Глава 4. Компоненты системы RAG

Это продолжение перевода книги Кит Борн. Раскрытие потенциала данных с помощью генеративного ИИ и технологии RAG. В главе 1 мы познакомили с генерацией, дополненной поиском (RAG), в главе 2 разобрали код конвейера RAG, в главе 3 рассмотрели области применения RAG. При разработке RAG важно понимать тонкости каждого компонента, способы их интеграции и технологии, которые расширяют возможности этих систем. В этой главе мы рассмотрим следующие темы:

- Обзор ключевых компонентов
- Индексация
- Извлечение и генерация
- Создание промтов
- Определение языковой модели
- Пользовательский интерфейс
- Оценка

Предыдущая глава Содержание Следующая глава

Код для этой главы размещен в репозитории GitHub.

Обзор ключевых компонентов

Начнем с обзора всей системы. В главе 1 мы представили три основных этапа системы RAG с технической точки зрения: индексация, извлечение, генерация

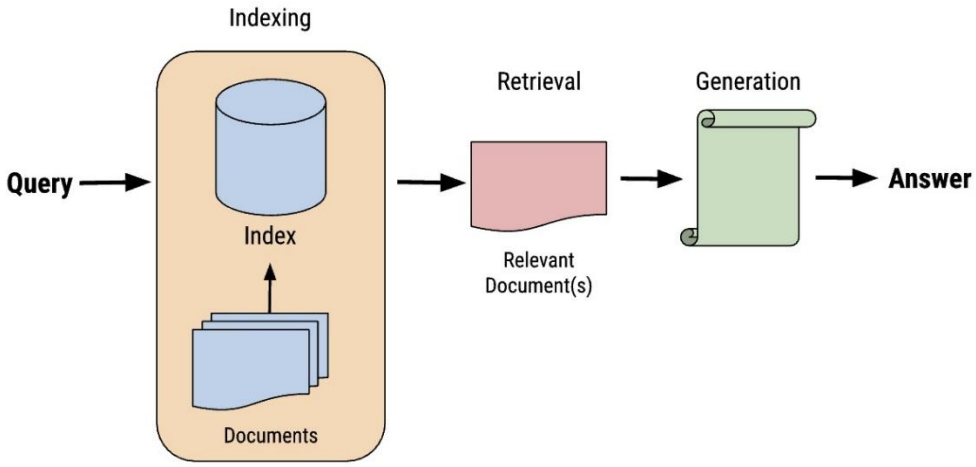

Рис. 4.1. Три этапа системы RAG

Мы продолжим развивать эту концепцию, но также познакомим с практическими аспектами разработки, которые необходимы для создания приложения. К ним относятся запросы, определение большой языковой модели (LLM), пользовательский интерфейс и компонент оценки. В последующих главах каждая из этих областей будет рассмотрена еще подробнее. Все это будет сделано с помощью кода, чтобы вы могли связать концептуальную основу, которую мы обсудим, непосредственно с реализацией. Начнем с индексации.

Индексация

Обратите внимание, мы пропускаем настройку, где устанавливаем и импортируем пакеты, а также настраиваем OpenAI и связанные с ним аккаунты. Это типичный шаг в каждом проекте генеративного искусственного интеллекта, а не только в системах RAG. Мы предоставили подробное руководство по настройке в главе 2, так что вернитесь туда, если хотите ознакомиться с библиотеками, которые мы добавили для работы с RAG.

Индексация – первый основной этап RAG. Это шаг после запроса пользователя. В коде из главы 2 индексация — это первый раздел кода. На этом этапе обрабатываются данные, которые вы вводите в систему RAG. Как видно из кода, данные в этом сценарии — это веб-документ, который загружается WebBaseLoader. Вот начало этого документа:

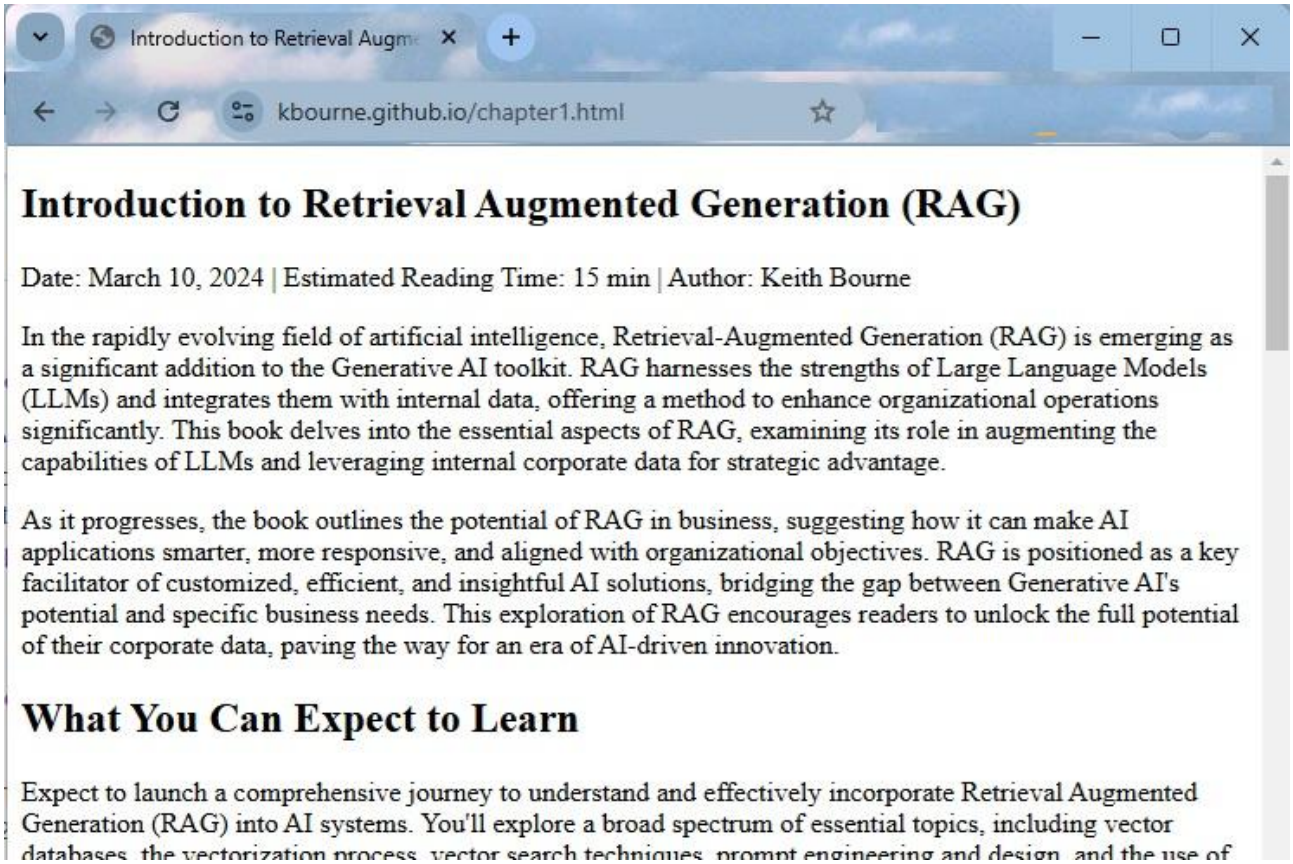

Рис. 4.3. Веб-страница, которую мы будем обрабатывать

В главе 2 вы заметили, что код на этапах извлечения и генерации используется после того, как пользовательский запрос передается в цепочку. Это делается в режиме реального времени, то есть происходит в то время, когда пользователь взаимодействует с системой. Индексирование обычно происходит задолго до того, как пользователь взаимодействует с приложением RAG. Этот этап может позволить себе больше гибкости, поскольку выполняется задолго до использования приложения. Это называется предварительной обработкой в автономном режиме. Иногда индексация может быть выполнена в режиме реального времени, но это встречается реже.

Следующий код извлекает документы:

```
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()
```

Здесь мы загружаем веб-страницу. Но это может быть извлечение данных из документа PDF или Word или других форм неструктурированных данных. Как обсуждалось в главе 3, неструктурированные данные являются очень популярным форматом в приложениях RAG. Исторически сложилось так, что компаниям было очень трудно получить доступ к неструктурированным данным по сравнению со структурированными данными (из баз данных SQL и аналогичных приложений). Но RAG все изменила, и компании, наконец, понимают, как использовать эти данные. В главе 11 мы рассмотрим, как получить доступ к другим типам данных с помощью загрузчиков документов и как это сделать с помощью LangChain.

Независимо от того, какой тип данных вы извлекаете, все они проходят через один и тот же процесс:

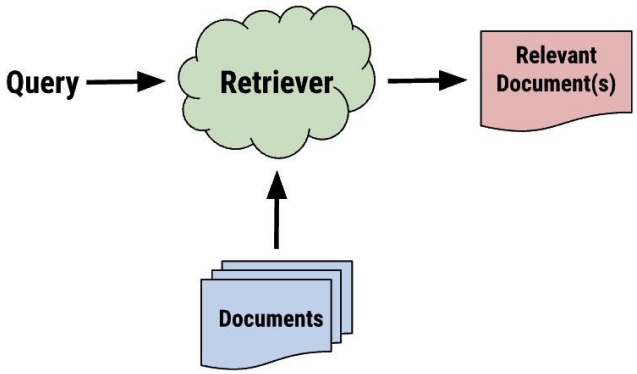

Рис. 4.4. Создание модуля поиска (ретривера, retriever) на этапе индексации процесса RAG

Загрузчик заполняет компонент Documents, чтобы их можно было извлечь позже при пользовательском запросе. В большинстве приложений RAG вы должны преобразовать эти данные в формат удобный для поиска – векторы. Подробнее о векторах мы поговорим чуть позже, но сначала, чтобы перевести данные в векторный формат, необходимо применить разбиение. В нашем коде за это отвечает раздел

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

Сплиттер разбивает содержимое на части, которые можно векторизовать. Разные алгоритмы векторизации имеют разные требования к максимальному размеру содержимого, которое вы можете передать. В данном случае мы используем векторизатор OpenAIEmbeddings(), который в настоящее время имеет максимальный вход 8191 токенов.

В API OpenAI текст токенизируется с помощью словаря байтовой парной кодировки (byte pair encoding, BPE). Это означает, что исходный текст разбивается на подслова, а не на отдельные символы. Количество лексем, используемых для данного входного текста, зависит от конкретного содержимого, так как общие слова и подслова представлены отдельными лексемами, в то время как менее распространенные слова могут быть разбиты на несколько лексем. В среднем один токен равен примерно четырем символам для английского текста. Однако это лишь приблизительная оценка, которая может значительно варьироваться в зависимости от конкретного текста. Например, короткие слова, такие как ***a*** или ***the*** могут быть одной лексемой, в то время как длинное необычное слово может быть разбито на несколько лексем.

Эти удобоваримые блоки должны быть меньше лимита токенов 8191, а другие сервисы встраивания (эмбеддинга) имеют свои лимиты токенов. Если вы используете разделитель, который определяет размер и перекрытие блоков, помните о перекрытии блоков также и для этого ограничения токенов. Вы должны добавить это перекрытие к общему размеру блока, чтобы иметь возможность определить, насколько велик этот кусок. Вот пример использования RecursiveCharacterTextSplitter, где размер блока равен 1000, а перекрытие блока равно 200:

```
text_splitter = RecursiveCharacterTextSplitter(
   chunk_size=1000, chunk_overlap=200)
splits = text_splitter.split_documents(docs)
```

Перекрытие блоков — это распространенный подход, гарантирующий, что контекст не будет потерян между блоками. Например, если в юридическом документе какой-то фрагмент разрезает адрес пополам, маловероятно, что вы найдете этот адрес при его поиске. Но с перекрытием блоков вы можете учесть подобные проблемы. В главе 11 мы рассмотрим различные варианты разделителя, в том числе рекурсивный TextSplitter в LangChain.

Последней частью этапа индексирования является определение векторного хранилища и добавление встраиваемых систем, построенных из ваших разбиений данных, в это векторное хранилище. Вы видите это здесь в этом коде:

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

retriever = vectorstore.as_retriever()

Мы используем Chroma DB в качестве векторной базы данных (или хранилища), передаем ей разбиение и применяем алгоритм встраивания OpenAI. Как и в случае с другими этапами индексирования, все это чаще всего выполняется в автономном режиме до того, как пользователь получит доступ к приложению. Эти векторные вложения хранятся в векторной базе данных для запросов и извлечения в будущем. Chroma DB – это лишь одна из многих баз данных, которые можно здесь использовать. API OpenAIEmbeddings — это лишь один из многих алгоритмов векторизации, которые можно использовать. Мы рассмотрим эту тему подробнее в главах 7 и 8, где будем обсуждать векторы, хранилища векторов и векторный поиск.

Возвращаясь к нашей схеме процесса индексирования, следующие рисунок представляет собой еще более точное представление того, как он выглядит:

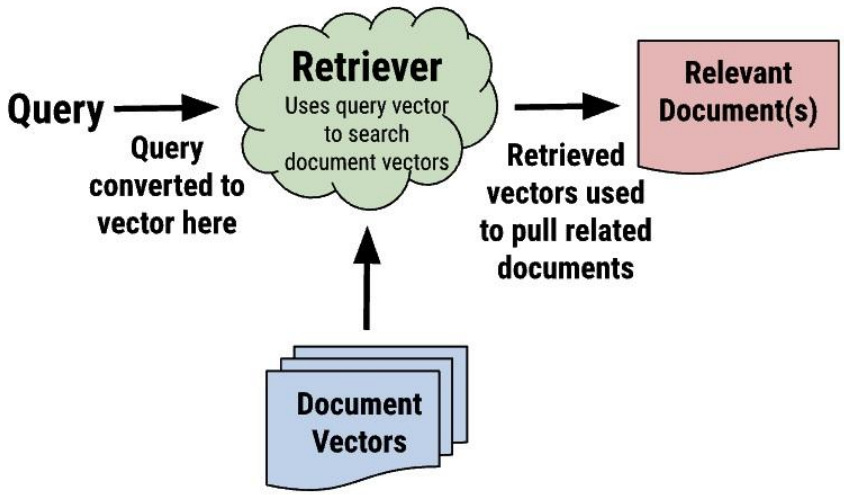

Рис. 4.5. Векторы на этапе индексации процесса RAG

Возможно, вы задаетесь вопросом, почему мы не называем шаг, на котором мы определяем ретривера, частью этапа Извлечения. Это связано с тем, что мы устанавливаем его в качестве механизма, с помощью которого мы извлекаем данные, но не применяем извлечение до тех пор, пока пользователь не выполнит запрос. На этапе индексации внимание уделяется созданию инфраструктуры, с которой работают два других этапа. В конце этой части кода у вас есть ретривер, готовый и ожидающий приема пользовательского запроса.

Извлечение и Генерация

В то время как Извлечение и Генерация являются двумя отдельными этапами, выполняющими две важные функции приложения RAG, в нашем коде они объединены. Когда мы вызываем rag_chain в качестве последнего шага, мы проходим через оба этих этапа, что затрудняет их разделение при обсуждении кода. Но концептуально мы разделим их здесь.

Шаги, ориентированные на извлечение

В полном коде (см. главу 2) есть две области, где обрабатывается Извлечение. Первая:

```
# Post-processing
def format_docs(docs):
   return "\n\n".join(doc.page_content for doc in docs)
```

Вторую можно найти в первом шаге цепочки RAG:

```
   {"context": retriever | format_docs, "question":
RunnablePassthrough()}
```

Когда код инициируется, он выполняется в следующем порядке:

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

Цепочка вызывается с помощью пользовательского запроса и выполняется по шагам:

```
rag_chain = (
```

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

Пользовательский запрос передается в ретривер, который мы определили ранее. Здесь выполняется поиск сходства, чтобы сопоставить пользовательский запрос с данными в векторном хранилище. На выходе получаем список строк контента, который контекстуально похож на запрос пользователя.

Как показано в главе 2, в Извлечения есть небольшой сбой из-за форматирования используемых инструментов. {question} и {context} ожидают строки, но механизм извлечения представляет собой длинный список отдельных строк содержимого. Нам нужен механизм для преобразования этого списка фрагментов содержимого в формат строки.

Ретривер на самом деле находится в мини-цепочке (retriever | format_docs), поэтому вывод ретривера передается в функцию format_docs:

```
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)
```

Данные были получены, но они не в том формате. Функция format_docs завершает задачу и возвращает контент в нужном формате. Это дает нам {context}. Для {question} такой проблемы нет, поскольку вопрос уже является строкой. Таким образом, мы можем использовать удобный объект под названием RunnablePassThrough, который, как следует из его названия, передает входные данные (вопрос) «как есть».

Но подождите минутку. Если вы выполняете векторный поиск, вам нужно преобразовать пользовательский запрос в вектор, верно? Разве мы не говорили, что берем математическое представление пользовательского запроса и измеряем расстояние до других векторов, находя, какие из них ближе? Итак, где это происходит? Ретривер был создан из метода хранилища векторов:

```
retriever = vectorstore.as_retriever()
```

Векторное хранилище, из которого это было сгенерировано, является векторной базой данных Chroma, которая была объявлена с использованием объекта OpenAIEmbeddings() в качестве функции встраивания:

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

Этот метод .as_retriever() имеет всю встроенную функциональность для приема пользовательского запроса, преобразования его в эмбеддинг, который соответствует формату встраивания других эмбеддингов, а затем запуска процесса извлечения.

Поскольку при этом используется объект OpenAIEmbeddings(), он отправляет ваши эмбеддинги в API OpenAI, и вы будете нести расходы за это. В данном случае это всего лишь одно встраивание; с OpenAI в настоящее время это стоит $0,10 за 1 млн токенов. Итак, в чем преимущества использования RAG? вход, который составляет десять токенов по данным OpenAI, будет стоить колоссальные 0,000001 доллара. Может показаться, что это не так уж и много, но мы хотим быть абсолютно прозрачными, когда речь идет о любых расходах!

На этом мы завершаем наш этап Извлечения с выводом, который правильно отформатирован для следующего шага – промта! Далее мы обсудим стадию генерации, на которой используем LLM для выполнения последнего шага генерации ответа.

Генерация

Это заключительный этап, на котором вы используем LLM для генерации ответа на запрос пользователя на основе содержимого, полученного на этапе извлечения. Генерация представлена двумя частями кода. Во-первых, промт:

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

Во-вторых, LLM:

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

После определения промта и LLM в цепочке RAG используются компоненты:

```
  | prompt
  | llm
```

Промтинг

Промты – фундаментальная часть любого приложения генеративного ИИ, а не только RAG. Когда вы начинаете говорить о промтах, особенно в RAG, вы знаете, что вскоре будет использована LLM. Но сначала нужно создать и подготовить правильный запрос для LLM. Теоретически вы могли бы написать свой промт, но я хотел воспользоваться оказией, чтобы показать распространенный шаблон и приучить вас использовать его. В этом примере мы получим промт из LangChain Hub.

LangChain описывает свой хаб как место для «обнаружения, обмена и контроля версий». Другие пользователи поделились здесь своими отполированными промтами. Это хороший способ начать с тщательно разработанных промтов и посмотреть, как они написаны. Позже вы, возможно, захотите перейти к написанию собственных индивидуализированных промтов.

Для чего нужен промт с точки зрения Извлечения. Промт — следующее звено в цепочке после этапа Извлечения. Вы можете увидеть его в rag_chain:

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

Оставаясь верным шаблону LangChain, входные данные промта являются выходами предыдущего шага. Вы можете увидеть эти входные данные в любое время, выведя их на экран:

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

В результате получите выходные данные:

['context', 'question']

Это соответствует тому, что мы определили на предыдущем шаге:

```
  {"context": retriever | format_docs,
   "question": RunnablePassthrough()}
```

Вывод промта с помощью print(prompt) показывает, что существует гораздо больше, чем просто текстовая подсказка и входные переменные:

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={})]

Изучим промт подробнее. Есть список сообщений messages = []. В нашем списке одно сообщение. Оно является экземпляром HumanMessagePromptTemplate – шаблона сообщения. Он инициализируется с помощью объекта PromptTemplate. Объект PromptTemplate создается с

указанной input_variables и строкой шаблона. input_variables это контекст и вопрос, и вы можете видеть, где они расположены в строке шаблона :

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:"

Заполнители {question} и {context} будут заменены фактическими значениями переменных question и context при использовании промта в цепочке. Выходными данными этого звена цепочки является шаблон строки, который был заполнен {question} и {context}. Последняя часть – это *Answer:* Она побуждает LLM к ответу и является распространенным шаблоном.

Итак, промт — это объект, который подключается к цепочке LangChain с входными данными для заполнения шаблона, генерируя промт, который вы передадите в LLM. По сути, это подготовка к этапу Генерации системы RAG.

На следующем этапе мы привлечем мозги LLM, стоящие за всей операцией!

Определение вашего LLM

Выбрав шаблон подсказки, мы можем выбрать LLM, центральный компонент для любого приложения RAG. В следующем коде модель LLM показана в качестве следующего звена цепочки в

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

Как обсуждалось ранее, выходные данные предыдущего шага, которые были объектом *prompt*, будут входными данными следующего шага, LLM. В этом случае промт будет передаваться в LLM вместе с запросом, который мы сгенерировали на предыдущем шаге. Ранее мы определили LLM, которую хотим использовать:

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

Это создание экземпляра класса ChatOpenAI из модуля langchain_openai, который служит интерфейсом для языковых моделей OpenAI, в частности модели GPT-4o. LLM обычно получают промт методом invoke, и вы можете вызвать ее в коде, добавив следующее:

```
llm_only = llm.invoke("Answering in less than 100 words,
   what are the Advantages of using RAG?")
print(llm_only.content)
```

Если вы запустите этот код, он вернет ответ от GPT-4o, который будет знать о RAG. Но для сравнения, что, если мы изменим его на GPT3.5? Вот какой ответ я получил при использовании ChatGPT 3.5:

```
RAG (Red, Amber, Green) status reporting allows for clear and
straightforward communication of project progress or issues. It
helps to quickly identify areas that need attention or improvement,
enabling timely decision-making. RAG status also provides a visual
representation of project health, making it easy for stakeholders to
understand the current situation at a glance. Additionally, using
RAG can help prioritize tasks and resources effectively, increasing
overall project efficiency and success.'
```

Рис. 4.6. Ответ модели LLM ChatGPT 3.5

О-о-о! ChatGPT 3.5 не знает о RAG! По крайней мере, не в том контексте, о котором мы говорим. Это подчеркивает ценность использования RAG для добавления данных. Последней датой отсечения для ChatGPT 3.5 был январь 2022 года. Концепция RAG, ориентированная на генеративный ИИ, должно быть, не была настолько популярна, чтобы модель сразу поняла, что я имею в виду под аббревиатурой RAG.

Используя RAG, мы можем расширить знания модели и использовать другие навыки LLM по обобщению и поиску данных, чтобы получить более успешный результат. Но попробуйте изменить вопрос на «Каковы преимущества использования Retrieval Augmented Generation (RAG)?» и посмотреть, какие результаты получите. Попробуйте использовать более новую модель, которая, скорее всего, содержит больше информации о приложениях RAG в своих обучающих данных. Скорее всего, вы получите лучший ответ, потому что данные, на которых был обучен LLM, имеют более позднюю дату отсечения!

Но вместо того, чтобы обращаться к LLM напрямую, мы передаем ему промт, который структурировали на этапе Извлечения, и можем получить гораздо более интересный ответ. Вы можете завершить цепочку здесь, и результатом вашей цепочки будет ответ LLM. В большинстве случаев это не просто текст, который вы можете увидеть, когда вводите что-то в ChatGPT. Здесь ответ будет в формате JSON и содержит много других данных. Таким образом, если вы хотите получить хорошо отформатированный строковый вывод, отражающий ответ LLM, у вас есть еще одно звено цепи, в которое можно передать ответ LLM: объект StrOutputParser(). Это служебный класс в LangChain, который преобразует вывод языковой модели в строковый формат. Он не только удаляет всю информацию, с которой вы не хотели иметь дело прямо сейчас, но и гарантирует, что сгенерированный ответ будет возвращен в виде строки.

И, конечно же, последняя строка кода — это строка, с которой все начинается:

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

После этапа извлечения этот пользовательский запрос используется во второй раз в качестве одной из входных переменных для промта, который передается в LLM.

Как мы обсуждали в главе 2, в будущем этот запрос будет включать запрос, поступающий из пользовательского интерфейса. Давайте обсудим пользовательский интерфейс как еще одну важную составляющую системы RAG.

Пользовательский интерфейс, UI

В какой-то момент, чтобы сделать приложение более профессиональным и удобным в использовании, вы должны добавить возможность для обычных пользователей, у которых нет вашего кода, вводить запросы напрямую и видеть результаты. UI служит точкой взаимодействия между пользователем и системой и, следовательно, является критически важным компонентом при создании приложения RAG. Расширенные интерфейсы могут включать в себя возможности распознавания естественного языка (natural language understanding, NLU) для более точной интерпретации намерений пользователя, форму обработки естественного языка (natural language processing, NLP), которая фокусируется на понимании части естественного языка. Этот компонент имеет решающее значение для того, чтобы пользователи могли легко и эффективно сообщать системе о своих потребностях.

Для использования UI нам нужно будет изменить код последней строки:

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

Эта строка будет заменена полем ввода, чтобы пользователь мог отправить текстовый вопрос. UI также подразумевает отображение полученного ответа от LLM, например, в красиво оформленном экране. В главе 6 мы покажем это в коде, а пока поговорим о добавлении UI в приложение RAG.

Когда приложение загружается для пользователя, у него будет возможность взаимодействовать с ним. Обычно это облегчается с помощью интерфейса, который варьируется от простых полей ввода текста на веб-странице до более сложных систем распознавания голоса. Ключ в том, чтобы точно зафиксировать запрос пользователя в формате, который может быть обработан системой.

Предварительная обработка

После того как вопрос отправлен, часто происходит предварительная обработка, чтобы сделать этот запрос более удобным для LLM. В первую очередь это делается в промте, который также получает помощь от многих других функций. Но все это происходит за кулисами, а не в поле зрения пользователя. Все, что они увидят — это конечный результат, отображаемый в удобной для пользователя форме.

Постобработка

Даже после того, как LLM вернул ответ, этот ответ часто подвергается постобработке, прежде чем он будет показан пользователю. Вот как выглядит фактический результат LLM:

```
AIMessage(content="The advantages of using RAG include improved
accuracy and relevance of responses generated by large language
models, customization and flexibility in responses tailored to
specific needs, and expanding the model's knowledge beyond the initial
training data.")
```

Рис. 4.7. Ответ LLM

В качестве последнего шага в цепочке мы пропускаем его через StrOutput Parser() для выделения строки:

```
'The advantages of using RAG (Retrieval Augmented Generation) include
improved accuracy and relevance, customization, flexibility, and
expanding the model's knowledge beyond the training data. This means
that RAG can significantly enhance the accuracy and relevance of
responses generated by large language models, tailor responses to
specific needs, and access and utilize information not included
in initial training sets, making the models more versatile and
adaptable.'
```

Рис. 4.8. Ответ LLM в UI

Это, конечно, лучше, чем результат предыдущего шага, но он все еще отображается в вашем блокноте. В более профессиональном приложении вы захотите отобразить это на экране удобным для пользователя способом. Возможно, вы захотите отобразить другую информацию, например исходный документ, который мы показали в коде главы 3. Это зависит от целей вашего приложения и будет различаться в разных системах RAG.

Интерфейс вывода

Эта срока будет передана в UI, который отображает сообщение, возвращенное в цепочку. Интерфейс может быть очень простым, например, как у ChatGPT:

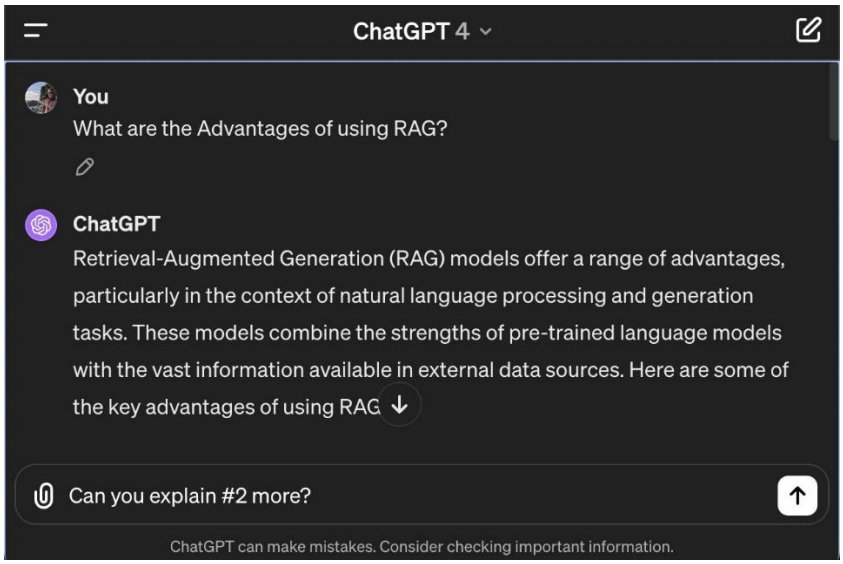

Рис. 4.9. Интерфейс ChatGPT 4

Интерфейс должен быть разработан таким образом, чтобы облегчить дальнейшее взаимодействие. Вы можете предоставить пользователям возможность уточнить свои запросы, задать дополнительные вопросы или запросить дополнительную информацию.

Еще одной общей функцией пользовательского интерфейса является сбор отзывов о полезности и точности ответа. Это может быть использовано для постоянного улучшения производительности системы. Анализируя взаимодействие с пользователем и обратную связь, система может научиться лучше понимать намерения пользователя, совершенствовать процесс векторного поиска и повышать релевантность и качество генерируемых ответов. Это подводит нас к последнему ключевому компоненту: оценке.

Оценка

Оценка имеет важное значение для повышения эффективности системы RAG. Несмотря на то, что существует множество распространенных методов оценки, наиболее эффективная система оценки должна быть сосредоточена на том, что наиболее важно для пользователей, и предоставлять оценку для улучшения функций и возможностей. Часто это включает в себя анализ выходных данных системы с использованием различных показателей, таких как точность, релевантность, время отклика и удовлетворенность пользователей. Эта обратная связь используется для определения областей, требующих улучшения, и внесения изменений в конструкцию системы, обработку данных и интеграцию с LLM. Непрерывная оценка имеет решающее значение для поддержания высокого качества ответов и обеспечения эффективного удовлетворения потребностей пользователей.

Как упоминалось ранее, вы также можете собирать отзывы пользователей различными способами, включая качественные данные (формы ввода заявок с открытыми вопросами) или количественные (верно/неверно, рейтинги или другие числовые представления) о полезности и точности ответа. Большой палец вверх/вниз часто используется для получения быстрой обратной связи от пользователя и оценки общей эффективности приложения среди многих пользователей.

Как включить оценку в ваш код мы рассмотрим в главе 10.