными гостями и рассмеяться
        добрым смехом.
        """

        # Определяем голос 1-го персонажа
        voice = user_voice
        # Определяем ключ сеанса 1-го персонажа
        my_session_key = user_session_key
        # Определяем роль 1-го персонажа
        interlocutor_role = "assistant"

```

```

elif role == "assistant":
    # Получаем предыдущее сообщение от 1-го персонажа
    previous_message = session.get(
        user_session_key,
        initial_prompt
    )
    # Определяем тип поведения 2-го персонажа
    personality = """"Ты Зоя. У тебя необычное и острое чувство юмора, сочетающее остроумные шутки с долей сарказма. Ты любишь игру слов и абсурд и часто удивляешь окружающих неожиданными умными выводами. Твоя цель - развлечь и вовлечь в диалог, заставив всех почувствовать себя вовлеченными в игру с неожиданными умными шутками.
    """"

    # Определяем голос 2-го персонажа
    voice = assistant_voice
    # Определяем ключ сеанса 2-го персонажа
    my_session_key = assistant_session_key
    # Определяем роль 2-го персонажа
    interlocutor_role = "user"
else:
    # Возвращаем ошибку, если роль задана неправильно
    # Разрешены только роли "user" или "assistant"
    return {
        "error": "Недопустимая роль"
    }, 400

# Определяем общий промпт для двух персонажей
base_prompt = """"
Ты находишься в середине разговора.
Не надо говорить "Привет" или "Здравствуйте".
Просто задай вопрос или расскажи что-то интересное.
Не будь скучным. Максимально придерживайся темы.
Тема может меняться, но не слишком резко.
Когда она меняется, это должен быть плавный переход,
всегда связанный с основной темой:""" + topic \
+ """"

Следующий текст описывает текущий контекст:

{content}

Это последнее сообщение от пользователя:

"""" + previous_message + """"

Если счетчик превышает"""" + \
str(conversation_length) + """".
ты и пользователь должны оба завершить разговор.
Текущее значение счетчика: """" + str(counter)

```

```

# Добавляем персонажей в промпт
prompt = personality + "\n" + base_prompt

# Сохраняем предыдущие сообщения в Weaviate
# перед генерацией ответа
if counter == 0:
    # Начальные данные
    interlocutor_data = {
        "role": "user",
        "content": initial_prompt,
        "index": counter,
    }
else:
    # Последующие данные
    interlocutor_data = {
        "role": interlocutor_role,
        "content": previous_message,
        "index": counter
    }

# Функция сохранения в Weaviate
weaviate_save_data(
    interlocutor_data["role"],
    interlocutor_data["content"],
    interlocutor_data["index"]
)
counter += 1

try:
    full_result = generate_response(
        prompt,
        previous_message
    )
    result_text = full_result['data']['Get']\
        [weaviate_class_name][0]\
        ['_additional']['generate']\
        ['groupedResult']
    response_code = 200
except Exception as e:
    app.logger.error(
        f"Ошибка при получении данных: {e}"
    )
    full_result = "Ошибка"
    result_text = "Не могли бы вы повторить?"
    response_code = 500

# Сохраняем result_text текущего сеанса
session[my_session_key] = result_text

# Переводим текст в речь при помощи модели TTS
destination = os.path.join(

```

```

        "static/",
        str(counter) + ".mp3"
    )
    text_to_speech(
        result_text,
        voice,
        destination
    )

    return {
        "result_text": result_text,
        "result_audio": destination,
        "full_result": full_result,
        "interlocutor_data": interlocutor_data,
    }, response_code

@app.route("/data", methods=["GET"])
def weaviate_get_all_data():
    try:
        result = weaviate_client.query.get(
            class_name=weaviate_class_name,
            properties=[
                "role",
                "content",
                "index"
            ]
        ).do()

        return {
            "data": result['data']['Get'][weaviate_class_name]
        }

    except Exception as e:
        app.logger.error(
            f"Ошибка при получении данных: {e}"
        )
        return {
            "Ошибка в weaviate_data": str(e)
        }, 500
EOT

```

Начальная тема диалога, заданная в коде, – это «*Joe Biden*» (Джо Байден). Вы можете задать любую другую начальную тему по своему усмотрению. Вы также вольны менять имена и характер персонажей и их голоса.

Для запуска приложения Flask и сервера Weaviate выполните следующую команду:

```
docker-compose up --build
```

В качестве первого шага инициализируйте диалог, обратившись по адресу `/init`:

```
curl -X GET http://localhost:5000/init
```

Затем запустите диалог обращением по адресу /chat:

```
curl -X GET http://localhost:5000/chat?role=assistant
```

Вы должны увидеть в терминале ответ помощника. Он будет выглядеть примерно так:

```
{
  "full_result": {
    "data": {
      "Get": {
        "Chat": [
          {
            "_additional": {
              "distance": 0.077032566,
              "generate": {
                "error": null,
                "groupedResult": "Конечно, давайте окунемся в мир Джо Байдена. Знаете ли вы, что до того, как стать 46-м президентом США, Байден работал спасателем в бассейне в своем родном городе Уилмингтон, штат Делавэр? Он утверждает, что эта работа помогла ему развить чувство сочувствия и понимания расовой дискриминации. Разве не удивительно, как опыт ранней жизни может сформировать будущее человека? Что вы думаете по этому поводу?"
              }
            },
            "content": "Давайте поговорим на эту тему: Джо Байден.",
            "index": 0,
            "role": "user"
          }
        ]
      }
    },
    "interlocutor_data": {
      "content": "Давайте поговорим на эту тему: Джо Байден.",
      "index": 0,
      "role": "user"
    },
    "result_audio": "static/1.mp3",
    "result_text": "Конечно, давайте окунемся в мир Джо Байдена. Знаете ли вы, что до того, как стать 46-м президентом США, Байден работал спасателем в бассейне в своем родном городе Уилмингтон, штат Делавэр? Он утверждает, что эта работа помогла ему развить чувство сочувствия и понимания расовой дискриминации. Разве не удивительно, как опыт ранней жизни может сформировать будущее человека? Что вы думаете по этому поводу?"
  }
}
```

Аудиофайл ответа сохранен в папке static.

Далее обратимся к адресу /chat еще раз, чтобы получить ответ пользователя:

```
curl -X GET http://localhost:5000/chat?role=user
```

Этот процесс можно повторять неоднократно.

После определенного количества сообщений (когда сработает ограничение

по счетчику сообщений) один персонаж завершит разговор, а за ним это сделает другой, отмечая конец разговора. Теперь вы можете воспроизвести аудиофайлы в папке `static`, чтобы услышать разговор двух персонажей ИИ (воспроизводите аудиофайлы в их естественном порядке: `0.mp3`, `1.mp3`, `2.mp3`, ...).

ИСПОЛЬЗОВАНИЕ АВАТАРОВ МОДЕЛИ

Теперь наша цель – создать «говорящую голову», которая делает диалог более наглядным и увлекательным. Существует много инструментов для создания ИИ-аватаров, таких как Synthesia¹, Rosebud² и RunwayML³.

Другой подход заключается в использовании инструментов с открытым кодом, таких как SadTalker⁴ и DeepFaceLab⁵, предназначенных для создания генеративного видео. В нашем примере для генерации изображения говорящего персонажа применяется SadTalker.

Начнем с установки необходимых пакетов.

Первым делом создадим виртуальную рабочую среду для Python 3.8:

```
mkvirtualenv -p python3.8 ai_avatar
```

Затем установим пакеты:

```
git clone https://github.com/OpenTalker/SadTalker.git
```

```
cd SadTalker
```

```
pip install \
torch==1.12.1+cu113 \
torchvision==0.13.1+cu113 \
torchaudio==0.12.1 \
--extra-index-url \
https://download.pytorch.org/whl/cu113
```

```
pip install ffmpeg
```

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

```
bash scripts/download_models.sh
```

Если вы работаете под Windows, действуйте согласно следующей инструкции:

- установите Python 3.8 и удостоверьтесь, что путь к нему прописан в системной переменной PATH;
- установите Git;
- установите FFmpeg согласно руководству⁶;

¹ <https://www.synthesia.io/>.

² <https://rosebud.ai/>.

³ <https://runwayml.com/>.

⁴ <https://sadtalker.github.io/>.

⁵ <https://github.com/iperov/DeepFaceLab>.

⁶ <https://www.wikihow.com/Install-FFmpeg-on-Windows>.

- скачайте репозиторий SadTalker при помощи команды `git clone https://github.com/winfredy/SadTalker.git`;
- скачайте контрольные точки и модель GFPGAN по ссылке с Google Drive¹;
- выполните командный сценарий `start.bat` как обычный пользователь (не администратор) для запуска демонстрации WebUI Gradio².

Я рекомендую ознакомиться с официальной документацией³ SadTalker, чтобы в деталях разобраться с процессом установки.

К этому моменту файловая структура вашего проекта должна выглядеть так:

```

├── app
│   ├── app.py
│   ├── Dockerfile.app
│   ├── __pycache__
│   ├── requirements.txt
│   └── static
├── docker-compose.yaml
├── SadTalker
│   ├── app_sadtalker.py
│   ├── checkpoints
│   ├── cog.yaml
│   ├── docs
│   ├── examples
│   ├── gfpgan
│   ├── inference.py
│   ├── launcher.py
│   ├── LICENSE
│   ├── predict.py
│   ├── quick_demo.ipynb
│   ├── README.md
│   ├── req.txt
│   ├── requirements3d.txt
│   ├── requirements.txt
│   ├── results
│   ├── scripts
│   ├── src
│   ├── webui.bat
│   └── webui.sh
├── weaviate
│   └── Dockerfile.weaviate

```

Что касается непосредственно изображений аватаров, вы можете сгенерировать два разных изображения с помощью любого онлайн-инструмента, или скопировать их из репозитория SadTalker. Различные изображения хранятся в папке по адресу `SadTalker/examples/source_image`.

Пользователи Linux могут получить изображения при помощи таких команд:

¹ https://drive.google.com/file/d/1gwWh45pF7aelNP_P78uDIL8Sycep-K7j/view?usp=sharing.

² <https://www.gradio.app/>.

³ <https://github.com/OpenTalker/SadTalker>.

```
# Пользователь
cp SadTalker/examples/source_image/art_1.png \
app/static/user.png
```

```
# Помощник
cp SadTalker/examples/source_image/art_2.png \
app/static/assistant.png
```

Далее выполните следующую команду для запуска генерации видео из первого аудиофайла:

```
python \
SadTalker/inference.py \
--driven_audio ../app/static/0.mp3 \
--source_image ../app/static/user.png \
--enhancer gfpgan
```

Аналогичным образом сгенерируйте второе видео:

```
python \
SadTalker/inference.py \
--driven_audio ../app/static/1.mp3 \
--source_image ../app/static/assistant.png \
--enhancer gfpgan \
--result_dir app/static
```

Сделайте то же самое для всех остальных звуковых файлов.

Генерацию можно автоматизировать при помощи скрипта (для примера представим, что вы работаете под Linux и у вас есть 10 аудиофайлов):

```
#!/bin/bash

# Цикл для четных номеров
for i in {0..10..2}; do
    python SadTalker/inference.py \
    --driven_audio app/static/$i.mp3 \
    --source_image app/static/user.png \
    --enhancer gfpgan \
    --result_dir app/static
done

# Цикл для нечетных номеров
for i in {1..10..2}; do
    python SadTalker/inference.py \
    --driven_audio app/static/$i.mp3 \
    --source_image app/static/assistant.png \
    --enhancer gfpgan \
    --result_dir app/static
done
```

В результате для каждого аудиофайла будет сгенерирован видеофайл, сохраненный в папке static.

Что дальше?

В этой главе вы научились создавать автономный диалог между двумя персонажами ИИ, используя модели OpenAI и Weaviate. Вы также научились генерировать аватары ИИ, чтобы визуализировать разговор и сделать его более интересным. Для создания говорящей головы я выбрал SadTalker, но вы можете воспользоваться и другими инструментами, такими как Synthesia, Rosebud или RunwayML. Вы можете применить для создания дипфейковых видео и инструменты с открытым исходным кодом, такие как SadTalker и DeepFaceLab.

В качестве следующего шага для закрепления понимания и навыков вы можете реализовать приложение AutoGPT, где пользователь вводит тему, а приложение генерирует разговор между более чем двумя агентами искусственного интеллекта. При этом каждый агент представляет свою точку зрения или экспертные знания по теме. Эти агенты ИИ способны имитировать групповую дискуссию, предоставляя различные идеи, аргументы и точки зрения. Например, в дискуссии об изменении климата один агент может представлять ученого-эколога, другой – экономиста, а третий – политика. Разговор будет развиваться в зависимости от заданной темы, при этом каждый агент ИИ будет вносить свой вклад в соответствии с назначенной ему ролью, что приведет к раскрытию предмета разговора с разных ракурсов.

Послесловие

Благодарю вас за то, что нашли время изучить экосистему OpenAI. Я убежден, что это путешествие по передовым рубежам искусственного интеллекта и машинного обучения с акцентом на API OpenAI и сопутствующий набор инструментов и методов было полезным.

Модели GPT представляют собой значительный технологический прорыв в машинной обработке естественного языка, знаменуя кардинальную перемену в разработке приложений искусственного интеллекта.

Уникальные возможности моделей GPT и их потенциальная применимость в рамках правильно подобранной архитектуры ИИ делают их незаменимым дополнением к любому набору инструментов. Вполне возможно, что этот инструмент станет «рычагом», который раздвинет существующие границы ИИ.

Оставайтесь на связи и будьте в курсе всех последних событий в экосистемах Python и AI/ML, присоединившись к моему сообществу разработчиков по адресу www.faun.dev/join. Подписчики получают еженедельный информационный бюллетень, в котором собраны лучшие материалы по теме ИИ. Эти информационные бюллетени составлены на основе коллективных знаний, которыми поделились эксперты в области искусственного интеллекта и разработки программного обеспечения. Не упустите возможность быть в курсе развивающихся тенденций в мире ИИ и разработки программного обеспечения.

Я рад, что вы присоединились к сообществу разработчиков и пользователей ИИ, и с нетерпением жду возможности поделиться с вами новыми идеями. Пожалуйста, не стесняйтесь присылать свои отзывы и пожелания. Я с нетерпением ожидаю услышать ваше мнение и принять участие в содержательных дискуссиях.

Предметный указатель

В

Векторная база данных 204
Встраивание 106

Г

Генеративная модель 17
Генеративный поиск 226

Д

Дополненная генерация ответа 173

Л

Лемматизация 129

М

Модель
 большая языковая 13
 встраивания 36
 генеративная 17
 детерминированная 58
 завершения 39
 модерации 28
 мультимодальная 29

Н

Накопленная вероятность 58
Нечеткий поиск 137

О

Обработка естественного языка 17

Обучение

 на нескольких примерах 43
 на ограниченных примерах 153
 с подкреплением 13
 с учителем 13

П

Последовательность остановки 48
Промпт 67
 разработка 67

Р

Расстояние Левенштейна 137

С

Самовнимание 17
Стемминг 129

Т

Токен 49
Токенизация 129
Тонкая настройка 154
Точность 147

G

GPT-4 17

T

Transformer 17

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «КТК Галактика» наложенным платежом, выслать открытку или письмо по почтовому адресу:

115487, г. Москва, пр. Андропова д. 38 оф. 10.

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.

Оптовые закупки: тел. (499) 782-38-89.

Электронный адрес: books@aliens-kniga.ru.

Аймен Эль Амри

GPT-4. Руководство по использованию API Open AI

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Яценков В. С.*

Корректор *Синяева Г. И.*

Верстка *Луценко С. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «PT Serif». Печать цифровая.

Усл. печ. л. 22,26. Тираж 100 экз.

Веб-сайт издательства: www.dmkpress.com

Книга представляет собой подробное пошаговое руководство по использованию моделей GPT-3/3.5/4 в ваших проектах. Помимо этого, она охватывает такие модели, как Whisper и TTS (преобразование текста в речь).

К концу чтения вы разработаете:

- точно настроенную систему прогнозирования категорий новостей;
- чат-боты, ориентированные на конкретную предметную область;
- автономную диалоговую систему двух ИИ для имитации человеческих разговоров и обсуждения проблем;
- интеллектуальный помощник, рекомендующий сорта кофе на основе ваших предпочтений;
- систему психологической консультации на базе ИИ, обученную на большом наборе данных вопросов и ответов о психоэмоциональном состоянии.

Не нужно быть специалистом по данным, инженером по машинному обучению или экспертом в области искусственного интеллекта, чтобы реализовать представленные здесь идеи, методы и учебные примеры.

Простые и понятные объяснения содержат базовый код на языке Python.



Аймен Эль Амри – инженер-программист, предприниматель и автор с большим опытом работы в различных областях технологий, включая DevOps и Cloud Native, NLP и многое другое. Подготовил сотни инженеров-программистов, стал автором множества книг и курсов. Является основателем нескольких проектов, включая FAUN и Marketto.



www.faun.dev



www.dmk.rf

Оптовая продажа:

КТК "Галактика"

books@alians-kniga.ru

Страница книги
на dmkpress.com



ISBN 978-5-93700-299-0



9 785937 002990 >