Кит Борн. Раскрытие потенциала данных с помощью генеративного ИИ и технологии RAG

Я уже более двух лет использую ChatGPT в быту, при подготовке публикаций и в преподавательской деятельности. В последнее время я думаю над внедрением LLM в процессы по основному месту работы. Готового решения на базе ChatGPT я не нашел, и все дороги, похоже ведут к использованию технологии Retrieval-Augmented Generation (RAG, Генерация, дополненная поиском). Я попросил ChatGPT рекомендовать мне книгу практической направленности для первого знакомства с RAG. И получил книгу Кита Борна, опубликованную на amazon в сентябре 2024 г.

Keith Bourne. Unlocking Data with Generative AI and RAG. – PublisherPackt Publishing, 2024. – 346 p.

Часть 1. Введение в генерацию, дополненную поиском (RAG)

Глава 1. Что такое генерация, дополненная поиском (RAG)

Глава 2. Лаборатория кода – полный конвейер RAG

Глава 3. Практическое применение RAG

Глава 4. Компоненты системы RAG

Глава 5. Управление безопасностью в приложениях RAG

Часть 2. Компоненты RAG

Глава 6. Сопряжение с RAG и Gradio

Глава 7. Ключевая роль векторов и магазинов Vector в RAG

Глава 8. Поиск сходства с помощью векторов

Глава 9. Количественная оценка RAG и визуализация

Глава 10. Ключевые компоненты RAG в LangChain

Глава 11. Использование LangChain для получения большего от RAG

Часть 3. Внедрение продвинутой RAG

Глава 12. Сочетание RAG с мощью агентов ИИ и LangGraph

Глава 13. Использование оперативного проектирования для повышения эффективности RAG

Глава 14. Передовые методы, связанные с RAG, для улучшения результатов

Предисловие

RAG позволяет системам ИИ выйти за рамки ограничений своих обучающих данных и получить доступ к актуальной и предметно-ориентированной информации, что делает их более универсальными, адаптируемыми и ценными в реальных сценариях. Эта книга служит путеводителем по миру RAG. Она наполнена примерами кода, демонстрирующими инструменты и технологии, такие как LangChain, векторный магазин Chroma и модели OpenAI ChatGPT-4o. Мы

рассмотрим основные темы, включая векторные хранилища, векторизацию, методы векторного поиска, оперативное проектирование, агенты искусственного интеллекта для приложений, связанных с RAG, а также методы оценки и визуализации результатов RAG.

Чтобы получить максимум пользы от этой книги

Читатели должны иметь базовое представление о программировании на Python и быть знакомы с концепциями машинного обучения. Знание обработки естественного языка (NLP) и LLM было бы полезным. Также полезен опыт работы с обработкой данных и управлением базами данных. Предполагается, что читатели имеют некоторый опыт работы со средами разработки ИИ, комфортно работают с API и имеют опыт работы в среде записных книжек Jupyter.

Программное и аппаратное обеспечение, описанное в книге: Python 3.x, LangChain, OpenAI API, Jupyter notebooks.

Вам потребуется доступ к среде разработки Python, которая поддерживает записные книжки Jupyter. Для многих примеров требуется ключ API OpenAI. Для некоторых глав могут потребоваться дополнительные ключи API для таких сервисов, как Tavily или Together AI, но в этих главах вы узнаете об их настройке. Для запуска более сложных примеров, особенно тех, в которых задействованы LLM, рекомендуется использовать машину с объемом оперативной памяти не менее 8 ГБ.

Вы можете скачать файлы с примерами кода с GitHub по адресу.

Часть 1. Введение в генерацию, дополненную поиском (RAG)

В этой части мы познакомим вас с RAG, расскажем о ее основах, преимуществах, проблемах и практическом применении в различных отраслях. Вы узнаете, как реализовать конвейер RAG с помощью Python, управлять рисками безопасности и создавать интерактивные приложения с помощью Gradio. Мы также рассмотрим ключевые компоненты систем RAG, включая индексацию, извлечение, генерацию и оценку, и продемонстрируем, как оптимизировать каждый этап для повышения производительности и удобства работы пользователей.

Глава 1. Что такое генерация, дополненная поиском (RAG)

RAG становится важным дополнением к набору инструментов генеративного искусственного интеллекта, использующим возможности больших языковых моделей (LLM) и интегрирующим их с внутренними данными компании.

Понимание RAG. Основы и принципы

Современные LLM впечатляют, но они никогда не видели личных данных вашей компании (надеюсь!). Это означает, что способность LLM помочь вашей компании в полной мере использовать свои данные очень ограничена. Этот барьер породил концепцию RAG, в которой вы используете мощь и возможности LLM, но сочетаете их со знаниями и данными, содержащимися во внутренних репозиториях данных вашей компании. В этом и заключается основная мотивация использования RAG: чтобы сделать новые данные доступными для LLM и значительно увеличить ценность, которую вы можете извлечь из этих данных.

Помимо внутренних данных, RAG также полезна в тех случаях, когда LLM не была обучена на общедоступных но недавних данных.

Преимущества RAG

К некоторым потенциальным преимуществам использования RAG относятся повышенная точность и релевантность, настройка, гибкость и расширение знаний модели за пределы обучающих данных.

RAG может значительно повысить точность и актуальность ответов, генерируемых LLM. RAG извлекает и включает конкретную информацию из базы данных или набора данных, как правило, в режиме реального времени, и гарантирует, что выходные данные основаны как на уже существующих знаниях модели, так и на самых актуальных данных, которые вы предоставляете напрямую.

RAG позволяет настраивать и адаптировать знания модели к вашей конкретной предметной области или сценарию использования. Указывая RAG на базы данных или наборы данных, непосредственно относящиеся к вашему приложению, вы можете настроить выходные данные модели таким образом, чтобы они точно соответствовали информации и стилю, которые наиболее важны для ваших

конкретных потребностей. Такая настройка позволяет модели предоставлять более целенаправленные и полезные ответы.

RAG обеспечивает гибкость с точки зрения источников данных, к которым модель может получить доступ. Вы можете применять RAG к различным структурированным и неструктурированным данным, включая базы данных, веб-страницы, документы и многое другое. Такая гибкость позволяет использовать различные источники информации и комбинировать их новыми способами для расширения возможностей модели. Кроме того, при необходимости можно обновлять или заменять источники данных, что позволяет модели адаптироваться к изменяющимся информационным ландшафтам.

LLM ограничены объемом своих обучающих данных. RAG преодолевает это ограничение, позволяя моделям получать доступ к информации, которая не была включена в их первоначальные обучающие наборы, и использовать ее. Это эффективно расширяет базу знаний модели без необходимости переобучения, делая LLM более универсальными и адаптируемыми к новым предметам или быстро развивающимся темам.

LLM является ключевым компонентом системы RAG. LLM могут предоставлять неверную информацию, также известную как галлюцинации. Эти галлюцинации могут проявляться несколькими способами, такими как выдуманные факты, неверные факты или даже бессмысленное словоблудие. Часто галлюцинация сформулирована таким образом, что может быть очень убедительной, что затрудняет ее идентификацию. Хорошо разработанное приложение RAG может устранить галлюцинации гораздо легче, чем при непосредственном использовании LLM.

Проблемы RAG

Использование RAG также сопряжено с некоторыми проблемами, которые включают в себя зависимость от качества внутренних данных, необходимость обработки и очистки данных, вычислительные издержки, более сложные интеграции и потенциальную информационную перегрузку.

Когда речь заходит о том, как данные могут повлиять на модель искусственного интеллекта, в кругах специалистов по обработке и анализу данных говорят : мусор на входе, мусор на выходе. Это означает, что если вы дадите модели плохие данные, она даст вам плохие результаты. RAG ничем не отличается. Эффективность RAG напрямую связана с качеством данных, которые она извлекает. Если база данных содержит устаревшую, предвзятую или неточную информацию, выходные данные, генерируемые RAG, скорее всего, будут страдать от тех же проблем.

Данные в глубинах компании имеют для нее большую ценность, но они не часто находятся в хорошей, доступной форме. Например, данные из выписок клиентов на основе PDF требуют большой обработки, чтобы их можно было преобразовать в формат, который может быть полезен для конвейера RAG.

Конвейер RAG вводит множество новых вычислительных этапов в процесс генерации ответа, включая извлечение, обработку и интеграцию данных. LLM становятся все быстрее с каждым днем, но даже самый быстрый отклик может составлять больше секунды, а в некоторых случаях может занять несколько секунд. Если вы объедините это с другими этапами обработки данных и, возможно, с несколькими вызовами LLM, результатом может стать значительное увеличение времени для получения ответа. Все это приводит к увеличению вычислительных затрат, что влияет на эффективность и масштабируемость системы. Как и в случае с любой другой ИТ-инициативой, организация должна сбалансировать преимущества повышенной точности и настройки с требованиями к ресурсам и потенциальной задержкой, вызванной этими дополнительными процессами.

Традиционно данные хранятся в источнике данных, который запрашивается различными способами, чтобы сделать их доступными для внутренних и внешних систем. Но при использовании RAG ваши данные находятся в нескольких формах и расположениях, таких как векторы в векторной базе данных, которые представляют те же данные, но в другом формате. Добавьте к этому сложность подключения этих различных источников данных к LLM и соответствующим механизмам, таким как векторный поиск, и вы получите значительное увеличение сложности. Такая повышенная сложность может быть ресурсоемкой. Поддержание этой интеграции с течением времени, особенно по мере

развития или расширения источников данных, еще больше усложняет работу и увеличивает затраты. Организациям необходимо инвестировать в технические знания и инфраструктуру, чтобы эффективно использовать возможности RAG, учитывая при этом быстрый рост сложности, которую несут с собой эти системы.

Системы, основанные на RAG, могут извлекать слишком много информации. Внедрение механизмов для решения этой проблемы так же важно, как и для того, чтобы справляться с ситуациями, когда не удается найти достаточно релевантной информации. Определение релевантности и важности полученной информации, которая должна быть включена в конечный результат, требует сложных механизмов фильтрации и ранжирования. Без этого качество генерируемого контента может быть скомпрометировано избытком ненужных или малозначимых деталей.

Хотя мы перечислили устранение галлюцинаций как преимущество использования RAG, галлюцинации действительно представляют собой одну из самых больших проблем для конвейеров RAG, если с ними не бороться должным образом. Хорошо спроектированное приложение RAG должно принять меры по выявлению и устранению галлюцинаций и пройти тщательное тестирование, прежде чем окончательный текст будет предоставлен конечному пользователю.

Типичное приложение RAG, как правило, имеет высокий уровень сложности, с большим количеством компонентов, которые необходимо оптимизировать для правильной работы всего приложения. Компоненты могут взаимодействовать друг с другом несколькими способами, часто с гораздо большим количеством шагов, чем в базовом конвейере RAG, с которого вы начинаете. Каждый компонент в конвейере требует значительного количества испытаний и тестов, включая оперативное проектирование и разработку, используемые вами LLM и способы их использования, различные алгоритмы и их параметры для извлечения, интерфейс, который вы используете для доступа к приложению RAG, а также множество других аспектов, которые вам потребуется добавить в ходе разработки.

Словарь RAG

LLM. Большая часть этой книги будет посвящена LLM. LLM — это технологии генеративного искусственного интеллекта, которые фокусируются на генерации текста. Мы не будем усложнять задачу, сосредоточившись на типе модели, которую использует большинство конвейеров RAG, — LLM. Тем не менее, мы хотели бы уточнить, что, хотя мы сосредоточимся в первую очередь на LLM, RAG также может быть применена к другим типам генеративных моделей, таких как модели для изображений, аудио и видео. В главе 14 мы сосредоточимся на этих других типах моделей и на том, как они используются в RAG. Некоторыми популярными примерами LLM являются модели OpenAI ChatGPT, модели Meta Llama, модели Gemini от Google и модели Claude от Anthropic.

Промты, дизайн промтов и промт-инжиниринг. Эти термины иногда используются как взаимозаменяемые, но технически они имеют разные значения:

- Промт – это акт отправки запроса LLM.
- Дизайн промта — это стратегия, которую вы реализуете для разработки промта. Многие из них мы рассмотрим в главе 13.
- Промт-инжиниринг основное внимание уделяет техническим аспектам, которые улучшают результат работы с LLM. Например, вы можете разбить сложный запрос на два или три взаимодействия с LLM. Мы также рассмотрим промт-инжиниринг в главе 13.

LangChain и LlamaIndex. В этой книге мы сосредоточимся на использовании LangChain в качестве основы для создания конвейеров RAG. LangChain — это фреймворк с открытым исходным кодом, который поддерживает не только RAG, но и любую разработку, которая хочет использовать LLM в рамках конвейерного подхода. С более чем 15 миллионами ежемесячных загрузок LangChain является самым популярным фреймворком для разработки генеративного ИИ. Он особенно хорошо поддерживает RAG, предоставляя модульный и гибкий набор инструментов, которые делают разработку RAG значительно более эффективной, чем отсутствие фреймворка. LlamaIndex является ведущей альтернативой LangChain с аналогичными возможностями. LlamaIndex известен своей ориентацией на задачи поиска и извлечения данных и может быть хорошим вариантом, если вам требуется расширенный поиск или вам нужно работать с большими наборами данных.

Вывод (Inference). Время от времени мы будем использовать термин *вывод*. В общем случае это относится к процессу генерации LLM выходных данных или прогнозов на основе заданных входных

данных с использованием предварительно обученной языковой модели. Например, когда вы задаете вопрос ChatGPT, шаги, необходимые для предоставления вам ответа, называются выводом.

Контекстное окно в LLM относится к максимальному количеству токенов, которые модель может обработать за один проход. Он определяет объем текста, который модель может видеть или обрабатывать одновременно при выполнении прогнозов или создании ответов. Размер контекстного окна является ключевым параметром архитектуры модели и обычно фиксируется во время обучения модели. Он напрямую связан с входным размером модели, поскольку устанавливает верхний предел количества токенов, которые могут быть поданы в модель за один раз.

Например, если размер контекстного окна модели равен 4 096 токенов, это означает, что модель может обрабатывать и создавать последовательности до 4 096 токенов. При обработке более длинных текстов, таких как документы или беседы, вводимые данные необходимо разделить на более мелкие сегменты, которые помещаются в контекстном окне. Это часто делается с помощью таких методов, как раздвижные окна или усечение.

Размер контекстного окна влияет на способность модели понимать и поддерживать долгосрочные зависимости и контекст. Модели с большими контекстными окнами могут захватывать и использовать больше контекстуальной информации при формировании ответов, что может привести к более согласованным и контекстуально релевантным результатам. Однако увеличение размера контекстного окна также увеличивает вычислительные ресурсы, необходимые для обучения и запуска модели.

В контексте RAG размер контекстного окна имеет важное значение, поскольку он определяет, какой объем информации из извлеченных документов может быть эффективно использован моделью при формировании окончательного ответа. Последние достижения в области языковых моделей привели к разработке моделей со значительно большими контекстными окнами, что позволяет им обрабатывать и сохранять больше информации из полученных источников. В таблице 1.1 приведены контекстные окна многих популярных LLM, как закрытых, так и с открытым исходным кодом:

LLM	Context Window (Tokens)
ChatGPT-3.5 Turbo 0613 (OpenAI)	4,096
Llama 2 (Meta)	4,096
Llama 3 (Meta)	8,000
ChatGPT-4 (OpenAI)	8,192
ChatGPT-3.5 Turbo 0125 (OpenAI)	16,385
ChatGPT-4.0-32k (OpenAI)	32,000
Mistral (Mistral AI)	32,000
Mixtral (Mistral AI)	32,000
DBRX (Databricks)	32,000
Gemini 1.0 Pro (Google)	32,000
ChatGPT-4.0 Turbo (OpenAI)	128,000
ChatGPT-4o (OpenAI)	128,000
Claude 2.1 (Anthropic)	200,000
Claude 3 (Anthropic)	200,000
Gemini 1.5 Pro (Google)	1,000,000

Рис. 1.1. Контекстные окна различных LLM

Обратите внимание, что старые модели, как правило, имели меньшие контекстные окна. Тенденция к росту вероятно, сохранится, увеличивая типичное контекстное окно в будущем.

Тонкая настройка: полная настройка (full-model fine-tuning, FMFT) или параметроэффективная настройка (parameter-efficient fine-tuning, PEFT). FMFT — это когда вы берете базовую модель и обучаете ее дальше, чтобы получить новые возможности. Вы можете просто дать новые знания для определенной области или вы можете дать ему навык, например, быть разговорным чат-ботом. FMFT обновляет все параметры и смещения в модели. PEFT, с другой стороны, — это тип тонкой настройки, при котором вы фокусируетесь только на определенных частях параметров или

смещений при тонкой настройке модели, но с той же целью, что и общая тонкая настройка. Последние исследования в этой области показывают, что вы можете достичь результатов, аналогичных FMFT, с гораздо меньшими затратами, временем и данными.

Хотя эта книга не фокусируется на тонкой настройке, очень хорошей стратегией является попытка использовать модель, тонко настроенную на ваши данные, чтобы дать ей больше знаний из вашей области или дать ей больше голоса из вашей области. Например, вы можете обучить ее говорить больше как ученый, чем как общая базовая модель, если вы используете ее в научной области. В качестве альтернативы, если вы развиваетесь в юридической сфере, вы можете захотеть, чтобы она звучала больше как юрист.

Векторный магазин или векторная база данных? Оба! Все векторные базы данных являются векторными хранилищами, но не все векторные хранилища являются векторными базами данных. Хорошо, пока вы достаете свою классную доску, чтобы нарисовать диаграмму Винна, я продолжу объяснять это утверждение.

Существуют способы хранения векторов, которые не являются полными базами данных. Они просто являются носителями векторов. Итак, чтобы охватить все возможные способы хранения векторов, LangChain называет их все векторными хранилищами. Давайте сделаем то же самое! Просто знайте, что не все векторные хранилища , к которым подключается LangChain, официально считаются векторными базами данных, но в целом большинство из них таковы, и многие люди называют их все векторными базами данных, даже если технически они не являются полноценными базами данных с точки зрения функциональности. Фух - рад, что мы прояснили это!

Векторное хранилище или векторная база данных? И то, и другое! Все векторные базы данных являются векторными хранилищами, но не все векторные хранилища являются полноценными базами данных. Хорошо, пока вы достаете свою классную доску для рисования диаграммы Венна, я продолжу объяснение. Существуют способы хранения векторов, которые не являются полноценными базами данных. Это просто устройства хранения для векторов. Поэтому, чтобы охватить все возможные способы хранения векторов, в LangChain используют общий термин **векторные хранилища**. Давайте придерживаться этого же подхода! Однако стоит помнить, что не все векторные хранилища, с которыми работает LangChain, официально считаются базами данных. Тем не менее, большинство из них можно отнести к этой категории, и многие называют их векторными базами данных, даже если они с технической точки зрения не обладают всеми функциями баз данных. Фух, надеюсь, теперь стало понятнее!

Векторы, векторы, векторы! Вектор — это математическое представление ваших данных. В контексте обработки естественного языка (NLP) и крупных языковых моделей (LLM) векторы часто называют *эмбеддингами*. Векторы являются одним из самых важных понятий для понимания, так как многие части конвейера RAG используют их.

Векторы

Можно утверждать, что понимание векторов и всех способов, которыми они используются в RAG, является самой важной частью этой книги. Векторы — это просто математические представления ваших внешних данных, и их часто называют эмбеддингами. Эти представления захватывают семантическую информацию в формате, который может быть обработан алгоритмами, облегчая выполнение таких задач, как поиск сходства, который является важным этапом в процессе RAG.

Векторы обычно имеют определенную размерность в зависимости от того, сколько чисел они представляют. Например, это четырехмерный вектор: [0,123; 0,321; 0.312; 0,231]. Если бы вы не знали, что речь идет о векторах, и увидели это в коде Python, вы могли бы принять это за список из четырех чисел с плавающей точкой – и вы были бы недалеки от истины. Однако при работе с векторами в Python предпочтительнее использовать массивы NumPy вместо списков. Массивы NumPy более удобны для машинного обучения, так как они оптимизированы для обработки данных гораздо быстрее и эффективнее, чем списки Python. Они также широко признаны стандартным представлением эмбеддингов в большинстве пакетов для машинного обучения, таких как SciPy, pandas, scikit-learn, TensorFlow, Keras, PyTorch и многие другие.

NumPy также позволяет выполнять векторизованные математические операции непосредственно над массивами, например, выполнять покомпонентные вычисления, без необходимости писать

циклы и использовать другие подходы, которые потребовались бы при работе с другими типами последовательностей.

При работе с векторами для векторизации часто существуют сотни или тысячи измерений. Более высокая размерность позволяет захватывать более подробную семантическую информацию, что имеет решающее значение для точного сопоставления входных данных запроса с соответствующими документами или данными в приложениях RAG.

Понимание векторов важная базовая концепция для понимания того, как реализовать RAG, но как RAG используется в практических приложениях на предприятии? Мы обсудим эти практические приложения RAG в области искусственного интеллекта в следующем разделе.

Внедрение RAG в приложения ИИ

RAG быстро становится краеугольным камнем платформ генеративного искусственного интеллекта в корпоративном мире. RAG сочетает в себе возможности извлечения внутренних или новых данных с генеративными языковыми моделями для повышения качества и актуальности генерируемого текста. Этот метод может быть особенно полезен для компаний в различных отраслях для улучшения их продуктов, услуг и операционной эффективности. Ниже приведены некоторые примеры использования RAG:

Поддержка клиентов и чат-боты: они могут существовать без RAG, но при интеграции с RAG они могут связать этих чат-ботов с прошлыми взаимодействиями с клиентами, часто задаваемыми вопросами, справочными документами и всем остальным, что было характерно для этого клиента.

Техническая поддержка: благодаря лучшему доступу к истории и информации о клиентах, чат-боты, усовершенствованные RAG, могут значительно улучшить текущие чат-боты технической поддержки.

Автоматизированная отчетность: RAG может помочь в создании первоначальных проектов или обобщении существующих статей, исследовательских работ и других типов неструктурированных данных в более удобоваримых форматах.

Поддержка электронной коммерции: RAG может помочь создать динамические описания продуктов и пользовательский контент, а также сделать более качественные рекомендации по продуктам.

Использование баз знаний: RAG улучшает возможности поиска и полезность как внутренних, так и общих баз знаний, генерируя резюме, предоставляя прямые ответы на запросы и извлекая соответствующую информацию в различных областях, таких как юриспруденция, исследования, медицина, научные разработки, патенты и техническая документация.

Поиск инноваций: это похоже на поиск в общих базах знаний, но с акцентом на инновации. Благодаря этому компании могут использовать RAG для сканирования и обобщения информации из качественных источников, чтобы определить тенденции и потенциальные области для инноваций, которые имеют отношение к специализации компании.

Обучение и образование: RAG может использоваться образовательными организациями и корпоративными учебными программами для создания или адаптации учебных материалов на основе конкретных потребностей и уровня знаний учащихся. С помощью RAG гораздо более глубокий уровень внутренних знаний от организации может быть включен в образовательную программу в соответствии с индивидуальными потребностями человека или роли.

Вы можете задаться вопросом: «Если я использую LLM, такую как ChatGPT, для ответа на свои вопросы в моей компании, означает ли это, что моя компания уже использует RAG?»

Ответ: «Нет».

Если вы просто входите в ChatGPT и задаете вопросы, это не то же самое, что внедрение RAG. И ChatGPT, и RAG являются формами генеративного ИИ, и иногда их используют вместе, но это две разные концепции. В следующем разделе мы обсудим различия между генеративным ИИ и RAG.

Сравнение RAG с обычным генеративным ИИ

Обычный генеративный ИИ уже показал себя революционным изменением для компаний, помогая их сотрудникам выйти на новый уровень производительности. LLM, такие как ChatGPT, помогают пользователям с быстро растущим списком приложений, которые включают в себя написание бизнес-планов, написание и улучшение кода, написание маркетинговых текстов и даже

предоставление более здоровых рецептов для определенной диеты. В конечном счете, большая часть того, что делают пользователи, выполняется быстрее.

Однако обычный генеративный ИИ не знает того, чего он не знает. И это включает в себя большую часть внутренних данных вашей компании. Можете ли вы представить, что вы могли бы сделать со всеми преимуществами, упомянутыми ранее, но в сочетании со всеми данными внутри вашей компании – обо всем, что ваша компания когда-либо делала, о ваших клиентах и всех их взаимодействиях, или обо всех ваших продуктах и услугах в сочетании со знанием потребностей конкретного клиента? Вам не нужно это представлять – это то, что делает RAG!

До появления RAG большинство сервисов, которые связывали клиентов или сотрудников с информационными ресурсами компании, были лишь малой частью того, что возможно по сравнению с тем, если бы они могли получить доступ ко всем данным в компании. С появлением RAG и генеративного искусственного интеллекта в целом, корпорации оказались на пороге чего-то очень, очень большого.

Еще одна область, с которой вы можете спутать RAG, — это концепция тонкой настройки модели. Давайте обсудим, в чем разница между этими типами подходов.

Сравнение RAG с тонкой настройкой модели

LLM могут быть адаптированы к вашим данным двумя способами:

- Тонкая настройка: при тонкой настройке вы корректируете веса и/или смещения, которые определяют интеллект модели на основе новых обучающих данных. Это напрямую влияет на модель, постоянно изменяя то, как она будет взаимодействовать с новыми входными данными.
- Промт: вы используете расширенный промт для введения в модель новых знаний, на основе которых LLM может действовать.

Почему бы не использовать тонкую настройку во всех ситуациях? Как только вы познакомитесь с новыми знаниями, они всегда будут у LLM! Это также то, как была создана модель - путем обучения на данных, верно? В теории это звучит правильно, но на практике тонкая настройка оказалась более надежной при обучении модели специализированным задачам (например, обучению модели разговаривать определенным образом) и менее надежной для запоминания фактов.

Причина сложная, но в целом знание фактов моделью подобно долговременной памяти человека. Если вы запомните длинный отрывок из речи или книги, а затем попытаетесь вспомнить его через несколько месяцев, вы, скорее всего, все еще будете понимать контекст информации, но можете забыть конкретные детали. С другой стороны, добавление знаний с помощью входных данных модели похоже на нашу кратковременную память, где факты, детали и даже порядок формулировок очень свежи и доступны для памяти. Именно этот последний сценарий лучше подходит в ситуации, когда вы хотите успешного запоминания фактов. А учитывая, насколько более дорогостоящей может быть тонкая настройка, это делает гораздо более важным рассмотрение RAG.

Однако есть и компромисс. Несмотря на то, что обычно существуют способы передачи всех имеющихся данных в модель для тонкой настройки, входные данные ограничены контекстным окном модели. Это та сфера, которой активно занимаются. Например, ранние версии ChatGPT 3.5 имели контекстное окно из 4 096 токенов, что эквивалентно примерно пяти страницам текста. Когда был выпущен ChatGPT 4, они расширили контекстное окно до 8 192 токенов (10 страниц), и была версия Chat 4 – 32k, которая имела контекстное окно 32 768 токенов (40 страниц). Этот вопрос настолько важен, что они включили размер контекстного окна в название модели. Это сильный показатель того, насколько важно контекстное окно!

По мере расширения контекстных окон создается еще одна проблема. Ранние модели с расширенными контекстными окнами теряли много деталей, особенно в середине текста. Этот вопрос также решается. Модель Gemini 1.5 с контекстным окном в 1 миллион токенов показала хорошие результаты в тестах, называемых тестами «иголка в стоге сена», на хорошее запоминание всех деталей по всему тексту, который она может принять в качестве входа. К сожалению, модель не показала таких же хороших результатов в тестах с несколькими иголками в стоге сена. Ожидайте больших усилий в этой области по мере увеличения этих контекстных окон. Имейте это в виду, если вам нужно работать с большими объемами текста за один раз.

Важно отметить, что подсчет лексем отличается от подсчета слов, так как лексемы включают знаки препинания, символы, цифры и другие текстовые представления. То, как составное слово, такое как мороженое , обрабатывается с точки зрения токенов, зависит от схемы токенизации и может варьироваться в зависимости от LLM. Но большинство известных LLM (таких как ChatGPT и Gemini) рассматривают мороженое как два токена. При определенных обстоятельствах в NLP вы можете утверждать, что это должен быть один токен, основываясь на концепции, что токен должен представлять собой полезную семантическую единицу для обработки, но это не так для этих моделей.

RAG, как правило, лучше подходит для извлечения фактической информации, которая отсутствует в обучающих данных LLM или является конфиденциальной. Она позволяет динамически интегрировать внешние знания без изменения весов модели. Тонкая настройка, с другой стороны, больше подходит для обучения модели специализированным задачам или ее адаптации к конкретной предметной области. Учитывайте ограничения размеров контекстных окон и возможность переобучения при тонкой настройке определенного набора данных.

Архитектура систем RAG

Этапы процесса RAG с точки зрения пользователя:

А. Пользователь вводит запрос.
Б. Приложение некоторое время думает, прежде чем проверить данные, к которым у него есть доступ, чтобы увидеть, какие из них наиболее актуальны.
В. Приложение предоставляет ответ, который фокусируется на ответе на вопрос пользователя, но с использованием данных, которые были предоставлены ему через конвейер RAG.

С технической точки зрения, есть два этапа, которые вы будете кодировать: извлечение и генерация. Но есть еще один этап, известный как индексирование, который может быть выполнен и часто выполняется до того, как пользователь вводит запрос. С помощью индексирования вы преобразуете вспомогательные данные в векторы, сохраняете их в векторной базе данных и, вероятно, оптимизируете функциональность поиска, чтобы шаг извлечения был максимально быстрым и эффективным.

После того, как пользователь передает свой запрос в систему, выполняются следующие шаги:

А. Пользовательский запрос векторизуется.
Б. Векторизованный запрос передается в векторный поиск для извлечения наиболее релевантных данных в векторной базе данных, представляющей внешние данные.
В. Векторный поиск возвращает наиболее релевантные результаты и уникальные ключи, ссылающиеся на исходное содержимое, которое эти векторы представляют.
Г. Уникальные ключи используются для извлечения исходных данных, связанных с этими векторами, часто в пакете из нескольких документов.
Д. Исходные данные могут быть отфильтрованы или обработаны, но, как правило, затем передаются в LLM на основе того, что вы ожидаете от процесса RAG.
Е. К LLM прилагается подсказка, в которой обычно говорится что-то вроде: «Вы полезный помощник в задачах с ответами на вопросы. Возьмите следующий вопрос (запрос пользователя) и используйте эту полезную информацию (данные, полученные при поиске сходства) для ответа на него. Если вы не знаете ответа на основе предоставленной информации, просто скажите, что не знаете.
Ж. LLM обрабатывает этот запрос и предоставляет ответ на основе предоставленных вами внешних данных. В зависимости от области применения системы RAG эти шаги могут быть выполнены в режиме реального времени или такие шаги, как индексирование, могут быть выполнены до запроса, чтобы он был готов к поиску, когда придет время.

Как упоминалось ранее, мы можем разбить эти аспекты на три основных этапа (см. рисунок 1.2):

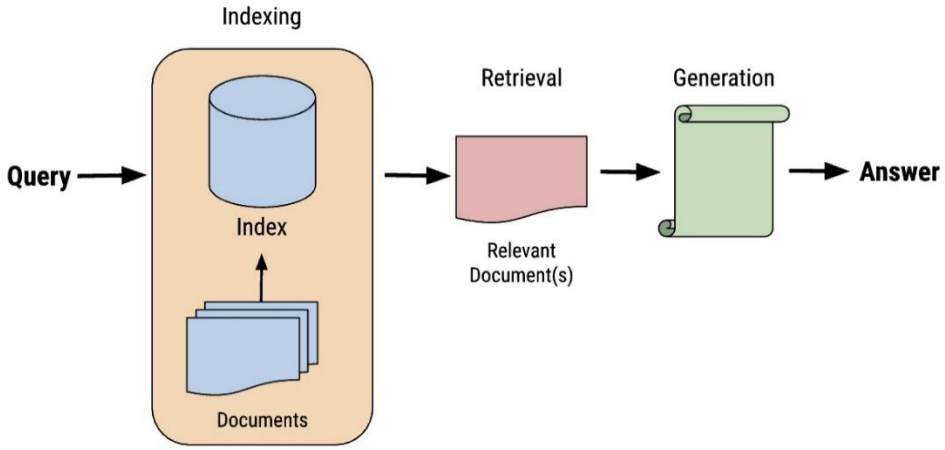

Рисунок 1.2. Три стадии RAG

Глава 2. Лаборатория кода – полный конвейер RAG

Эта глава закладывает основу для всего остального кода в книге. Мы создадим весь конвейер генерации, дополненной поиском (RAG). Затем, по мере продвижения по книге, мы будем рассматривать различные части кода, добавляя усовершенствования, чтобы вы понимали, как код может развиваться для решения все более сложных задач. В этой главе мы рассмотрим компоненты конвейера RAG:

- Нет интерфейса
- Настройка учетной записи большой языковой модели (LLM) в OpenAI
- Установка пакетов Python
- Индексирование данных путем просмотра веб-страниц, разбивки документов и встраивания фрагментов
- Поиск релевантных документов с помощью поиска по векторному подобию
- Генерирование ответов путем интеграции найденного контекста в подсказки LLM

По мере изучения кода вы получите полное представление о каждом этапе процесса RAG с помощью таких инструментов, как LangChain, Chroma DB и API OpenAI. Это обеспечит прочный фундамент, на который мы будем опираться в последующих главах, улучшая и развивая код для решения все более сложных задач.

Технические требования

Код для этой главы доступен здесь. Вам потребуется выполнить код этой главы в среде, настроенной для работы с блокнотами Jupyter. Опыт работы с блокнотами Jupyter является обязательным условием для использования этой книги, и слишком сложно охватить его в коротком объеме текста. Существует множество способов настройки среды блокнотов. Существуют онлайн-версии, версии, которые можно скачать, среды блокнотов, которые предоставляют студентам университеты, и различные интерфейсы, которые вы можете использовать.

Если вы работаете в компании, то у них, скорее всего, есть среда, с которой вы захотите ознакомиться. Для настройки каждого из этих вариантов требуются совершенно разные инструкции, и эти инструкции часто меняются. Если вам нужно подтянуть свои знания об этой среде, начните с сайта Jupyter, а затем обратитесь к вашей любимой LLM за дополнительной помощью в настройке среды.

Я использую свой Chromebook, часто в поездках, я использую блокнот, настроенный в одной из облачных сред. Я предпочитаю Google Colab или их ноутбуки Colab Enterprise, которые можно найти в разделе Vertex AI на Google Cloud Platform. Но эти среды стоят денег.

В качестве экономичной альтернативы, когда я настолько активен, я использую Docker Desktop на моем Mac, который размещает кластер Kubernetes локально, и настраиваю свое окружение ноутбука в кластере. Все эти подходы имеют ряд требований к окружающей среде, которые часто меняются. Лучше всего провести небольшое исследование и выяснить, что лучше всего подходит для вашей ситуации. Существуют аналогичные решения для компьютеров на базе Windows. Главное требование – найти среду, в которой можно запустить Jupyter notebook с помощью Python 3. В коде, который мы предоставим, будет указано, какие еще пакеты вам нужно установить.

Весь код предполагает, что вы работаете в блокноте Jupyter. Вы можете сделать это прямо в файле Python (.py), но вам, возможно, придется изменить часть кода. Запуск в блокноте дает вам возможность пройтись по коду строка за строкой и посмотреть, что происходит в каждой точке, чтобы лучше понять весь процесс.

Нет интерфейса!

В следующем примере кодирования мы не будем работать с интерфейсами – о нем мы расскажем в главе 6. Пока же мы просто создадим строковую переменную, которая будет представлять подсказку, вводимую пользователем, и используем ее в качестве замены полноценного интерфейсного ввода.

Настройка учетной записи большой языковой модели (LLM)

Для широкой публики наиболее популярными и известными LLM в настоящее время являются модели ChatGPT от OpenAI. Однако на рынке существует множество других LLM, которые подходят для самых разных целей. Вам не всегда нужно использовать самую дорогую и самую мощную LLM.

Некоторые LLM сфокусированы на одной области, например, Meditron LLM, которые являются медицинскими исследовательскими версиями Llama 2 с тонкой настройкой. Если вы работаете в области медицины, возможно, вы захотите использовать эту LLM, так как она может оказаться лучше, чем большая неспециализированная LLM. Иногда LLM могут использоваться для перепроверки других LLM, в таких случаях нужно иметь несколько аккаунтов. Я рекомендую искать LLM, которая лучше всего подходит для ваших нужд. Но чтобы не усложнять книгу, в начале я расскажу о настройке ChatGPT от OpenAI:

Перейдите в раздел API на сайте OpenAI: https://openai.com/api/.

Если вы еще не создали учетную запись, сделайте это сейчас. Веб-страница может часто меняться, но ищите, где можно зарегистрироваться.

> Использование API OpenAI стоит денег! Используйте его экономно!

После регистрации перейдите к документации по адресу https://platform.openai.com/docs/quickstart и следуйте инструкциям по настройке вашего первого ключа API.

При создании ключа API дайте ему запоминающееся имя и выберите тип разрешений, которые вы хотите реализовать (Все, Ограниченный или Только чтение). Если вы не знаете, какой вариант выбрать, лучше всего выбрать "Все". Однако обратите внимание на другие варианты - возможно, вы захотите разделить различные обязанности с другими членами команды, но ограничить определенные типы доступа:

- Все: этот ключ будет иметь доступ для чтения/записи ко всем API OpenAI.
- Ограниченный: появится список доступных API, позволяющий контролировать, к каким API имеет доступ ключ. У вас есть возможность предоставить доступ к каждому API только на чтение или только на запись. Убедитесь, что вы включили API для моделей и встраивания, которые вы будете использовать в этих демонстрациях.
- Только чтение: эта опция предоставляет доступ ко всем API только для чтения.

Скопируйте предоставленный ключ. Вскоре вы добавите его в свой код. Тем временем имейте в виду, что если этот ключ будет передан кому-либо еще, тот, кому вы его предоставили, сможет им воспользоваться, а вам будет предъявлен счет. Так что этот ключ следует считать совершенно секретным и принять надлежащие меры предосторожности, чтобы предотвратить его несанкционированное использование.

API OpenAI требует, чтобы вы заранее купили кредиты для использования API. Покупайте столько, сколько вам удобно, а для большей безопасности убедитесь, что опция "Включить автоматическое пополнение" выключена. Это позволит вам тратить только то, что вы намерены потратить.

Таким образом, вы настроили ключевой компонент, который будет служить мозгом в вашем конвейере RAG: LLM! Далее мы настроим вашу среду разработки так, чтобы вы могли подключиться к LLM.

Установка необходимых пакетов

Убедитесь, что эти пакеты установлены в вашей среде Python. Добавьте следующие строки кода в первую ячейку вашего блокнота:

```
%pip install langchain_community
%pip install langchain_experimental
%pip install langchain-openai
%pip install langchainhub
%pip install chromadb
%pip install langchain
%pip install beautifulsoup4
```

Код устанавливает несколько библиотек Python с помощью менеджера пакетов pip, которые понадобятся вам для запуска кода. Описание каждой библиотеки:

langchain_community. Это пакет сообщества для библиотеки LangChain, которая представляет собой фреймворк с открытым исходным кодом для создания приложений с LLM. Она предоставляет набор инструментов и компонентов для работы с LLM и их интеграции в различные приложения.

langchain_experimental. Библиотека предлагает дополнительные возможности и инструменты за пределами основной библиотеки LangChain, которые еще не полностью стабильны или готовы к производству, но доступны для экспериментов и исследований.

langchain-openai. Пакет обеспечивает интеграцию между LangChain и языковыми моделями OpenAI. Он позволяет легко включать модели OpenAI, такие как ChatGPT 4 или сервис встраивания OpenAI, в ваши приложения LangChain.

langchainhub. Пакет предоставляет коллекцию готовых компонентов и шаблонов для приложений LangChain. Он включает в себя различных агентов, компоненты памяти и полезные функции, которые могут быть использованы для ускорения разработки приложений на основе LangChain.

chromadb. Это пакет для Chroma DB, высокопроизводительной встраиваемой/векторной базы данных, предназначенной для эффективного поиска и извлечения сходств.

langchain. Это основная библиотека LangChain. Она предоставляет фреймворк и набор абстракций для создания приложений с LLM. LangChain включает в себя компоненты, необходимые для эффективного конвейера RAG, в том числе промты, управление памятью, агентов и другие интеграции с различными внешними инструментами и сервисами.

После выполнения предыдущей первой строки вам нужно будет перезапустить ядро, чтобы получить доступ ко всем новым пакетам, которые вы только что установили в среду. В зависимости от того, в какой среде вы находитесь, это можно сделать разными способами. Как правило, вы увидите кнопку обновления или опцию Restart kernel в меню.

Импорт

Теперь давайте импортируем библиотеки, необходимые для выполнения задач, связанных с RAG. Я снабдил каждую группу импортируемых библиотек комментариями, чтобы указать, к какой области RAG они относятся. Это, в сочетании с описанием в следующем списке, дает базовое представление обо всем, что вам нужно для вашего первого конвейера RAG:

```
import os
from langchain_community.document_loaders import WebBaseLoader
import bs4
import openai
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain import hub
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
import chromadb
from langchain_community.vectorstores import Chroma
from langchain_experimental.text_splitter import SemanticChunker
```

Давайте рассмотрим каждый из этих импортов:

import os обеспечивает взаимодействие с операционной системой. Он полезен для доступа к переменным окружения и работы с путями к файлам.

from langchain_community.document_loaders import WebBaseLoader – это загрузчик документов, который может получать и загружать веб-страницы как документы.

import bs4 (bs4 расшифровывается как Beautiful Soup 4) – это популярная библиотека для веб-скрапинга и разбора HTML или XML-документов. Поскольку мы будем работать с веб-страницей, это дает нам простой способ извлечь отдельно заголовок, содержимое и иные элементы.

import openai обеспечивает интерфейс для взаимодействия с API OpenAI.

from langchain_openai import ChatOpenAI, OpenAIEmbeddings импортирует ChatOpenAI (для LLM) и OpenAIEmbeddings (для эмбеддинга)

from langchain import hub предоставляет доступ к различным готовым компонентам и утилитам для работы с языковыми моделями.

from langchain_core.output_parsers import StrOutputParser разбирает вывод, сгенерированный языковой моделью, и извлекает из него необходимую информацию. В данном случае он предполагает, что вывод модели языка является строкой, и возвращает его как есть.

from langchain_core.runnables import RunnablePassthrough пропускает через себя или запрос без каких-либо изменений. Он позволяет использовать вопрос как есть на последующих этапах цепочки.

import chromadb импортирует векторное хранилище Chroma DB, базу данных эмбеддинга/векторов, предназначенную для эффективного поиска и извлечения сходства.

from langchain_community.vectorstores import Chroma обеспечивает интерфейс для взаимодействия с векторной базой данных Chroma с помощью LangChain.

from langchain_experimental.text_splitter import SemanticChunkerp. Разделитель текста – это, как правило, функция, которую мы используем для разбиения текста на небольшие фрагменты на основе заданного размера фрагмента и перекрытия. Этот сплиттер называется SemanticChunker, экспериментальная утилита для разбиения текста, предоставляемая библиотекой Langchain_experimental. Основная цель SemanticChunker – разбить длинный текст на более удобные фрагменты, сохраняя при этом семантическую связность и контекст каждого фрагмента.

Эти импорты содержат основные пакеты Python, которые понадобятся для настройки конвейера RAG. Следующим шагом будет подключение вашей среды к API OpenAI.

Подключение OpenAI

Следующая строка кода показывает как ваш API-ключ попадает в систему. Однако такой способ использования API-ключа не является безопасным. Существует множество способов сделать это более безопасно. Если у вас есть предпочтения, реализуйте их сейчас, но в противном случае мы рассмотрим популярный способ сделать это более безопасным в главе 5.

Вам нужно заменить решетки на ваш реальный ключ API OpenAI:

```
os.environ['OPENAI_API_KEY'] = 'sk-####################'
openai.api_key = os.environ['OPENAI_API_KEY']
```

Вы, наверное, догадались, что этот API-ключ OpenAI будет использоваться для подключения к ChatGPT LLM. Но ChatGPT – не единственный сервис, который мы будем использовать от OpenAI. Этот API-ключ также используется для доступа к сервису встраивания OpenAI. В следующем разделе, посвященном кодированию этапа индексирования процесса RAG, мы воспользуемся сервисом встраивания OpenAI для преобразования вашего контента в векторные встраивания – ключевой аспект конвейера RAG.

Индексирование

Следующие несколько шагов представляют собой этап индексирования, на котором мы получаем целевые данные, предварительно обрабатываем их и векторизуем. Эти шаги часто выполняются в автономном режиме, то есть для подготовки приложения к последующему использованию. Но в некоторых случаях имеет смысл делать все это в реальном времени, например, в быстро меняющихся средах данных, где используемые данные относительно невелики. В данном конкретном примере шаги выглядят следующим образом:

1. Загрузка и сканирование web-страниц.

2. Разбивка данных на удобные фрагменты (чанки, chunks) для алгоритма векторизации Chroma DB.
3. Встраивание (эмбеддинг, embedding) и индексирование этих фрагментов.
4. Добавление чанков и эмбеддингов в векторное хранилище Chroma DB.

Загрузка и просмотр веб-страниц

Для нашего примера я привожу пример веб-страницы, основанный на материалах из главы 1. Оригинальную структуру я взял из примера, предоставленного LangChain по адресу. Вы можете попробовать и эту веб-страницу, если она все еще доступна, но не забудьте изменить вопрос, который вы используете для запроса содержимого, на вопрос, более подходящий для содержимого той страницы. Также необходимо перезапустить ядро при смене веб-страницы; в противном случае при повторном запуске загрузчика в него будет включено содержимое обеих веб-страниц.

Я также рекомендую вам попробовать сделать это с другими веб-страницами и посмотреть, какие проблемы возникают на этих страницах. Этот пример включает в себя очень чистые данные по сравнению с большинством веб-страниц, которые, как правило, изобилуют рекламой и другим контентом, который вы не хотите видеть. Но, может быть, вы сможете найти относительно чистую запись в блоге и вставить ее? Может быть, вы сможете создать свой собственный?

```
loader = WebBaseLoader(
   web_paths=("https://kbourne.github.io/chapter1.html",),
   bs_kwargs=dict(
      parse_only=bs4.SoupStrainer(
         class_=("post-content", "post-title",
            "post-header")
      )
   ),
)
docs = loader.load()
```

Код начинается с использования класса WebBaseLoader из модуля langchain_community document_loaders для загрузки веб-страниц в качестве документов.

Создается экземпляр WebBaseLoader: класс WebBaseLoader с параметрами:

- web_paths: кортеж, содержащий URL-адреса веб-страниц, которые должны быть загружены. В данном случае он содержит один URL: https://kbourne.github.io/chapter1.html.
- bs_kwargs: словарь аргументов ключевых слов, передаваемых парсеру BeautifulSoup.
- parse_only: объект bs4.SoupStrainer задает элементы HTML для разбора. В данном случае он установлен в значение для разбора только элементов с классами CSS, таких как post-content, post-title и post-header.

Экземпляр WebBaseLoader инициирует серию шагов, которые представляют собой загрузку документа в вашу среду: метод load вызывается для loader, экземпляре WebBaseLoader, который получает и загружает указанные веб-страницы в качестве документов. Вот шаги, которые выполняет loader, основываясь на этом небольшом количестве кода:

- HTTP-запросы к указанным URL-адресам для получения веб-страниц.
- Разбирает HTML-содержимое веб-страниц с помощью BeautifulSoup, учитывая только те элементы, которые указаны в параметре parse_only.
- Извлекает соответствующее текстовое содержимое из разобранных HTML-элементов.
- Создает объекты Document для каждой веб-страницы, которые содержат извлеченный текстовый контент, а также метаданные, такие как URL-адрес источника.

Полученные объекты Document сохраняются в переменной docs для дальнейшего использования в коде.

Классы, которые мы передаем в bs4 (post-content, post-title и post-header), являются классами CSS. Если вы используете HTML-страницу, на которой нет этих CSS-классов, это не сработает. Поэтому, если вы используете другой URL-адрес и не получаете данных, посмотрите, какие теги CSS есть в HTML- странице, которую вы просматриваете. Многие веб-страницы используют этот шаблон, но не все! При просмотре веб-страниц возникает множество проблем, подобных этой.

После того как вы собрали документы из источника данных, их необходимо предварительно обработать. В данном случае это включает в себя разбиение на части.

Разделение

Если вы используете указанный URL-адрес, будут обработаны только элементы с CSS-классами post-content, post-title и post-header. В результате будет извлечен текстовый контент из основного тела статьи (обычно обозначается классом post-content), заголовок записи блога (обычно обозначается классом post-title) и любая информация заголовка (обычно обозначается классом post-header).

Вот как выглядит этот документ в Интернете:

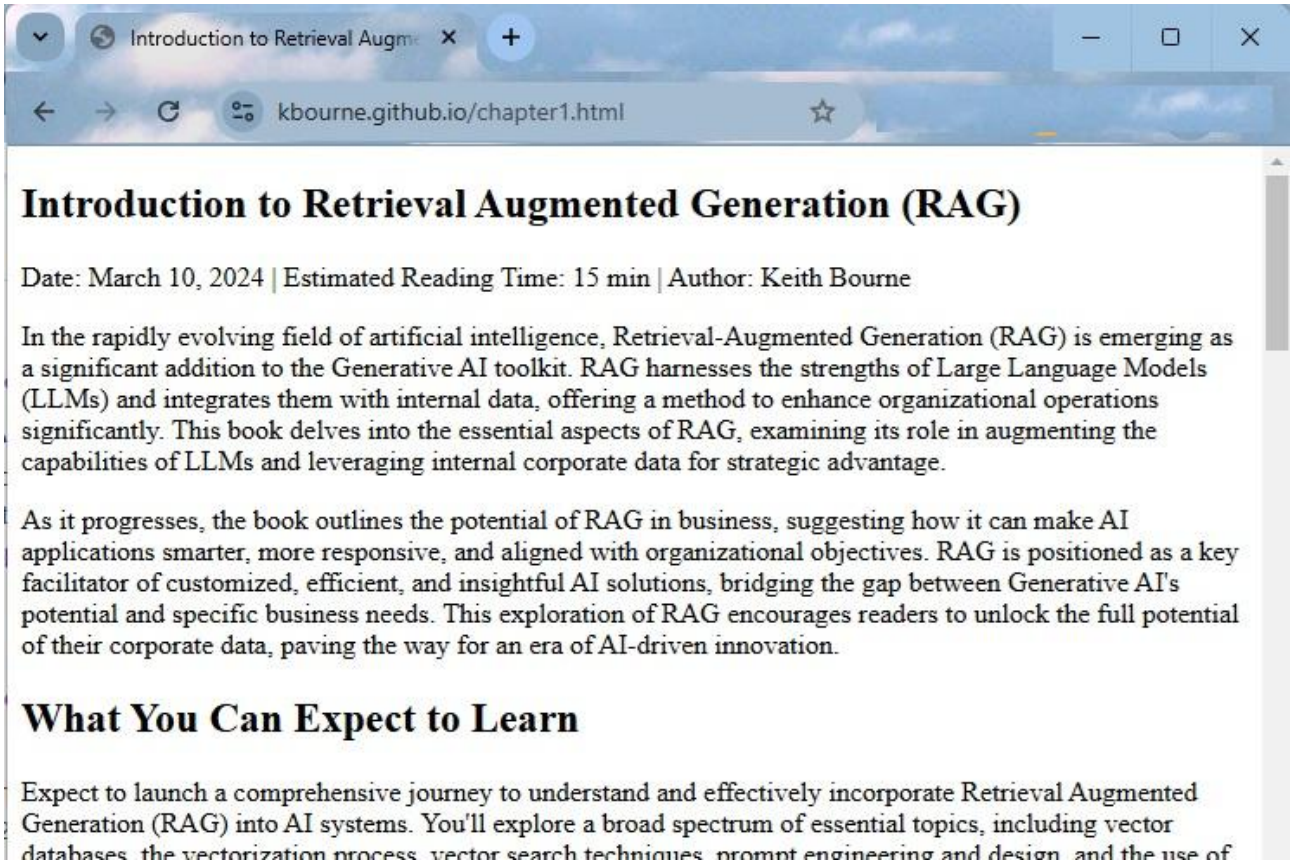

Рис. 2.1. Веб-страница, которую мы будем обрабатывать

На этой странице очень много контента, слишком много для того, чтобы LLM могла обработать его напрямую. Поэтому нам придется разделить документ на удобные для восприятия фрагменты:

```
text_splitter = SemanticChunker(OpenAIEmbeddings())
splits = text_splitter.split_documents(docs)
```

В LangChain доступно множество разделителей текста, но я решил начать с экспериментального, но очень интересного варианта под названием SemanticChunker. Как я уже упоминал ранее, говоря об импорте, SemanticChunker фокусируется на разбиении длинного текста на более удобные фрагменты, сохраняя при этом семантическую связность и контекст каждого фрагмента.

Другие разделители текста обычно используют произвольную длину фрагмента, не учитывающую контекст, что создает проблемы, когда важное содержимое оказывается разделено разделителем. Есть способы решения этой проблемы, о которых мы поговорим в главе 11, а пока просто знайте, что SemanticChunker фокусируется на учете контекста, а не просто произвольной длины ваших кусков. Следует также отметить, что он все еще считается экспериментальным и находится в стадии постоянной разработки. В главе 11 мы испытаем его в сравнении с другим, вероятно, самым важным разделителем текста, RecursiveCharacter TextSplitter, и посмотрим, какой разделитель лучше работает с этим контентом.

Следует также отметить, что сплиттер SemanticChunker, который вы используете в этом коде, использует OpenAIEmbeddings, а обработка эмбеддингов стоит стоит денег. Модели эмбеддинга OpenAI в настоящее время стоят от 0,02 до 0,13 доллара за миллион токенов, в зависимости от того,

какую модель вы используете. На момент написания статьи, если вы не укажете модель эмбеддинга, OpenAI по умолчанию будет использовать модель text-embedding-ada-002, которая стоит 0,02 доллара за миллион токенов. Если вы хотите избежать расходов, вернитесь к RecursiveCharacter TextSplitter, о котором мы расскажем в главе 11.

Я рекомендую вам попробовать разные сплиттеры и посмотреть, что получится! Например, как вы думаете, RecursiveCharacter TextSplitter дает лучшие результаты, чем SemanticChunker, который мы используем здесь? Возможно, в вашем конкретном случае скорость важнее качества – какой из них быстрее?

После того как вы разбили контент на куски, следующим шагом будет преобразование его в векторные эмбеддинги.

Эмбеддинг и индексирование фрагментов

Следующие несколько шагов представляют собой шаги поиска и генерации, где мы будем использовать Chroma DB в качестве базы данных векторов. Я выбрал это векторное хранилище, потому что его легко запустить локально, и оно хорошо подходит для демонстраций, подобных этой, но это довольно мощное векторное хранилище. Как вы помните, когда мы говорили о словарном запасе и разнице между векторными хранилищами и векторными базами данных, Chroma DB действительно является и тем, и другим! В главе 7 мы обсудим многие варианты векторных хранилищ и причины, по которым стоит выбрать один из них. Некоторые из этих вариантов даже предоставляют бесплатную генерацию векторных вложений.

Здесь мы также используем эмбеддинги OpenAI, которые с помощью ключа отправят фрагменты данных в API OpenAI, преобразуют их в эмбеддинги, а затем отправят обратно в математической форме. Обратите внимание, что это стоит денег! Это доли пенни за каждый эмбеддинг, но все же. Поэтому будьте осторожны при использовании этого кода, если у вас ограниченный бюджет! В главе 7 мы рассмотрим некоторые способы использования бесплатных сервисов векторизации для бесплатной генерации эмбеддингов:

```
vectorstore = Chroma.from_documents(
   documents=splits,
   embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()
```

Сначала мы создаем векторное хранилище Chroma с помощью метода Chroma.from_documents, который вызывается для создания векторного хранилища Chroma из разделенных документов. Это один из многих методов, которые мы можем использовать для создания базы данных Chroma. Обычно это зависит от источника, но в данном конкретном методе он принимает следующие параметры:

- documents – список разделенных документов (splits), полученный из предыдущего фрагмента кода
- embedding – экземпляр класса OpenAIEmbeddings, который используется для генерации эмбеддингов для документов

Внутри метод выполняет несколько действий:

- Итерацию над каждым объектом Document в списке Splits.
- Для каждого объекта Document он использует предоставленный экземпляр OpenAIEmbeddings для создания вектора встраивания.
- Он сохраняет текст документа и соответствующий ему вектор встраивания в векторной базе данных Chroma. Теперь у вас есть база данных векторов под названием vectorstore, и она полна эмбеддингов, которые являются...? Правильно – математические представления всего содержимого веб-страницы, которую вы только что просмотрели!

retriever создает механизм, который вы будете использовать для поиска векторного сходства в векторной базе данных. Вы вызываете метод as_retriever прямо на экземпляре vectorstore. retriever – это объект, который предоставляет удобный интерфейс для выполнения поиска по сходству и извлечения соответствующих документов из векторной базы данных на основе этого поиска. Если вы хотите выполнить процесс поиска документов, вы можете это сделать. Это не является официальной частью кода, но если вы хотите проверить, добавьте код новую ячейку и запустите его:

```
query = "How does RAG compare with fine-tuning?"
relevant_docs = retriever.get_relevant_documents(query)
relevant_docs
```

На выходе должно получиться то, что я перечислю позже в этом коде, когда укажу, что передается в LLM, но по сути это список содержимого, хранящегося в векторной базе данных vectorstore, которое наиболее похоже на запрос.

На данном этапе вы создали приемник. Вы еще не использовали его в конвейере RAG. Далее мы рассмотрим, как это сделать!

Извлечение и генерация

В коде этапы поиска и генерации объединяются в цепочку, которую мы создали для представления всего процесса RAG. Для этого используются готовые компоненты из LangChain Hub, такие как шаблоны промтов, и они интегрируются с выбранной LLM. Мы также используем язык LangChain Expression Language (LCEL) для определения цепочки операций, которые извлекают документы на основе входного вопроса, форматируют извлеченный контент и передают его в LLM для создания ответа. Поиск и генерация выполняются следующим образом:

1. Примите запрос пользователя.
2. Векторизируйте запрос пользователя.
3. Выполните поиск по сходству в хранилище векторов, чтобы найти наиболее близкие векторы к вектору запроса пользователя, а также их связанное содержимое.
4. Передайте полученное содержимое в шаблон промта; в контексте LLM процесс заполнения шаблона промта называют гидратацией (hydrating).
5. Передайте этот заполненный промт LLM.
6. Получите ответ от LLM и представьте его пользователю.

С точки зрения кодирования мы начнем с определения шаблона промта, чтобы у нас было что заполнять при получении пользовательского запроса.

Шаблоны промтов из LangChain Hub

LangChain Hub – это коллекция готовых компонентов и шаблонов, которые можно легко интегрировать в приложения LangChain. Он представляет собой централизованное хранилище для обмена и поиска компонентов многократного использования, таких как промты, агенты и утилиты. Здесь мы вызываем шаблон промта из LangChain Hub и присваиваем его нашеиу шаблону:

```
prompt = hub.pull("jclemens24/rag-prompt")
print(prompt)
```

Этот код извлекает предварительно созданный шаблон подсказки из LangChain Hub, используя метод pull модуля hub. Шаблон промта идентифицируется строкой jclemens24/rag-prompt. Этот идентификатор соответствует соглашению репозиторий/компонент, где репозиторий представляет организацию или пользователя, размещающего элемент, а компонент – конкретный извлекаемый элемент. Компонент rag-prompt указывает на то, что это промт, предназначенный для приложений RAG.

print(prompt) выводит этот шаблон:

```
input_variables=['context', 'question'] input_types={} partial_variables={} metadata={'lc_hub_owner': 'jclemens24', 'lc_hub_repo': 'rag-prompt', 'lc_hub_commit_hash': '1a1f3ccb9a5a92363310e3b130843dfb2540239366ebe712ddd94982acc06734'} messages=[HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['context', 'question'], input_types={}, partial_variables={}, template="You are an assistant for question-answering tasks. Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know.\nQuestion: {question} \nContext: {context} \nAnswer:"), additional_kwargs={})]
```

```
/usr/local/lib/python3.10/dist-packages/langsmith/client.py:261: LangSmithMissingAPIKeyWarning: API key must be provided when using hosted LangSmith API
```

```
warnings.warn(
```

Это начальная часть запроса, которая передается в LLM и в данном случае говорит ему следующее:

"Вы являетесь ассистентом для выполнения заданий с вопросами. Используйте следующие фрагменты найденного контекста, чтобы ответить на вопрос. Если вы не знаете ответа, просто скажите, что не знаете.

Вопрос: {question}

Контекст: {context}

Ответ: "

Позже вы добавите вопрос и контекстные переменные, чтобы наполнить промт конкретикой, но начало работы с этим форматом оптимизирует его для работы с приложениями RAG.

Строка jclemens24/rag-prompt – это один из вариантов предопределенных стартовых промтов. Посетите LangChain Hub, чтобы найти множество других – возможно, вы даже найдете тот, который лучше подходит для ваших целей.

Шаблон промта – это ключевая часть конвейера RAG, поскольку он представляет собой способ взаимодействия с LLM для получения искомого ответа. Но в большинстве конвейеров RAG получение запроса в формате, позволяющем работать с шаблоном запроса, не просто передача ему строки. В этом примере контекстная переменная представляет собой содержимое, которое мы получаем от ретривера, а оно еще не имеет строкового формата! Далее мы рассмотрим, как преобразовать полученное содержимое в нужный нам строковый формат.

Форматирование функции таким образом, чтобы она соответствовала входным данным следующего шага

Сначала мы создадим функцию, которая будет принимать на вход список найденных документов

def format_docs(docs): return "\n\n".join(doc.page_content for doc in docs)
Внутри этой функции генераторное выражение (doc.page_content for doc in docs) используется для извлечения атрибута page_content из каждого объекта документа. Атрибут page_content представляет собой текстовое содержимое каждого документа.

В данном случае документ – это не весь документ, который вы просмотрели ранее. Это лишь небольшая его часть, но мы обычно называем их документами.

Метод join вызывается для строки \n\n, чтобы объединить содержимое страниц каждого документа с двумя символами новой строки между содержимым каждого документа. Отформатированная строка возвращается функцией format_docs и представляет собой контекстный ключ в словаре, который передается в объект prompt.

Цель этой функции – отформатировать вывод ретривера в строковый формат, в котором он должен быть на следующем шаге цепочки, после шага ретривера. Мы объясним это позже, но короткие функции, подобные этой, часто необходимы в цепочках LangChain для согласования входов и выходов по всей цепочке.

Далее мы рассмотрим последний шаг перед созданием нашей цепочки LangChain – определение LLM, который мы будем использовать в этой цепочке.

Определите свою LLM

Давайте создадим модель LLM, которую вы будете использовать:

llm = ChatOpenAI(model_name="gpt-4o-mini", temperature=0)

Приведенный выше код создает экземпляр класса ChatOpenAI из модуля langchain_openai, который служит интерфейсом для языковых моделей OpenAI, в частности для модели GPT-4o mini. Несмотря на то, что эта модель является более новой, она была выпущена со значительной скидкой по сравнению со старыми моделями. Использование этой модели поможет вам снизить затраты на создание выводов и при этом позволит использовать новейшую модель. Если вы хотите попробовать другую версию ChatGPT, например gpt-4, вы можете просто изменить название модели. Ищите самые новые модели на сайте OpenAI API – они часто добавляют их!

Настройка цепочки LangChain с помощью LCEL

Эта цепочка выполнена в формате кода, характерном для LangChain, который называется LCEL. С этого момента вы увидите, что я использую LCEL во всем коде. Это не только облегчает чтение кода и

делает его более лаконичным, но и открывает новые техники, направленные на повышение скорости и эффективности вашего кода LangChain.

Если вы пройдете по этой цепочке, то увидите, что она отлично отображает весь процесс RAG:

```
rag_chain = (
   {"context": retriever | format_docs,
    "question": RunnablePassthrough()}
      | prompt
      | llm
      | StrOutputParser()
)
```

Все эти компоненты уже были описаны, но если подвести итог, то переменная rag_chain представляет собой цепочку операций с использованием фреймворка LangChain. Давайте пройдемся по каждому шагу этой цепочки, разбираясь с тем, что происходит в каждой точке.

Извлечение. Первое звено – извлечение. Рассмотрим этот шаг еще более подробно.

Когда мы будем обращаться к переменной rag_chain, мы передадим ей "вопрос". Цепочка начинается со словаря, в котором определены два ключа: "context" и "question". С вопросом все понятно, а вот откуда берется контекст? Назначенный ключ "context" является результатом работы ретривера| format_docs.

format_docs – функция, которую мы определили выше. Оператор| , называемый pipe (пайп, конвейер), между retriever и format_docs указывает на то, что мы соединяем эти операции в цепочку. Здесь объект retriever передается в функцию format_docs. Мы запускаем операцию retriever, которая представляет собой поиск векторного сходства. Он вернет набор совпадений, который и передается в функцию. format_docs использует содержимое, предоставленное ретривером, чтобы отформатировать результаты в одну строку. Эта строка присваивается контексту, который является переменной в промте.

Ожидаемый формат ввода на следующем шаге – это словарь с двумя ключами, то есть "контекст" и "вопрос". Значения, которые присваиваются этим ключам, должны быть строками. Поэтому мы не можем просто передать ретриверу вывод, который представляет собой список объектов. Именно поэтому мы используем функцию format_docs.

Переданный в цепочку вопрос (question) уже имеет формат строки. Нам не нужно дополнительного форматирования! Поэтому мы используем объект RunnablePassthrough(), чтобы просто пропустить этот входной сигнал (переданный вопрос) в виде строки, в которой он уже отформатирован. Этот объект принимает вопрос , который мы передаем в переменную rag_chain, и пропускает его через себя без каких-либо изменений. Теперь у нас есть первый шаг в цепочке, который заключается в определении двух переменных, которые принимает промт на следующем шаге.

Мы видим еще один пайп |, за которым следует объект prompt, и мы передаем переменные (в словаре) в этот объект prompt. Это и есть заполнение (гидратирование) промта. prompt – это шаблон, определяющий, что мы будем передавать в LLM, и он обычно включает входные переменные (контекст и вопрос), которые сначала заполняются/гидратируются. В результате этого шага появляется полный текст промта в виде строки, а переменные заполняют места контекста и вопроса.

Следующий пайп | добавляет llm. Этот шаг в цепочке принимает результат предыдущего шага – промта (шаблон + контекст + вопрос). Объект llm – установленная нами языковую модель, в данном случае ChatGPT 4o. Отформатированная строка промта передается на вход языковой модели, которая генерирует ответ на основе предоставленного контекста и вопроса.

Кажется, что этого достаточно, но когда вы используете LLM API, он не просто отправляет вам текст, который вы видите, когда вводите что-то в ChatGPT. Он представлен в формате JSON и содержит множество других данных. Поэтому, чтобы не усложнять ситуацию, мы собираемся передать вывод LLM на следующий шаг и использовать объект StrOutputParser() из LangChain. Обратите внимание, что StrOutputParser() – это служебный класс в LangChain, который разбирает ответ языковой модели в строковый формат. Он не только удаляет лишнюю информацию, но и гарантирует, что сгенерированный ответ будет возвращен в виде строки.

Итак, цепочка, которую мы создали с помощью LangChain, представляет собой основной код для конвейера RAG, и она состоит всего из нескольких строк!

Цепочка начнет работу с пользовательского запроса. Но с точки зрения кодирования мы настраиваем все остальное, чтобы правильно обработать запрос. На данный момент мы готовы принять запрос пользователя, поэтому давайте рассмотрим этот последний шаг в нашем коде.

Подача вопроса в RAG

Пока что вы определили цепочку, но не запустили ее. Поэтому давайте запустим весь конвейер RAG в этой одной строке, используя запрос:

rag_chain.invoke("What are the advantages of using RAG?")

"What are the advantages of using RAG?" – это строка, которую мы собираемся передать в цепочку. Первый шаг цепочки ожидает эту строку как вопрос, в качестве одной из двух переменных. В некоторых приложениях она может быть не в правильном формате, и потребуется дополнительная функция для ее подготовки, но в данном приложении она уже находится в ожидаемом формате строки, поэтому мы передаем ее прямо в объект RunnablePassThrough().

В будущем этот промт будет включать в себя запрос из пользовательского интерфейса, но пока мы будем представлять ее в виде этой переменной строки. Имейте в виду, что это не единственный текст, который увидит LLM; ранее вы указали, что prompt управляется переменными "context" и "question".

И это все с точки зрения кодирования! Но что происходит, когда вы запускаете код? Давайте рассмотрим результаты, которые вы можете ожидать от этого кода RAG pipeline.

Окончательный результат

Итоговый результат будет выглядеть примерно так:

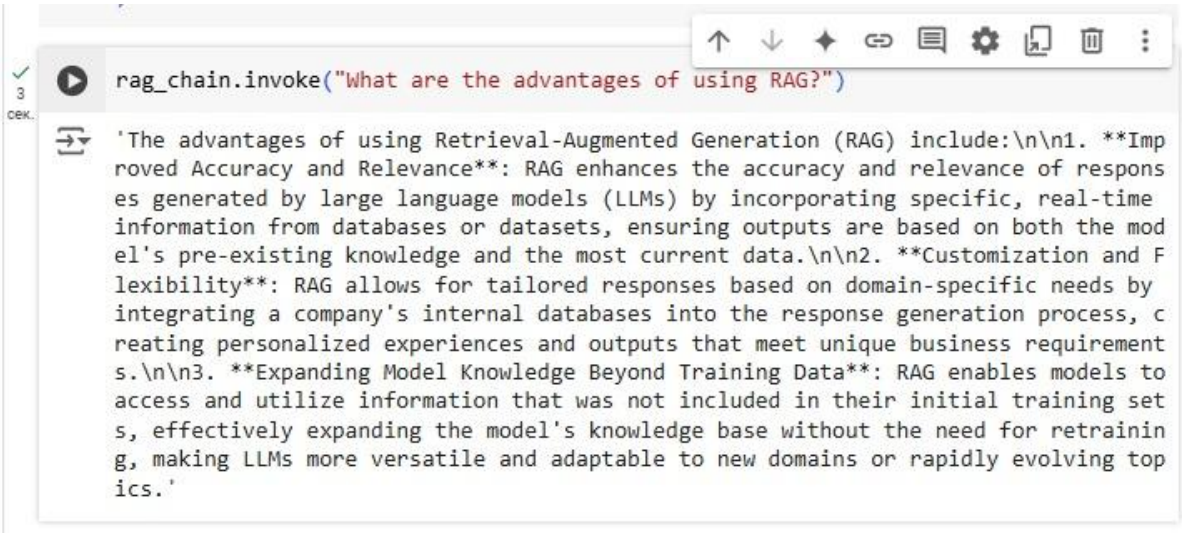

Рис. 2.2. Результат запроса в Jupyter notebooks

В нем есть базовое форматирование, поэтому при отображении он будет выглядеть следующим образом (включая буллиты и жирный текст):

The advantages of using Retrieval-Augmented Generation (RAG) include:

- **Improved Accuracy and Relevance**: RAG enhances the accuracy and relevance of responses generated by large language models (LLMs) by incorporating specific, real-time information from databases or datasets, ensuring outputs are based on both the model's pre-existing knowledge and the most current data.
- **Customization and Flexibility**: RAG allows for tailored responses based on domain-specific needs by integrating a company's internal databases into the response generation process, creating personalized experiences and outputs that meet unique business requirements.

- **Expanding Model Knowledge Beyond Training Data**: RAG enables models to access and utilize information that was not included in their initial training sets, effectively expanding the model's knowledge base without the need for retraining, making LLMs more versatile and adaptable to new domains or rapidly evolving topics.

Вот что увидит LLM, когда вы объедините вопрос с контекстом, извлеченным с помощью RAG:

> "You are an assistant for question-answering tasks. Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know.
>
> Question: What are the Advantages of using RAG?
>
> Context: Can you imagine what you could do with all of the benefits mentioned above, but combined with all of the data within your company, about everything your company has ever done, about your customers and all of their interactions, or about all of your products and services combined with a knowledge of what a specific customer's needs are? You do not have to imagine it, that is what RAG does! Even smaller companies are not able to access much of their internal data resources very effectively. Larger companies are swimming in petabytes of data that are not readily accessible or are not being fully utilized. Before RAG, most of the services you saw that connected customers or employees with the data resources of the company were just scratching the surface of what is possible compared to if they could access ALL of the data in the company. With the advent of RAG and generative AI in general, corporations are on the precipice of something really, really big. Comparing RAG with Model Fine-Tuning#\nEstablished Large Language Models (LLM), what we call the foundation models, can be learned in two ways:\n Fine-tuning - With fine-tuning, you are adjusting the weights and/or biases that define the model\'s intelligence based
>
> [TRUNCATED FOR BREVITY!]
>
> Answer:"

Как видите, контекст довольно большой – он возвращает всю наиболее релевантную информацию из исходного документа, чтобы помочь LLM определить, как ответить на новый вопрос. Контекст – это то, что было возвращено в результате поиска по векторному сходству, о чем мы подробнее поговорим в главе 8.

Полный код

Вот код в полном объеме:

```
%pip install langchain_community
%pip install langchain_experimental
%pip install langchain-openai
%pip install langchainhub
%pip install chromadb
%pip install langchain
%pip install beautifulsoup4
```

> У меня после этого кода возвращалась ошибка. ChatGPT для исправления ошибки предложил установить еще две библиотеки:
>
> ```
> %pip install jedi
> !apt-get update && apt-get install -y libcairo2-dev
> %pip install pycairo
> ```

Перед выполнением следующего кода перезапустите ядро:

> У меня код работал и без перезапуска ядра

```
import os
from langchain_community.document_loaders import WebBaseLoader
import bs4
import openai
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain import hub
from langchain_core.output_parsers import StrOutputParser
```

```
from langchain_core.runnables import RunnablePassthrough
import chromadb
from langchain_community.vectorstores import Chroma
from langchain_experimental.text_splitter import SemanticChunker
os.environ['OPENAI_API_KEY'] = 'sk-####################'
openai.api_key = os.environ['OPENAI_API_KEY']
#### INDEXING ####
loader = WebBaseLoader(
   web_paths=("https://kbourne.github.io/chapter1.html",),
   bs_kwargs=dict(parse_only=bs4.SoupStrainer(
      class_=("post-content",
         "post-title",
         "post-header")
      )
   ),
)
docs = loader.load()
text_splitter = SemanticChunker(OpenAIEmbeddings())
splits = text_splitter.split_documents(docs)
vectorstore = Chroma.from_documents(
   documents=splits,
   embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()
#### RETRIEVAL and GENERATION ####
prompt = hub.pull("jclemens24/rag-prompt")
def format_docs(docs):
   return "\n\n".join(doc.page_content for doc in docs)
llm = ChatOpenAI(model_name="gpt-4o-mini")
rag_chain = (
   {"context": retriever | format_docs,
    "question": RunnablePassthrough()}
      | prompt
      | llm
      | StrOutputParser()
)
rag_chain.invoke("What are the Advantages of using RAG?")
```

На этом все! Вы можете закрыть эту книгу и по-прежнему быть в состоянии построить целое приложение RAG. Удачи! Но прежде чем вы уйдете, необходимо рассмотреть еще много концепций, чтобы оптимизировать ваш RAG-конвейер. Если вы сделаете поиск в Интернете по запросу "Проблемы с RAG" или чему-то подобному, вы найдете миллионы вопросов, где приложения RAG имеют проблемы со всеми приложениями, кроме самых простых. Остальная часть книги посвящена тому, чтобы помочь вам накопить знания, которые помогут преодолеть любую из этих проблем и сформировать множество новых решений.