Глава 10. Ключевые компоненты RAG в LangChain

Это продолжение перевода книги Кит Борн. Раскрытие потенциала данных с помощью генеративного ИИ и технологии RAG. В этой главе рассматриваются ключевые технические компоненты, связанные с LangChain и генерацией, дополненной поиском (RAG):

- Хранилища векторов (vector stores).
- Ретриверы (извлекатели, retrievers).
- Большие языковые модели (LLMs).

Мы шаг за шагом рассмотрим версию кода, представленную в главе 8. Мы продемонстрируем различные варианты для каждого из ключевых компонентов с использованием LangChain. Мы поговорим о сценариях, в которых тот или иной вариант может быть предпочтительнее.

Предыдущая глава Содержание Следующая глава

Код для этой главы размещен в репозитории GitHub.

Цель этой главы — показать, как доступные варианты ключевых компонентов LangChain могут улучшить систему RAG. Мы изучим, как работает каждый компонент, какие функции доступны, какие параметры оказывают наибольшее влияние, и в конечном итоге рассмотрим все варианты, которые можно использовать для более качественной реализации RAG. Мы начнем с практикума, посвященного выбору хранилища векторов.

Лаборатория кода 10.1. Хранилище векторов в LangChain

Загрузите код практикума 8.3 (пропускаем код главы 9, посвященной оценке). Мы будем разбирать элементы в порядке их появления в коде. Финальный код для этого практикума.

Векторные хранилища, LangChain и RAG

Векторные хранилища играют ключевую роль в системах Генерации, дополненной поиском (RAG), эффективно сохраняя и индексируя векторные представления документов базы знаний. LangChain обеспечивает бесшовную интеграцию с различными векторными хранилищами, такими как Chroma, Weaviate, FAISS (Facebook AI Similarity Search), pgvector и Pinecone. В этой лабораторной работе мы покажем код для добавления данных в Chroma, Weaviate и FAISS, заложив основу для интеграции любого векторного хранилища из множества доступных в LangChain.

Класс векторного хранилища в LangChain служит унифицированным интерфейсом для работы с различными хранилищами. Он предоставляет методы для добавления документов, выполнения поиска по сходству и извлечения сохраненных документов. Такая абстракция позволяет разработчикам легко переключаться между реализациями векторных хранилищ, не изменяя основную логику поиска.

Выбор конкретного хранилища зависит от таких факторов, как масштабируемость, производительность поиска и требования к развертыванию. Например, Pinecone предлагает полностью управляемую векторную базу данных с высокой масштабируемостью и возможностями поиска в реальном времени, что делает его подходящим для промышленных RAG-систем. FAISS, в свою очередь, представляет собой открытый программный продукт для эффективного поиска по сходству, который можно использовать для локальной разработки и экспериментов. Chroma — популярный выбор для разработчиков, начинающих создавать свои первые конвейеры RAG, благодаря простоте использования и эффективной интеграции с LangChain.

Если посмотреть на код, обсуждавшийся в предыдущих главах, мы уже используем Chroma. Вот фрагмент кода, демонстрирующий его использование:

```
chroma_client = chromadb.Client()
collection_name = "google_environmental_report"
vectorstore = Chroma.from_documents(
   documents=dense_documents,
   embedding=embedding_function,
   collection_name=collection_name,
   client=chroma_client
)
```

LangChain называет это интеграцией, поскольку он взаимодействует со сторонним сервисом Chroma. Существует множество других интеграций с LangChain.

На текущий момент на веб-сайте *LangChain* в меню *Integrations* можно просматривать интеграции по поставщикам или компонентам. Список поставщиков Providers отсортирован в алфавитном порядке. Прокрутив его, можно найти *Chroma* и ознакомиться с полезной документацией LangChain, особенно по созданию векторного хранилища и поискового модуля.

Еще один способ – нажать на *Vector stores* в разделе Components. В данный момент доступно 49 вариантов векторных хранилищ.[1] Актуальная ссылка для версии 0.2.0.

Мы ранее обсуждали векторные хранилища, но давайте еще раз рассмотрим Chroma и разберемся, в каких случаях оно наиболее полезно.

Chroma

Chroma — это открытая, ориентированная на ИИ векторная база данных, созданная для удобства разработчиков. Она обеспечивает высокую скорость поиска и бесшовную интеграцию с LangChain через Python SDK. Chroma поддерживает различные режимы развертывания, включая хранение в памяти, постоянное хранилище и контейнеризацию с использованием Docker.

Одним из ключевых преимуществ Chroma является простота использования и удобный API. Он предоставляет понятные методы для добавления, обновления, удаления и запроса документов в векторном хранилище. Chroma также поддерживает динамическую фильтрацию коллекций на основе метаданных, что позволяет выполнять более точечные поисковые запросы. Кроме того, в Chroma встроены функции разбиения документов на фрагменты и индексирования, что упрощает работу с большими текстовыми наборами данных.

При выборе Chroma в качестве векторного хранилища для приложения RAG важно учитывать его архитектуру и критерии отбора. Архитектура Chroma включает слой индексации для быстрого извлечения векторов, слой хранения для эффективного управления данными и слой обработки для выполнения операций в реальном времени. Chroma интегрируется с LangChain, позволяя разработчикам использовать его возможности в рамках этой экосистемы. Клиент Chroma легко инициализируется и передается в LangChain, обеспечивая эффективное хранение и извлечение векторных представлений документов. Chroma также поддерживает расширенные методы поиска, такие как максимальная предельная релевантность (MMR) и фильтрация по метаданным для более точных результатов.

В целом, Chroma — это отличный выбор для разработчиков, которым требуется простая в использовании, производительная и интегрируемая с LangChain векторная база данных. Простота, высокая скорость поиска и встроенная обработка документов делают Chroma привлекательным вариантом для создания RAG-приложений. Именно поэтому мы выбрали Chroma для рассмотрения в нескольких главах этой книги. Однако важно учитывать специфику вашего проекта и сравнивать Chroma с другими векторными хранилищами, чтобы выбрать наиболее подходящий вариант.

FAISS

Теперь давайте рассмотрим, как изменится наш код, если мы захотим использовать FAISS в качестве векторного хранилища. Для начала необходимо установить FAISS:

```
%pip install faiss-cpu
```

После установки и перезапуска ядра (так как был добавлен новый пакет), выполните весь код до ячейки, связанной с векторным хранилищем, и замените код, относящийся к Chroma, на инициализацию векторного хранилища FAISS:

```
from langchain_community.vectorstores import FAISS
vectorstore = FAISS.from_documents(
   documents=dense_documents,
   embedding=embedding_function
)
```

[1] На 09.02.2025 доступно 112 векторных хранилищ.

Вызов метода *Chroma.from_documents()* заменен на *FAISS.from_documents()*. Параметры *collection_name* и *client* не применимы к FAISS, поэтому они удалены из вызова метода. Мы повторяем часть кода, использованного для векторного хранилища *Chroma*, включая генерацию документов, чтобы показать точный эквивалент кода для обоих вариантов хранилища. После этих изменений код теперь использует *FAISS* вместо *Chroma*.

FAISS — это библиотека с открытым исходным кодом, разработанная *Facebook AI*. Она предлагает высокопроизводительный поиск и способна обрабатывать большие наборы данных, которые могут не помещаться полностью в память. Как и другие векторные хранилища, *FAISS* включает слой индексации, организующий векторы для быстрого поиска, слой хранения для эффективного управления данными и необязательный слой обработки для операций в реальном времени. *FAISS* поддерживает различные методы индексации, такие как кластеризация и квантизация, которые оптимизируют производительность поиска и использование памяти. Кроме того, *FAISS* может использовать ускорение на *GPU* для еще более быстрого поиска по сходству.

Если у вас есть доступ к *GPU*, вместо ранее установленного пакета можно установить:

%pip install faiss-gpu

Использование GPU-версии *FAISS* значительно ускоряет процесс поиска по сходству, особенно для крупных наборов данных. GPU позволяют обрабатывать большое количество векторных сравнений параллельно, обеспечивая более быстрое извлечение релевантных документов в приложении RAG. Если ваша среда обрабатывает огромные объемы данных и требует значительного прироста производительности по сравнению с Chroma, обязательно протестируйте FAISS GPU, чтобы оценить его влияние.

Документация FAISS для LangChain содержит подробные примеры и руководства по использованию FAISS в рамках LangChain. Она охватывает такие темы, как загрузка документов, выполнение запросов к векторному хранилищу, сохранение и загрузка индексов, а также выполнение расширенных операций, таких как фильтрация и объединение. В документации также рассматриваются специфические функции FAISS, включая поиск по сходству с оценками релевантности и сериализацию/десериализацию индексов.[2]

В целом, FAISS – это мощное и эффективное решение для хранения векторов в приложениях RAG с использованием LangChain. Высокая производительность поиска, масштабируемость и бесшовная интеграция с LangChain делают FAISS отличным выбором для разработчиков, которым требуется надежное и настраиваемое векторное хранилище.

Weaviate

Существует несколько способов использования и доступа к Weaviate. Мы рассмотрим вариант встраиваемого (embedded) использования, при котором экземпляр Weaviate запускается непосредственно из кода приложения, а не из отдельного сервера Weaviate.

При первом запуске Embedded Weaviate создает постоянное хранилище данных в указанном *persistence_data_path*. Когда клиент завершает работу, экземпляр Embedded Weaviate также завершается, но данные сохраняются. При следующем запуске клиента создается новый экземпляр Embedded Weaviate, который использует ранее сохраненные данные.

Если вы знакомы с GraphQL, то, изучая код Weaviate, можете заметить его влияние. Язык запросов и API Weaviate вдохновлены GraphQL, но при этом Weaviate не использует GraphQL напрямую. Вместо

---

[2] ChatGOT пояснил. Сериализация и десериализация — это процессы преобразования данных между их представлением в памяти и форматом, который можно сохранить или передать.

- **Сериализация** — это преобразование объекта (например, индекса FAISS) в формат, который можно записать в файл, базу данных или передать по сети (например, JSON, бинарный файл, pickle).
- **Десериализация** — это обратный процесс, когда сохраненные данные загружаются обратно в оперативную память в виде объекта, с которым можно работать.

В контексте FAISS:

- Сериализация позволяет сохранить индекс в файл, чтобы потом его не пересоздавать заново.
- Десериализация загружает этот индекс обратно, ускоряя запуск системы, поскольку нет необходимости пересчитывать векторные представления и строить индексы заново.

этого он работает через RESTful API с языком запросов, который по своей структуре и функциональности напоминает GraphQL.

Weaviate использует предопределенные типы данных для свойств в определении схемы, аналогично скалярным типам GraphQL. Доступные типы данных в Weaviate включают string, int, number, Boolean, date и другие.

Одним из преимуществ Weaviate является поддержка пакетных (batch) операций для создания, обновления или удаления нескольких объектов данных в одном запросе. Это похоже на операции mutation в GraphQL, позволяющие вносить несколько изменений за один вызов. В Weaviate используется контекстный менеджер *client.batch* для группировки нескольких операций в один пакет, что мы продемонстрируем далее.

Теперь рассмотрим, как изменится наш код, если мы захотим использовать Weaviate в качестве векторного хранилища. Сначала необходимо установить Weaviate и сопутствующие зависимости.

```
%pip install weaviate-client
%pip install langchain-weaviate
```

После установки и перезапуска ядра (так как добавлен новый пакет) выполните весь код до ячейки, связанной с векторным хранилищем, и обновите его, заменив инициализацию Chroma или FAISS на Weaviate.

```
import weaviate
from langchain_weaviate.vectorstores import WeaviateVectorStore
from weaviate.embedded import EmbeddedOptions
from langchain.vectorstores import Weaviate
from tqdm import tqdm
```

Для работы с Weaviate потребуется импортировать дополнительные пакеты. Также устанавливается *tqdm*, который не является специфичным для Weaviate, но необходим, поскольку Weaviate использует его для отображения индикаторов выполнения при загрузке данных.

Сначала необходимо объявить weaviate_client как клиент Weaviate:

```
weaviate_client = weaviate.Client(
  embedded_options=EmbeddedOptions())
```

Отличия между нашим оригинальным кодом для Chroma и использованием Weaviate сложнее, чем в других рассмотренных случаях. В Weaviate мы инициализируем клиент WeaviateClient и задаем параметры встраивания (embedding), чтобы активировать встроенный режим, как было показано ранее.

Перед продолжением необходимо убедиться, что не запущен предыдущий экземпляр клиента Weaviate, иначе код завершится с ошибкой:

```
try:
  weaviate_client.schema.delete_class(collection_name)
except:
  pass
```

В Weaviate также важно удалить все старые схемы данных, так как они могут сохраняться в фоновом режиме и приводить к конфликтам.

Далее с помощью клиента Weaviate создается база данных, используя определение схемы, похожее на GraphQL:

```
# Create the 'Google_environmental_report' class in the schema
weaviate_client.schema.create_class({
  "class": collection_name,
  "description": "Google Environmental Report",
  "properties": [
    {
      "name": "text",
```

```
        "dataType": ["text"],
        "description": "Text content of the document"
      },
      {
        "name": "doc_id",
        "dataType": ["string"],
        "description": "Document ID"
      },
      {
        "name": "source",
        "dataType": ["string"],
        "description": "Document source"
      }
    ]
})
```

Этот этап включает создание полного класса схемы, который затем передается в определение векторного хранилища в составе объекта weaviate_client. Вам необходимо определить эту схему для вашей коллекции с помощью метода client.collections.create().

Определение схемы включает указание имени класса, свойств и их типов данных. Свойства могут иметь различные типы данных, такие как string, integer и Boolean. Как можно заметить, Weaviate применяет более строгую проверку схемы по сравнению с тем, что мы использовали ранее в Chroma.

Хотя такая GraphQL-подобная схема усложняет настройку векторного хранилища, она также предоставляет больше возможностей для контроля базы данных. В частности, Weaviate позволяет более точно определять структуру схемы.

Далее идет фрагмент кода, который выглядит похоже на переменные dense_documents и sparse_documents, которые мы использовали ранее. Однако, если присмотреться, есть одно важное изменение, специфичное для Weaviate:

```
# Create new versions of these with "doc_id"
# because Weaviate reserves the "id" in the metadata for their internal id
dense_documents = [Document(page_content=text, metadata=
  {"doc_id": str(i), "source": "dense"}) for i, text in enumerate(splits)]
sparse_documents = [Document(page_content=text, metadata=
  {"doc_id": str(i), "source": "sparse"}) for i, text in enumerate(splits)]
```

При предварительной обработке документов с метаданными для Weaviate используется doc_id вместо id. Это связано с тем, что id зарезервирован для внутреннего использования и недоступен для пользователя. Позже в коде, при извлечении идентификатора из результатов метаданных, также необходимо обновить соответствующий участок, чтобы использовать doc_id.

Следующим шагом мы определяем векторное хранилище, аналогично тому, что мы делали ранее с Chroma и FAISS, но с параметрами, специфичными для Weaviate:

```
vectorstore = Weaviate(
  client=weaviate_client,
  embedding=embedding_function,
  index_name=collection_name,
  text_key="text",
  attributes=["doc_id", "source"],
  by_text=False
)
```

При инициализации векторного хранилища Chroma использует метод from_documents, который позволяет создать хранилище непосредственно из документов. В отличие от этого, в Weaviate сначала создается векторное хранилище, а затем в него добавляются документы.

Кроме того, Weaviate требует дополнительной конфигурации, такой как text_key, attributes и by_text. Одним из ключевых отличий Weaviate является его обязательное использование схемы данных.

На заключительном этапе загружается экземпляр векторного хранилища Weaviate с фактическим содержимым, при этом также применяется функция эмбеддинга.

Итоги сравнения

- **Chroma** предлагает более простой и гибкий подход к определению схемы данных, ориентируясь на хранение и извлечение эмбеддингов. Он легко встраивается в приложение.
- **Weaviate** предоставляет более структурированное и функционально богатое решение для работы с векторами, включая явное определение схемы, поддержку нескольких типов хранилищ и встроенную работу с различными моделями эмбеддингов. Он может быть развернут как отдельный сервер или в облаке.

Выбор между Chroma, Weaviate или другими векторными хранилищами зависит от ваших требований:

- уровень гибкости в схеме данных,
- предпочтения в развертывании,
- необходимость дополнительных функций помимо хранения эмбеддингов.

Важно отметить, что независимо от выбранного векторного хранилища, остальная часть кода будет работать с загруженными в него данными. Это одно из преимуществ LangChain — возможность легко заменять компоненты.

В мире генеративного ИИ, где постоянно появляются новые и улучшенные технологии, такой подход особенно важен. Если вы обнаружите более эффективное векторное хранилище, его можно быстро и без значительных изменений интегрировать в ваш конвейер RAG.

Теперь рассмотрим еще один ключевой компонент LangChain, который играет центральную роль в приложениях RAG — **ретривер**.

Лаборатория кода 10.2. Ретриверы LangChain

Мы рассмотрим несколько примеров самого важного компонента в процессе поиска: ретривера LangChain. Как и в случае с векторными хранилищами LangChain, существует много вариантов ретриверов. Мы сосредоточимся на нескольких популярных вариантах, которые особенно применимы к RAG-приложениям, и рекомендуем вам изучить остальные, чтобы найти наилучшее решение для вашей конкретной ситуации. Точно так же, как мы обсуждали векторные хранилища, на сайте LangChain есть подробная документация, которая поможет вам выбрать оптимальное решение. Документация для пакета ретриверов доступна здесь.

Теперь давайте перейдем к написанию кода для ретриверов!

Ретриверы, LangChain и RAG

Ретриверы отвечают за выполнение запросов к векторному хранилищу и извлечение наиболее релевантных документов на основе входного запроса. LangChain предлагает широкий выбор реализаций ретриверов, которые можно использовать вместе с различными векторными хранилищами и кодировщиками запросов.

В нашем коде до этого момента мы уже рассмотрели три варианта ретривера. Давайте сначала их разберем, так как они относятся к исходному векторному хранилищу на основе Chroma.

Базовый ретривер (плотные эмбеддинги)

Начнем с плотного ретривера. Этот код мы использовали в нескольких предыдущих практических занятиях:

```
dense_retriever = vectorstore.as_retriever(search_kwargs={"k": 10})
```

Плотный ретривер создается с помощью функции vectorstore.as_retriever, указывая количество извлекаемых результатов (k=10). Под капотом этого ретривера Chroma использует плотные векторные представления документов и выполняет поиск по сходству, используя косинусное расстояние или евклидово расстояние для поиска наиболее релевантных документов на основе эмбеддинга запроса.

Этот ретривер относится к самому простому типу — ретриверу векторного хранилища. Он просто создает эмбеддинги для каждого фрагмента текста и использует их для поиска. Фактически, ретривер представляет собой обертку вокруг векторного хранилища. Этот подход дает доступ к встроенному функционалу поиска/извлечения векторного хранилища, но в таком виде, который интегрируется в экосистему LangChain. Это легковесная обертка вокруг класса векторного хранилища, которая предоставляет единый интерфейс для всех ретриверов в LangChain. Благодаря этому после создания векторного хранилища очень легко создать ретривер. Если вам нужно сменить векторное хранилище или ретривер, это также можно сделать без труда.

Существует два основных метода поиска, доступные в этих ретриверах, поскольку они напрямую зависят от используемого векторного хранилища: поиск по сходству и MMR.

Извлечение по пороговому значению оценки сходства

По умолчанию ретриверы используют поиск по сходству. Однако, если вам нужно установить пороговое значение сходства, достаточно установить тип поиска в similarity_score_threshold и задать пороговое значение сходства в параметрах kwargs, передаваемых объекту ретривера. Код выглядит следующим образом:

```
# adding similarity score threshold
dense_retriever = vectorstore.as_retriever(
   search_type="similarity_score_threshold",
   search_kwargs={"score_threshold": 0.5}
)
```

Это полезное улучшение стандартного поиска по сходству, которое может быть полезно во многих RAG-приложениях. Однако поиск по сходству — не единственный тип поиска, который поддерживают эти ретриверы; также существует MMR.

MMR

MMR (Maximal Marginal Relevance) — это техника, используемая для извлечения релевантных элементов запроса при одновременном предотвращении избыточности. Она балансирует релевантность и разнообразие извлекаемых элементов, в отличие от стандартного поиска, который просто возвращает самые релевантные элементы, часто схожие друг с другом. MMR широко используется в информационном поиске и может применяться для суммаризации документов, вычисляя сходство между частями текста.

Чтобы настроить ретривер для использования этого типа поиска вместо поиска по сходству, необходимо добавить search_type="mmr" в параметры при определении ретривера:

```
# Searching with Maximum marginal relevance (MMR)
dense_retriever = vectorstore.as_retriever(
   search_type="mmr"
)
```

Добавление этого параметра к любому ретриверу на основе векторного хранилища приведет к использованию MMR-поиска.

Поиск по сходству и MMR могут поддерживаться любыми векторными хранилищами, которые поддерживают соответствующие методы поиска. Далее рассмотрим механизм разреженного поиска, который мы представили в главе 8, — ретривер BM25.

Ретривер BM25

BM25 — это функция ранжирования, используемая для разреженного текстового поиска. BM25Retriever — это реализация BM25 в LangChain, предназначенная для задач разреженного поиска. Этот ретривер мы уже использовали в качестве компонента гибридного поиска в главе 8. В коде он представлен следующими настройками:

```
sparse_retriever = BM25Retriever.from_documents(sparse_documents, k=10)
```

Метод BM25Retriever.from_documents() вызывается для создания разреженного ретривера на основе разреженных документов, с указанием количества извлекаемых результатов (k=10). BM25 работает, вычисляя оценку релевантности для каждого документа на основе терминов запроса, частоты терминов в документе и их обратной частоты (TF-IDF). Он использует вероятностную модель для

оценки релевантности документов заданному запросу. Ретривер возвращает top-k документов с наивысшими BM25-оценками.

Комибинорованный ретривер

Комбинированный ретривер объединяет несколько методов поиска и использует дополнительный алгоритм для комбинирования их результатов в единый набор. Идеальный случай использования такого ретривера — объединение плотного и разреженного ретриверов для поддержки гибридного поиска, подобного тому, который мы создали в коде главы 8 (Code lab 8.3):

```
ensemble_retriever = EnsembleRetriever(
   retrievers=[dense_retriever, sparse_retriever],
   weights=[0.5, 0.5], c=0, k=10
)
```

В данном случае комбинированный ретривер объединяет плотный ретривер Chroma и разреженный ретривер BM25 для достижения лучшей производительности поиска. Он создается с помощью класса EnsembleRetriever, который принимает список ретриверов и соответствующие им веса. В этом примере плотный и разреженный ретриверы передаются с равными весами 0.5 для каждого.

Параметр c в комбинированном ретривере — это параметр повторного ранжирования, который управляет балансом между исходными оценками поиска и оценками повторного ранжирования. Он используется для регулирования влияния этапа повторного ранжирования на итоговые результаты поиска. В данном случае параметр *c* = 0, что означает, что повторное ранжирование не выполняется. Если *c* устанавливается в ненулевое значение, комбинированный ретривер выполняет дополнительный этап повторного ранжирования найденных документов. Этот этап пересчитывает оценки найденных документов на основе отдельной модели или функции повторного ранжирования. Такая модель может учитывать дополнительные признаки или критерии для оценки релевантности документов по отношению к запросу.

В RAG-приложениях качество и релевантность найденных документов напрямую влияют на сгенерированный результат. Используя параметр *c* и подходящую модель повторного ранжирования, можно улучшить результаты поиска, адаптируя их под конкретные требования RAG-приложения. Например, можно разработать модель повторного ранжирования, которая учитывает такие факторы, как релевантность документа, его согласованность с запросом или специфические для домена критерии. Выбирая подходящее значение *c*, можно установить баланс между исходными оценками поиска и оценками повторного ранжирования, увеличивая вес модели повторного ранжирования при необходимости. Это позволяет отдавать приоритет более релевантным и информативным документам, что улучшает качество итогового результата генерации.

Когда запрос передается в комбинированный ретривер, он отправляет его как плотному, так и разреженному ретриверам. Затем комбинированный ретривер объединяет результаты обоих ретриверов, учитывая их веса, и возвращает top-k документов. Под капотом комбинированный ретривер использует сильные стороны обоих методов поиска. Плотный поиск выявляет семантическое сходство с помощью плотных векторных представлений, тогда как разреженный поиск основан на совпадении ключевых слов и частотах терминов. Объединяя их результаты, комбинированный ретривер стремится предоставить более точные и всесторонние результаты поиска.

Конкретные классы и методы, используемые в коде, могут различаться в зависимости от используемой библиотеки или фреймворка. Однако общие концепции — плотный поиск с использованием векторного поиска по сходству, разреженный поиск с BM25 и комбинированный поиск с несколькими ретриверами — остаются неизменными. Это охватывает ретриверы, которые мы уже рассмотрели в предыдущем коде, все они работают с данными, к которым мы получили доступ и обработали на этапе индексации. Существует множество других типов ретриверов, работающих с данными, извлеченными из документов, которые можно изучить на сайте LangChain, чтобы выбрать подходящий вариант. Однако не все ретриверы предназначены для работы с документами, которые вы обрабатываете. Далее мы рассмотрим пример ретривера, построенного на основе открытого источника данных — Wikipedia.

Ретривер Wikipedia

Как описано создателями ретривера Wikipedia на сайте LangChain: Wikipedia — это самая крупная и наиболее читаемая справочная работа в истории, представляющая собой многоязычную бесплатную онлайн-энциклопедию, написанную и поддерживаемую сообществом волонтеров. Wikipedia – отличный источник полезных знаний в ваших RAG-приложениях! Мы добавим новую ячейку после существующей ячейки ретривера, где будем использовать этот ретривер Wikipedia для получения страниц из wikipedia.org в формате Document, который затем будет использоваться в дальнейшем процессе.

Сначала нам нужно установить несколько новых пакетов:

```
# New installations for supporting new retrievers
%pip install langchain-core==0.3.6
%pip install --upgrade --quiet wikipedia==1.4.0
```

Как всегда, не забудьте перезапустить ядро после установки новых пакетов!

С помощью ретривера WikipediaRetriever у нас теперь есть механизм, который может извлекать данные из Wikipedia в соответствии с пользовательским запросом, аналогично другим ретриверам, которые мы использовали, но с доступом ко всему массиву данных Wikipedia.

```
from langchain_community.retrievers import WikipediaRetriever
retriever = WikipediaRetriever(load_max_docs=10)
docs = retriever.get_relevant_documents(
   query="What defines the golden age of piracy in the Caribbean?")
metadata_title = docs[0].metadata['title']
metadata_summary = docs[0].metadata['summary']
metadata_source = docs[0].metadata['source']
page_content = docs[0].page_content
print(f"First document returned:\n")
print(f"Title: {metadata_title}\n")
print(f"Summary: {metadata_summary}\n")
print(f"Source: {metadata_source}\n")
print(f"Page content:\n\n{page_content}\n")
```

В этом коде мы импортируем класс WikipediaRetriever из модуля langchain_community.retrievers. WikipediaRetriever — это класс ретривера, специально разработанный для поиска релевантных документов в Wikipedia на основе заданного запроса. Затем мы создаем экземпляр этого ретривера, используя класс WikipediaRetriever, и присваиваем его переменной retriever. Параметр load_max_docs установлен в 10, что означает, что ретривер должен загружать не более 10 релевантных документов. В данном случае пользовательский запрос — "Что определяет золотой век пиратства в Карибском море?", и мы можем посмотреть на полученный ответ, чтобы увидеть, какие статьи Wikipedia были найдены для ответа на этот вопрос.

Мы вызываем метод get_relevant_documents объекта retriever, передавая в качестве аргумента строку запроса, и получаем в ответе первый документ:

```
First document returned:
Title: Golden Age of Piracy
Summary: The Golden Age of Piracy is a common designation for the
period between the 1650s and the 1730s, when maritime piracy was a
significant factor in the histories of the North Atlantic and Indian
Oceans.
Histories of piracy often subdivide the Golden Age of Piracy into
three periods:
The buccaneering period (approximately 1650 to 1680)…
```

Соответствующий контент можно найти по ссылке: https://en.wikipedia.org/wiki/Golden_Age_of_Piracy предоставленной ретривером в качестве источника. Этот код демонстрирует, как использовать класс WikipediaRetriever из модуля langchain_community.retrievers для поиска релевантных документов в

Wikipedia по заданному запросу. Затем он извлекает и выводит определенную метаинформацию (заголовок, краткое описание, источник) и содержимое первого найденного документа.

WikipediaRetriever автоматически обрабатывает процесс запроса к API Wikipedia или использует функциональность поиска, извлекает релевантные документы и возвращает их в виде списка объектов Document. Каждый объект Document содержит метаинформацию и фактическое содержимое страницы, к которому можно получить доступ и использовать по необходимости.

Существуют и другие ретриверы, которые могут обращаться к публичным источникам данных, но ориентированы на конкретные области. Например, для научных исследований существует PubMedRetriever. Для других исследовательских областей, таких как математика и информатика, есть ArxivRetriever, который получает данные из открытого архива более 2 миллионов научных статей по этим темам. В финансовой сфере существует ретривер KayAiRetriever, который может получать доступ к отчетности Комиссии по ценным бумагам и биржам США (SEC), содержащей финансовые отчеты публичных компаний.

Для проектов, работающих с данными меньшего масштаба, у нас есть еще один ретривер, который стоит рассмотреть: kNN-ретривер.

kNN-ретривер

Алгоритмы ближайших соседей, с которыми мы работали до этого момента и которые отвечают за поиск наиболее связанного с пользовательским запросом контента, были основаны на методе приближенного ближайшего соседа (ANN). Однако существует более традиционный и старый алгоритм, который является альтернативой ANN, — это метод k-ближайших соседей (kNN). Но ведь kNN основан на алгоритме, разработанном еще в 1951 году; почему мы должны его использовать, если у нас есть более сложный и мощный алгоритм, такой как ANN? Потому что kNN по-прежнему лучше всего справляется с задачей поиска ближайших соседей. Это не ошибка. kNN по-прежнему является наиболее эффективным методом поиска ближайших соседей. Он превосходит ANN, который рекламируется как лучшее решение всеми компаниями, работающими в сфере баз данных, векторных баз данных и информационного поиска. ANN может давать близкие результаты, но kNN все же считается лучшим.

Почему же тогда ANN называют оптимальным решением? Потому что kNN не масштабируется до уровня, необходимого крупным предприятиям, на которые ориентированы поставщики этих решений. Однако это все относительно. Возможно, у вас есть миллион точек данных, что звучит внушительно, с векторами размерностью 1 536, но даже это считается небольшим объемом в масштабах глобального предприятия. kNN справляется с таким объемом без проблем! Многие небольшие проекты, использующие ANN, на самом деле могут получить больше пользы от kNN.

Теоретический предел kNN зависит от множества факторов, таких как среда разработки, данные, размерность векторов, подключение к интернету (если используются API) и многое другое. Поэтому нельзя назвать точное количество обрабатываемых точек данных. Это нужно тестировать. Однако если ваш объем данных меньше описанного выше проекта (1 миллион точек данных с векторами размерностью 1 536) и у вас достаточно мощная среда разработки, вам стоит всерьез рассмотреть использование kNN! В какой-то момент вы заметите значительное увеличение времени обработки, и когда ожидание станет слишком долгим для работы вашего приложения, тогда можно переключиться на ANN. Но до этого момента обязательно воспользуйтесь преимуществами более точного поиска, который предлагает kNN.

К счастью, kNN доступен в виде простого в настройке ретривера под названием KNNRetriever. Этот ретривер использует те же плотные эмбеддинги, что и наши другие алгоритмы, поэтому мы просто заменим dense_retriever на основанный на kNN KNNRetriever. Вот код для его реализации:

```
# Use kNN for dense retriever
from langchain_community.retrievers import KNNRetriever
dense_retriever = KNNRetriever.from_texts(splits, embedding_function, k=10)
# Re-initialize the ensemble retriever
ensemble_retriever = EnsembleRetriever(
   retrievers=[dense_retriever, sparse_retriever], weights=[0.5, 0.5], c=0, k=10)
```

Запустите оставшийся код в лабораторной работе, чтобы увидеть, как он заменяет наш предыдущий dense_retriever и выполняет его функции. В данной ситуации, с очень ограниченным набором данных, сложно оценить, работает ли он лучше, чем алгоритм на основе ANN, который мы использовали ранее. Однако по мере масштабирования проекта мы настоятельно рекомендуем использовать этот подход, пока проблемы масштабируемости не станут слишком серьезными.

На этом завершается наше изучение ретриверов, поддерживающих RAG. Существуют дополнительные типы ретриверов, а также важные интеграции с векторными хранилищами, поддерживающими эти ретриверы, которые можно изучить на сайте LangChain. Например, есть ретривер с временным взвешиванием, который позволяет учитывать актуальность данных при поиске. Также существует ретривер Long-Context Reorder, предназначенный для улучшения работы моделей с длинным контекстом, которые испытывают трудности с фокусировкой на информации, расположенной в середине найденных документов. Обязательно изучите доступные варианты, так как они могут значительно повлиять на эффективность вашего RAG-приложения.

Теперь мы перейдем к обсуждению "мозга" всей системы и этапа генерации: LLM.

Лабораторная работа 10.3 – LangChain LLMs

Теперь мы сосредоточимся на последнем ключевом компоненте RAG: LLM. Как и ретривер на этапе поиска, без LLM на этапе генерации RAG невозможен. Этап поиска просто извлекает данные из источника, обычно из того, о чем LLM не имеет предварительных знаний. Однако это не означает, что LLM не играет важной роли в реализации RAG. Предоставляя найденные данные модели, мы быстро вводим ее в контекст, позволяя ей делать то, что у нее получается лучше всего — формировать обоснованный и связный результат на основе этих данных, отвечая на исходный пользовательский запрос.

Синергия между LLM и системами RAG объясняется тем, что они дополняют друг друга. RAG расширяет возможности LLM, интегрируя внешние источники знаний, что позволяет генерировать не только контекстно релевантные, но и фактически точные и актуальные ответы. В свою очередь, LLM улучшает работу RAG, обеспечивая глубокое понимание контекста запроса, что способствует более эффективному извлечению информации из базы знаний. Такое взаимодействие значительно повышает производительность ИИ-систем в задачах, требующих как глубокой языковой обработки, так и доступа к актуальной информации, объединяя сильные стороны каждого компонента для создания более мощной и универсальной системы.

В этой лабораторной работе мы рассмотрим несколько примеров ключевого компонента этапа генерации — LangChain LLM.

LLM, LangChain и RAG

Как и в случае с предыдущими ключевыми компонентами, сначала приведем ссылки на документацию LangChain, относящуюся к этому важному элементу — LLM. Еще один полезный источник информации по объединению LLM с LangChain — API-документация. Начнем с API, которое мы уже использовали ранее: OpenAI.

OpenAI

Этот код у нас уже есть, но давайте обновим понимание его работы, разобрав ключевые элементы лабораторной работы, которые обеспечивают работу этого компонента. Сначала необходимо установить пакет langchain-openai:

```
%pip install langchain-openai==0.2.1
```

Библиотека langchain-openai предоставляет интеграцию между языковыми моделями OpenAI и LangChain.

Далее мы импортируем библиотеку openai, которая является официальной Python-библиотекой для работы с API OpenAI. В этом коде она используется в основном для передачи API-ключа в модель, чтобы мы могли получить доступ к платному API. Затем мы импортируем классы ChatOpenAI и OpenAIEmbeddings из библиотеки langchain_openai:

```
import openai
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
```

ChatOpenAI используется для взаимодействия с чат-моделями OpenAI, а OpenAIEmbeddings — для генерации эмбеддингов из текста.

В следующей строке мы загружаем переменные окружения из файла env.txt с помощью функции load_dotenv:

_ = load_dotenv(dotenv_path='env.txt')

Мы используем файл env.txt для безопасного хранения конфиденциальной информации (API-ключа), скрывая его от системы контроля версий и следуя лучшим практикам управления секретами.

Затем мы передаем API-ключ в OpenAI, используя следующий код:

```
os.environ['OPENAI_API_KEY'] = os.getenv('OPENAI_API_KEY')
openai.api_key = os.environ['OPENAI_API_KEY']
```

Сначала мы устанавливаем API-ключ в переменную окружения OPENAI_API_KEY. Затем передаем его в библиотеку openai, используя значение, полученное из переменной окружения. На этом этапе мы можем использовать интеграцию OpenAI с LangChain для вызова LLM, размещенной в OpenAI, с правильными настройками доступа.

Далее в коде мы определяем, какую LLM будем использовать:

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

Эта строка создает экземпляр класса ChatOpenAI, задавая модель gpt-4o-mini и устанавливая параметр temperature в 0. Параметр temperature управляет случайностью сгенерированных ответов: чем ниже значение, тем более точные и детерминированные будут результаты. В настоящее время gpt-4o-mini — самая новая и производительная модель, а также наиболее экономичная среди серии GPT-4. Однако даже эта модель стоит в 10 раз дороже, чем gpt-3.5-turbo, которая тоже является достаточно мощной.

Самая дорогая модель OpenAI, gpt-4-32k, уступает gpt-4o-mini по скорости и качеству, но имеет контекстное окно в 4 раза больше. В ближайшее время могут появиться новые модели, включая gpt-5, которые могут быть дешевле и мощнее. Важно помнить, что новейшая модель не обязательно будет самой дорогой, и постоянно появляются альтернативные версии, которые могут быть более производительными и экономичными. Следите за последними релизами моделей и оценивайте их стоимость, возможности и другие характеристики, чтобы принимать взвешенные решения о переходе на новую версию.

Однако вы не ограничены только OpenAI. LangChain упрощает смену LLM и позволяет рассматривать альтернативные решения внутри сообщества LangChain. Рассмотрим другие доступные варианты.

Together AI

Together AI предлагает удобный для разработчиков набор сервисов с доступом к различным моделям. Их цены на размещенные LLM сложно превзойти, и часто предоставляется $5.00 бесплатного кредита для тестирования различных моделей. Если вы новичок в Together API, используйте ссылку, чтобы создать API-ключ и добавить его в файл env.txt так же, как мы сделали это с OpenAI API-ключом.

При переходе на эту страницу вам предложат $5.00 кредитов, которые станут доступны после нажатия кнопки Get started. Для получения этого кредита вводить данные кредитной карты не требуется. Не забудьте добавить новый API-ключ в файл env.txt под именем TOGETHER_API_KEY. После входа в систему вы можете просмотреть текущие тарифы для каждой модели.

Например, Meta Llama 3 70B instruct (Llama-3-70b-chat-hf) в настоящее время стоит $0.90 за 1 миллион токенов. Эта модель сопоставима по возможностям с ChatGPT-4, но ее запуск в Together AI обходится значительно дешевле, чем у OpenAI. Еще одна мощная модель — Mixtral (mixture of experts), стоимость которой составляет $1.20 за 1 миллион токенов.

Настройка Together AI

Следуйте этим шагам для подключения и использования Together AI. Начнем с установки пакета, необходимого для работы с Together API:

```
# new installation
```

```
%pip install --upgrade langchain-together==0.2.0
```

Это подготовит нас к использованию интеграции между Together API и LangChain.

```
from langchain_together import ChatTogether
_ = load_dotenv(dotenv_path='env.txt')
```

Мы импортируем необходимые компоненты LangChain для работы с ChatTogether и загружаем API-ключ (не забудьте добавить его в файл env.txt перед запуском этого кода!).

Как и ранее с OpenAI API-ключом, мы подтягиваем TOGETHER_API_KEY, чтобы обеспечить доступ к вашему аккаунту.

```
os.environ['TOGETHER_API_KEY'] = os.getenv('TOGETHER_API_KEY')
```

Мы будем использовать Llama 3 Chat и Mistral's Mixtral 8X22B Instruct, но можно выбрать среди более чем 50 моделей по ссылке. Возможно, для ваших задач найдется более подходящая модель!

Далее мы определяем модель:

```
# Selection of models to test:
llama3llm = ChatTogether(
    together_api_key=os.environ['TOGETHER_API_KEY'],
    model="meta-llama/Llama-3-70b-chat-hf",
)
mistralexpertsllm = ChatTogether(
    together_api_key=os.environ['TOGETHER_API_KEY'],
    model="mistralai/Mixtral-8x22B-Instruct-v0.1",
)
```

В приведенном фрагменте кода мы создаем два экземпляра LLM, которые затем можно будет использовать в оставшейся части кода для получения результатов.

Далее обновим финальный код для работы с моделью Llama 3.

```
# Using Llama 3
llama3_rag_chain_from_docs = (
    RunnablePassthrough.assign(context=(lambda x: format_docs(x["context"])))
    | RunnableParallel(
        {"relevance_score": (
            RunnablePassthrough()
            | (lambda x: relevance_prompt_template.format(question=x['question'],
retrieved_context=x['context']))
            | llama3llm
            | str_output_parser
        ), "answer": (
            RunnablePassthrough()
            | prompt
            | llama3llm
            | str_output_parser
        )}
    )
    | RunnablePassthrough().assign(final_answer=conditional_answer)
)
```

Этот код должен выглядеть знакомо — это та же цепочка RAG, которую мы использовали ранее, но теперь она работает с Llama 3 LLM.

```
llama3_rag_chain_with_source = RunnableParallel(
    {"context": ensemble_retriever, "question": RunnablePassthrough()}
).assign(answer=llama3_rag_chain_from_docs)
```

Это финальный RAG-конвейер, обновленный для использования с предыдущей RAG-цепочкой, ориентированной на Llama 3.

Запуск RAG-конвейера с Llama 3. Теперь мы выполним аналогичный код, который использовался ранее, но заменим ChatGPT-4o-mini на Llama 3 LLM в RAG-конвейере.

```
# User Query
llama3_result = llama3_rag_chain_with_source.invoke(user_query)
llama3_retrieved_docs = llama3_result['context']
print(f"Original Question: {user_query}\n")
print(f"Relevance Score: {llama3_result['answer']['relevance_score']}\n")
print(f"Final Answer:\n{llama3_result['answer']['final_answer']}\n\n")
print("Retrieved Documents:")
for i, doc in enumerate(llama3_retrieved_docs, start=1):
   print(f"Document {i}: Document ID: {doc.metadata['id']} source: {doc.metadata['source']}")
   print(f"Content:\n{doc.page_content}\n")
```

Ответ на вопрос "Какие экологические инициативы есть у Google?":

```
Google's environmental initiatives include:
1. Empowering individuals to take action: Offering
sustainability features in Google products, such as eco-friendly
routing in Google Maps, energy efficiency features in Google
Nest thermostats, and carbon emissions information in Google
Flights…
[TRUNCATED]
10. Engagement with external targets and initiatives:
Participating in industry-wide initiatives and partnerships to
promote sustainability, such as the RE-Source Platform, iMasons
Climate Accord, and World Business Council for Sustainable
Development.
```

Давайте посмотрим, как это выглядит при использовании модели Mixture of Experts (MoE).

```
# Using Mistral
mistralexperts_rag_chain_from_docs = (
   RunnablePassthrough.assign(context=(lambda x: format_docs(x["context"])))
   | RunnableParallel(
      {"relevance_score": (
         RunnablePassthrough()
         | (lambda x: relevance_prompt_template.format(question=x['question'],
retrieved_context=x['context']))
         | mistralexpertsllm
         | str_output_parser
      ), "answer": (
         RunnablePassthrough()
         | prompt
         | mistralexpertsllm
         | str_output_parser
      )}
   )
   | RunnablePassthrough().assign(final_answer=conditional_answer)
)
```

Опять же, это тот же RAG-конвейер, но теперь он использует модель Mixture of Experts LLM.

```
mistralexperts_rag_chain_with_source = RunnableParallel(
   {"context": ensemble_retriever, "question": RunnablePassthrough()}
).assign(answer=mistralexperts_rag_chain_from_docs)
```

Как и ранее, мы обновляем финальный RAG-конвейер для работы с моделью Mixture of Experts вместо ChatGPT-4o-mini. Этот код позволяет получить результаты с Mixture of Experts вместо ChatGPT-4o-mini.

```
# User Query
mistralexperts_result = mistralexperts_rag_chain_with_source.invoke(user_query)
mistralexperts_retrieved_docs = mistralexperts_result['context']

print(f"Original Question: {user_query}\n")
print(f"Relevance Score: {mistralexperts_result['answer']['relevance_score']}\n")
print(f"Final Answer:\n{mistralexperts_result['answer']['final_answer']}\n\n")
print("Retrieved Documents:")
for i, doc in enumerate(mistralexperts_retrieved_docs, start=1):
   print(f"Document {i}: Document ID: {doc.metadata['id']} source: {doc.metadata['source']}")
   print(f"Content:\n{doc.page_content}\n")
```

Ответ на вопрос "Какие экологические инициативы есть у Google?":

```
Google's environmental initiatives are organized around three
key pillars: empowering individuals to take action, working
together with partners and customers, and operating their
business sustainably.
1. Empowering individuals: Google provides sustainability
features like eco-friendly routing in Google Maps, energy
efficiency features in Google Nest thermostats, and carbon
emissions information in Google Flights. Their goal is to help
individuals, cities, and other partners collectively reduce 1
gigaton of carbon equivalent emissions annually by 2030.
[TRUNCATED]
Additionally, Google advocates for strong public policy action
to create low-carbon economies, they work with the United
Nations Framework Convention on Climate Change (UNFCCC) and
support the Paris Agreement's goal to keep global temperature
rise well below 2°C above pre-industrial levels. They also
engage with coalitions and sustainability initiatives like
the RE-Source Platform and the Google.org Impact Challenge on
Climate Innovation.
```

Сравните это с исходным ответом, который мы видели в предыдущих главах.

```
Google's environmental initiatives include empowering
individuals to take action, working together with partners and
customers, operating sustainably, achieving net-zero carbon
emissions, focusing on water stewardship, engaging in a circular
economy, and supporting sustainable consumption of public goods.
They also engage with suppliers to reduce energy consumption
and greenhouse gas emissions, report environmental data, and
assess environmental criteria. Google is involved in various
sustainability initiatives, such as the iMasons Climate Accord,
ReFED, and supporting projects with The Nature Conservancy.
They also work with coalitions like the RE-Source Platform
and the World Business Council for Sustainable Development.
Additionally, Google invests in breakthrough innovation and
collaborates with startups to tackle sustainability challenges.
They also focus on renewable energy and use data analytics tools
to drive more intelligent supply chains.
```

Новые ответы от моделей Llama 3 и Mixture of Experts оказались более развернутыми и, по крайней мере, не уступающими, а возможно, даже превосходящими по детализации ответы, которые мы

получали с помощью модели gpt-4o-mini от OpenAI, при этом их использование обходится значительно дешевле по сравнению с более дорогими, но и более мощными моделями OpenAI.

Расширение возможностей LLM

Некоторые функции объектов LLM могут быть использованы более эффективно в вашем RAG-приложении. Как описано в документации LangChain по LLM (на февраль 2025 документ устарел).

Все LLM реализуют интерфейс Runnable, который включает в себя стандартные реализации всех методов, таких как ainvoke, batch, abatch, stream, astream. Это обеспечивает базовую поддержку асинхронных вызовов, потоковой обработки и пакетной обработки для всех LLM.

Эти ключевые функции могут значительно ускорить обработку запросов в RAG-приложении, особенно если выполняется несколько вызовов LLM одновременно. В следующих разделах мы рассмотрим основные методы и их преимущества.

Асинхронные вызовы (Async)

По умолчанию асинхронная поддержка выполняет стандартный синхронный метод в отдельном потоке. Это позволяет другим частям асинхронной программы продолжать выполнение, пока языковая модель обрабатывает запрос.

Потоковая обработка (Stream)

Поддержка потоковой обработки обычно возвращает Iterator (или AsyncIterator для асинхронных потоков) с одним элементом — финальным результатом от языковой модели. Хотя это не обеспечивает пословного стриминга, оно гарантирует совместимость с интеграциями LangChain, ожидающими поток токенов.

Пакетная обработка (Batch)

Пакетная обработка позволяет обрабатывать несколько входных данных одновременно. В синхронном режиме используются многопоточные вычисления. В асинхронном режиме применяется asyncio.gather. Количество одновременно выполняемых задач можно регулировать через параметр max_concurrency в RunnableConfig. Однако не все LLM нативно поддерживают все эти функции. Для рассмотренных нами моделей, а также многих других, LangChain предоставляет подробную таблицу совместимости.

Саммари главы

В этой главе мы рассмотрели ключевые компоненты систем RAG в контексте LangChain: векторные хранилища, ретриверы и LLM. Мы разобрали доступные варианты для каждого компонента, их сильные и слабые стороны, а также сценарии, в которых один вариант может быть предпочтительнее другого.

В начале главы мы изучили векторные хранилища, которые играют ключевую роль в эффективном хранении и индексации векторных представлений документов базы знаний. LangChain поддерживает интеграцию с различными векторными хранилищами, такими как Pinecone, Weaviate, FAISS и PostgreSQL с векторными расширениями. Выбор хранилища зависит от таких факторов, как масштабируемость, производительность поиска и требования к развертыванию.

Затем мы рассмотрели ретриверы, которые отвечают за поиск в векторном хранилище и извлечение релевантных документов на основе входного запроса. LangChain предлагает широкий набор ретриверов, включая плотные ретриверы, разреженные ретриверы (BM25), а также комбинированные ретриверы, объединяющие результаты нескольких методов поиска.

Наконец, мы изучили роль LLM в системах RAG. LLM дополняют RAG, обеспечивая глубокое понимание контекста запроса и способствуя эффективному извлечению информации из базы знаний. В главе были рассмотрены интеграции LangChain с различными провайдерами LLM, такими как OpenAI и Together AI, а также обсуждались возможности моделей, стоимость их использования и дополнительные функции. Мы также затронули расширенные возможности LLM в LangChain (асинхронные вызовы, потоковая и пакетная обработка) и сравнили нативную поддержку этих функций в различных LLM-интеграциях.

В следующей главе мы продолжим изучение возможностей LangChain для создания эффективных RAG-приложений, сосредоточив внимание на вспомогательных компонентах, которые могут поддерживать ключевые элементы, рассмотренные в этой главе.