Глава 12. Объединение RAG с возможностями AI-агентов и LangGraph

Это продолжение перевода книги Кит Борн. Раскрытие потенциала данных с помощью генеративного ИИ и технологии RAG. Эта глава открывает третью часть книги «Реализация продвинутого RAG». В ней вы изучите интеграцию AI-агентов с LangGraph, использование стратегий инженерии промтов для оптимизации поиска и генерации, расширение запросов, декомпозицию запросов и мультимодальный RAG.

Вызов большой языковой модели (LLM) может быть мощным инструментом, но если поместить вашу логику в цикл с целью решения более сложных задач, то разработка RAG (генерации, дополненной поиском) выйдет на новый уровень. Именно в этом заключается концепция агентов. За последний год разработка LangChain была сосредоточена на улучшении поддержки агентных рабочих процессов, добавляя функциональность, которая позволяет более точно управлять поведением и возможностями агентов.

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. Часть этого прогресса связана с появлением LangGraph — еще одного относительно нового компонента LangChain. В связке агенты и LangGraph образуют мощный подход к улучшению RAG-приложений.

В этой главе мы углубимся в изучение элементов агентов, и рассмотрим, как они могут быть интегрированы в вашу систему:

- Основы AI-агентов и их интеграция с RAG
- Графы, AI-агенты и LangGraph
- Добавление агента LangGraph в RAG-приложение
- Инструменты и наборы инструментов
- Состояние агента
- Основные концепции теории графов

Предыдущая глава Содержание Следующая глава

Код для этой главы размещен в репозитории GitHub.

В следующем разделе мы разберем основы AI-агентов и их интеграцию с RAG, чтобы подготовить вас к дальнейшим концепциям и лабораторной работе.

Основы AI-агентов и интеграции с RAG

Когда мы общаемся с новыми разработчиками в области генеративного ИИ, нам часто говорят, что концепция AI-агента является одной из самых сложных для понимания. Эксперты, говоря об агентах, часто используют абстрактные термины, описывая широкий спектр задач, которые агенты могут выполнять в RAG-приложении, но не объясняют конкретно, что такое AI-агент и как он работает. Я считаю, что проще всего развеять миф о сложной природе AI-агента, объяснив, чем он на самом деле является, а это довольно простой концепт. Чтобы создать AI-агента в его базовой форме, вам нужно просто взять ту же LLM, с которой вы уже работали в предыдущих главах, и добавить цикл, который завершается, когда поставленная задача выполнена. И все! Это просто цикл.

Рисунок 12.1 представляет цикл RAG-агента, с которым вы будете работать в лаборатории кода:

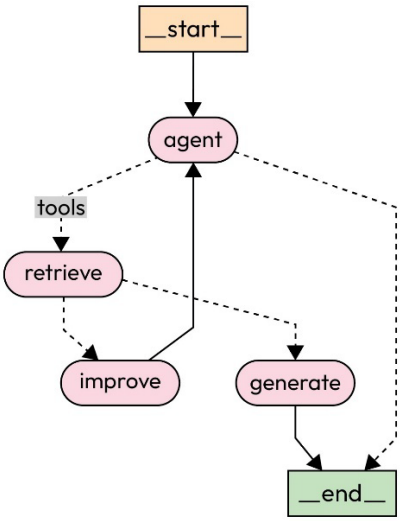

Рис. 12.1. График потока управления агента

Это простой набор шагов, которые повторяются до тех пор, пока агент не решит, что успешно завершил поставленную перед ним задачу. Овальные блоки, agent и retrieve, называются узлами, а соединяющие их линии — ребрами. Пунктирные линии также являются ребрами, но особого типа, называемого условными ребрами, которые обозначают точки принятия решений.

Несмотря на простоту, добавление цикла в вызовы LLM делает его намного более мощным, чем простое использование модели, поскольку позволяет лучше задействовать ее способность к рассуждению и разбиению сложных задач на более мелкие подзадачи. Это увеличивает вероятность успешного выполнения задачи, особенно если речь идет о сложных многошаговых процессах RAG.

Когда LLM выполняет агентные задачи, ей также предоставляются функции, называемые инструментами, и она использует свои возможности рассуждения, чтобы определить, какой инструмент использовать, как его применять и какие данные ему передавать. На этом этапе сложность может возрасти очень быстро. Можно использовать несколько агентов, различные инструменты, интегрированные графы знаний, которые помогают направлять агентов, различные фреймворки, предлагающие варианты агентных систем, различные архитектурные подходы и многое другое. Однако в этой главе мы сосредоточимся исключительно на том, как AI-агенты могут улучшить RAG-приложения. Как только вы увидите силу использования AI-агентов, я не сомневаюсь, что вам захочется применять их и в других генеративных AI-приложениях!

Жизнь в мире AI-агентов

С учетом всего ажиотажа вокруг агентов может показаться, что LLM уже устарели. Но это не так. Используя AI-агентов, вы фактически раскрываете еще более мощную версию LLM — версию, в которой LLM выступает в роли «мозга» агента, позволяя ему рассуждать и находить многошаговые решения, выходящие далеко за рамки простых одноразовых вопросов, для которых LLM чаще всего применяются. Агент просто добавляет прослойку между пользователем и LLM, направляя ее на выполнение задачи, которая может потребовать нескольких запросов.

Если задуматься, это больше похоже на то, как задачи решаются в реальном мире. Большинство задач основаны на длинной цепочке наблюдений, рассуждений и адаптации к новым обстоятельствам. В реальном мире мы редко взаимодействуем с людьми, задачами и объектами так же, как с LLM в сети. Чаще всего это процесс постепенного накопления понимания, знаний и контекста, который помогает нам находить оптимальные решения. AI-агенты лучше справляются с таким подходом к решению проблем.

Агенты могут значительно повлиять на вашу работу с RAG, но что насчет концепции LLM как их мозга? Давайте разберемся подробнее.

LLM как мозг агента

Если рассматривать LLM как мозг AI-агента, следующий логичный шаг — использовать для этой роли самую умную LLM, которую можно найти. Способности LLM определяют, насколько хорошо AI-агент может рассуждать и принимать решения, что, в свою очередь, влияет на качество запросов в вашем RAG-приложении.

Однако здесь есть одно важное отличие от реального мира, и оно играет нам на руку. В отличие от настоящих агентов, AI-агент может в любой момент заменить свой «мозг» LLM на другой. Более того, можно даже дать ему несколько «мозгов» LLM, которые будут проверять друг друга и следить за тем, чтобы процесс шел по плану. Это дает нам гибкость, позволяющую постоянно совершенствовать возможности наших агентов.

А как LangGraph или графы в целом связаны с AI-агентами? Об этом мы поговорим дальше.

Графы, AI-агенты и LangGraph

LangChain представил LangGraph в 2024 году, так что это относительно новая технология. Это расширение, построенное поверх LangChain Expression Language (LCEL), предназначенное для создания композиционных и настраиваемых агентных рабочих процессов. LangGraph опирается на концепции теории графов, такие как узлы и ребра, но с упором на управление AI-агентами. Хотя старый способ управления агентами через класс AgentExecutor все еще существует, теперь рекомендуется использовать LangGraph для создания агентов в LangChain.

LangGraph добавляет два важных компонента для поддержки агентов:

- Возможность легко определять циклы (цикличные графы)
- Встроенную память

Он предоставляет готовый объект, аналогичный AgentExecutor, который позволяет разработчикам организовывать работу агентов с помощью графового подхода.

За последние несколько лет появилось множество исследований, концепций и подходов к созданию агентов в RAG-приложениях, таких как оркестрационные агенты, ReAct-агенты, агенты самоусовершенствования и мультиагентные фреймворки. Общая идея этих подходов заключается в использовании **циклического графа**, который представляет поток управления агентом. Хотя многие из этих подходов с точки зрения реализации постепенно устаревают, их концепции остаются полезными и находят применение в графовой среде LangGraph.

LangGraph стал мощным инструментом для поддержки агентов и управления их процессами в RAG-приложениях. Он позволяет разработчикам описывать и представлять **одноагентные и многоагентные процессы** в виде графов, обеспечивая **четкий контроль за их выполнением**. Такая управляемость крайне важна для предотвращения ошибок, с которыми сталкивались разработчики на ранних этапах создания агентов.

Например, популярный подход **ReAct** был одной из первых парадигм для построения агентов. **ReAct** расшифровывается как **reason + act** (рассуждение + действие). В этом паттерне LLM сначала анализирует ситуацию и размышляет, что делать, затем выбирает действие. Действие выполняется в среде, после чего агент получает результат наблюдения. Затем LLM снова анализирует ситуацию, решает, что делать дальше, и повторяет процесс, пока не достигнет поставленной цели. Если отобразить этот процесс в виде схемы, он может выглядеть так:

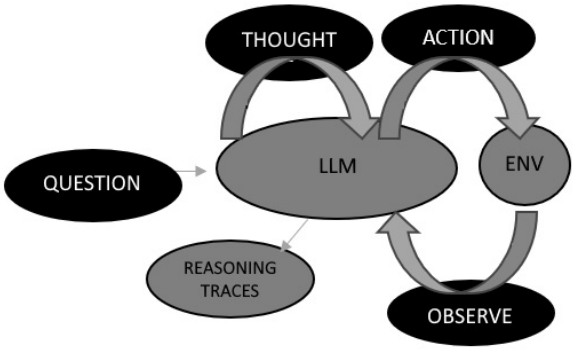

Рис. 12.2. Циклический граф ReAct

Набор циклов, показанный на рисунке 12.2, можно представить с помощью циклических графов в LangGraph, где каждый шаг отображается в виде узлов и ребер. Используя этот графовый подход, можно увидеть, как инструмент LangGraph, предназначенный для построения графов в LangChain, может стать основой вашего агентного фреймворка. По мере создания нашего фреймворка агентов

мы будем использовать LangGraph для представления этих агентных циклов, что поможет описывать и координировать поток управления.

Фокус на потоке управления критически важен для решения ранних проблем, возникавших при разработке агентов, когда отсутствие контроля приводило к появлению неуправляемых агентов, которые не могли завершить свои циклы или фокусировались не на той задаче.

Еще один ключевой элемент, встроенный в LangGraph, — персистентность. Она позволяет сохранять память агента, предоставляя ему информацию обо всех предыдущих действиях, что соответствует компоненту OBSERVE, показанному на рисунке 12.2. Это особенно полезно для одновременного ведения нескольких разговоров, запоминания предыдущих итераций и выполненных действий. Персистентность также поддерживает человека в цикле (human-in-the-loop), позволяя лучше контролировать действия агента в ключевые моменты его работы. Исходная научная работа, в которой был представлен подход ReAct, доступна по ссылке.

Перейдем к лабораторной работе, в ходе которой создадим агента и разберем важные концепции по мере их появления в коде.

Лаборатория кода 12.1 — Добавление агента LangGraph в RAG

В этой лабораторной работе мы добавим агента в существующий RAG-конвейер, который сможет принимать решения о том, стоит ли извлекать данные из индекса или выполнить веб-поиск. Мы увидим, как агент анализирует данные, полученные из различных источников, чтобы предоставить более развернутый и точный ответ на ваш вопрос.

Добавляя код агента, мы познакомимся с новыми компонентами, такими как инструменты (tools), наборы инструментов (toolkits), графы (graphs), узлы (nodes), ребра (edges) и сам агент. Для каждого компонента мы подробно разберем, как он взаимодействует с другими элементами системы и поддерживает работу RAG-приложения.

Мы также добавим функциональность, чтобы этот агент работал больше как чат, а не просто как система «вопрос-ответ».

Сначала мы установим несколько новых пакетов для поддержки разработки нашего агента:

```
%pip install tiktoken
%pip install langgraph
```

В первой строке мы устанавливаем пакет tiktoken, который является пакетом OpenAI, используемым для токенизации текстовых данных перед их передачей в языковые модели. Далее мы устанавливаем пакет langgraph, который мы обсуждали ранее.

Затем мы добавляем новое определение LLM и обновляем существующее:

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

Новый экземпляр agent_llm будет служить мозгом агента, управляя логикой и выполнением задач, тогда как исходный llm останется в системе для выполнения основных задач LLM, которые мы использовали ранее. Хотя в нашем примере обе LLM используют одну и ту же модель и параметры, вы можете (и должны) экспериментировать с разными моделями для этих задач, чтобы найти наиболее подходящую комбинацию для вашего RAG-приложения.

Вы также можете добавить дополнительные LLM для выполнения специализированных задач, таких как улучшение ответов или оценка документов (score_documents), если обнаружите, что другая модель справляется с этим лучше или если у вас есть своя обученная или дообученная модель для этих действий. Например, часто простые задачи передаются более быстрым и дешевым LLM, если они могут их успешно выполнять.

Этот код предоставляет широкие возможности для настройки, которыми вы можете воспользоваться! Также обратите внимание, что мы добавляем streaming=True в определение LLM. Это включает потоковую передачу данных, что особенно полезно для агента, который может делать несколько вызовов (иногда параллельно), постоянно взаимодействуя с LLM.

Теперь мы пропустим часть кода, относящуюся к определениям ретриверов (dense_retriever, sparse_retriever и ensemble_retriever), и добавим наш первый инструмент. У инструментов в контексте агентов есть очень специфическое и важное назначение, поэтому давайте разберемся, что это такое.

Инструменты и наборы инструментов

В следующем коде мы добавим инструмент веб-поиска:

```
# Tavily Setup
# set up your API key at https://tavily.com/
# add the API key to your env.txt file
from langchain_community.tools.tavily_search import TavilySearchResults
_ = load_dotenv(dotenv_path='env.txt')
os.environ['TAVILY_API_KEY'] = os.getenv('TAVILY_API_KEY')
!export TAVILY_API_KEY=os.environ['TAVILY_API_KEY']
web_search = TavilySearchResults(max_results=4)
web_search_name = web_search.name
```

Вам потребуется получить еще один API-ключ и добавить его в файл env.txt, который мы использовали ранее для OpenAI и Together API. Как и в случае с этими API, вам нужно зайти на соответствующий сайт, создать API-ключ и скопировать его в env.txt. Сайт Tavily доступен по адресу: https://tavily.com/

Мы снова выполняем код, который загружает данные из env.txt, затем создаем объект TavilySearchResults с параметром max_results = 4, что означает, что при поиске мы получим не более четырех результатов. Затем мы присваиваем web_search.name переменной web_search_name, чтобы позже использовать это значение для передачи агенту.

Вы можете запустить этот инструмент напрямую, используя следующий код:

```
web_search.invoke(user_query)
```

Запуск этого инструмента с user_query выдаст результат, который выглядит следующим образом (сокращен для удобства):

```
[{'url': 'http://sustainability.google/',
  'content': "Google Maps\nChoose the most fuel-efficient route\
nGoogle Shopping\nShop for more efficient appliances for your home\
nGoogle Flights\nFind a flight with lower per-traveler carbon
emissions\nGoogle Nest\...[TRUNCATED HERE]"},
…
  'content': "2023 Environmental Report. Google's 2023 Environmental
Report outlines how we're driving positive environmental outcomes
throughout our business in three key ways: developing products and
technology that empower individuals on their journey to a more
sustainable life, working together with partners and organizations
everywhere to transition to resilient, low-carbon systems, and
operating ..."}]
```

Рис. 12.3. Результат работы инструмента веб-поиска

Мы сократили вывод, чтобы занять меньше места в книге, но попробуйте запустить этот код, и вы увидите четыре результата, как мы и запросили. Все они относятся к теме, указанной в user_query. Обратите внимание, что вам не нужно будет запускать этот инструмент напрямую в вашем коде, как мы сделали сейчас.

Мы только что создали первый инструмент для агента! Это инструмент поиска, который агент сможет использовать для получения дополнительной информации из интернета.

Концепция инструментов (tools) в LangChain и при создании агентов основана на том, что агенту нужно предоставить действия, которые он может выполнять для решения задач. Инструменты — это механизм, который делает это возможным. Вы определяете инструмент, как мы только что сделали для веб-поиска, а затем добавляете его в список инструментов, которыми агент сможет воспользоваться.

Прежде чем мы создадим этот список, давайте добавим еще один ключевой инструмент для RAG-приложения — инструмент поиска (retriever tool):

```
from langchain.tools.retriever import create_retriever_tool
retriever_tool = create_retriever_tool(
   ensemble_retriever,
   "retrieve_google_environmental_question_answers",
   "Extensive information about Google environmental efforts from 2023.",
)
retriever_tool_name = retriever_tool.name
```

Обратите внимание, что веб-поисковый инструмент мы импортировали из langchain_community.tools.tavily_search, тогда как этот инструмент импортируется из langchain.tools.retriever. Это связано с тем, что Tavily — это сторонний инструмент, а retriever является частью функциональности LangChain.

После импорта create_retriever_tool мы используем его для создания инструмента retriever_tool. Аналогично web_search_name, мы извлекаем retriever_tool.name, чтобы позднее ссылаться на него в коде. Вы также можете заметить, что retriever использует ensemble_retriever — это тот же ретривер, который мы создавали в лабораторной работе 8.3 в главе 8!

Еще один важный момент: название инструмента для агента указывается во втором поле, и мы назвали его retrieve_google_environmental_question_answers. Обычно мы стараемся сокращать названия переменных, но для инструментов, которыми будет пользоваться агент, полезнее давать более подробные и понятные имена, чтобы агент лучше понимал, какие инструменты у него есть в распоряжении.

Теперь у нас есть два инструмента для агента! Однако мы еще не сообщили агенту о них, поэтому теперь упакуем их в список, который позже передадим агенту:

```
# Define tools for agent
tools = [web_search, retriever_tool]
```

Здесь мы добавляем два ранее созданных инструмента — web_search и retriever_tool — в список tools. Если бы у нас были другие инструменты, которые мы хотели бы сделать доступными для агента, мы также могли бы добавить их в этот список. В экосистеме LangChain существуют сотни инструментов.

Важно убедиться, что LLM, которую вы используете, хорошо справляется с рассуждениями и работой с инструментами. Как правило, чат-модели (chat-tuned models) лучше адаптированы к вызову инструментов, тогда как обычные LLM без такой настройки могут не справляться с этой задачей — особенно если инструменты сложные или требуют многократных вызовов. Также правильные названия и описания инструментов играют важную роль в успешной работе агента с ними.

В агенте, который мы строим, у нас уже есть все нужные инструменты, но вам также стоит обратить внимание на toolkits — это наборы инструментов, сгруппированные по задачам. LangChain предоставляет список доступных на данный момент toolkits на своем сайте (по той же ссылки, что и инструменты). Например, если вы работаете с данными в pandas DataFrame, вы можете использовать pandas DataFrame toolkit, который предоставляет агенту разные способы доступа к этим данным.

Как описано на сайте LangChain:

> *Для выполнения многих распространенных задач агенту понадобится набор связанных инструментов. Для этого LangChain предоставляет концепцию toolkits — групп из примерно 3-5 инструментов, необходимых для выполнения конкретных целей. Например, GitHub toolkit содержит инструменты для поиска по GitHub issues, чтения файлов, комментирования и других действий.*

Если ваш агент фокусируется на определенных задачах или работает с популярными интеграциями LangChain (например, с Salesforce), скорее всего, уже существует toolkit, который включает в себя все нужные инструменты сразу.

Теперь, когда инструменты готовы, давайте приступим к созданию компонентов агента, начиная с состояния агента (agent state).

Состояние агента

Состояние агента — это ключевой компонент любого агента, созданного с помощью LangGraph. Используя LangGraph, вы создаете класс AgentState, который определяет «состояние» агента и отслеживает его со временем. Это локальный механизм, доступный для всех частей графа, и его можно сохранять в слое персистентности.

Здесь мы настраиваем состояние для нашего RAG-агента:

```
from typing import Annotated, Literal, Sequence, TypedDict
from langchain_core.messages import BaseMessage
from langgraph.graph.message import add_messages
class AgentState(TypedDict):
   messages: Annotated[Sequence[BaseMessage], add_messages]
```

Этот код импортирует необходимые пакеты для настройки AgentState. Например, BaseMessage — это базовый класс для представления сообщений в диалоге между пользователем и AI-агентом. Он используется для определения структуры и свойств сообщений, хранящихся в состоянии агента. Затем создается граф и объект state, который передается между узлами.

Состояние агента может хранить различные типы данных, но в случае нашего RAG-агента оно представляется списком сообщений.

Затем нам нужно импортировать еще несколько пакетов для настройки других частей агента:

```
from langchain_core.messages import HumanMessage
from langchain_core.pydantic_v1 import BaseModel, Field
from langgraph.prebuilt import tools_condition
```

В этом коде мы импортируем HumanMessage. Это специальный тип сообщения, который представляет собой сообщение от пользователя. Он будет использоваться при формировании промта для генерации ответа агентом. Мы также импортируем BaseModel и Field. BaseModel — это класс из библиотеки Pydantic, который используется для определения моделей данных и их валидации. Field — это класс из Pydantic, который позволяет задавать свойства и правила валидации полей внутри модели данных.

Последним мы импортируем tools_condition. Это готовая функция из LangGraph, предназначенная для определения, какие инструменты агент должен использовать в зависимости от текущего состояния диалога.

Эти импортированные классы и функции используются по всему коду для определения структуры сообщений, валидации данных и управления потоком разговора на основе решений агента. Они предоставляют необходимые строительные блоки для создания языкового агента с использованием LangGraph.

Далее мы определяем основной промт (который пользователь вводит в систему):

```
generation_prompt = PromptTemplate.from_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. Provide a thorough description to
   fully answer the question, utilizing any relevant
   information you find.

   Question: {question}
   Context: {context}

   Answer:"""
)
```

Этот код заменяет предыдущий, который использовался в прошлых лабораторных работах:

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

Мы изменяем его название на generation_prompt, чтобы четче обозначить его назначение.

Использование графов в коде сейчас начнет значительно возрастать, но прежде чем мы перейдем к этому, нам нужно рассмотреть основные концепции теории графов.

Основные концепции теории графов

Чтобы лучше понять, как мы будем использовать LangGraph в следующих блоках кода, полезно рассмотреть ключевые концепции теории графов. Графы — это математические структуры, которые используются для представления взаимосвязей между объектами. Эти объекты называются узлами (nodes), а связи между ними, обычно изображаемые в виде линий, называются ребрами (edges). Вы уже видели эти концепции на рисунке 12.1, но важно понимать, как они относятся к любому графу и как используются в LangGraph.

В LangGraph существуют разные типы ребер, представляющие различные типы связей. Например, *условное ребро* (conditional edge), о котором мы упоминали в рисунке 12.1, обозначает момент принятия решения, то есть выбор следующего узла. Эти ребра обозначают точки принятия решений. В контексте парадигмы ReAct такое ребро иногда называют *ребром действия* (action edge), так как именно на этом этапе происходит выполнение действий, соответствующих принципу reason + action в ReAct.

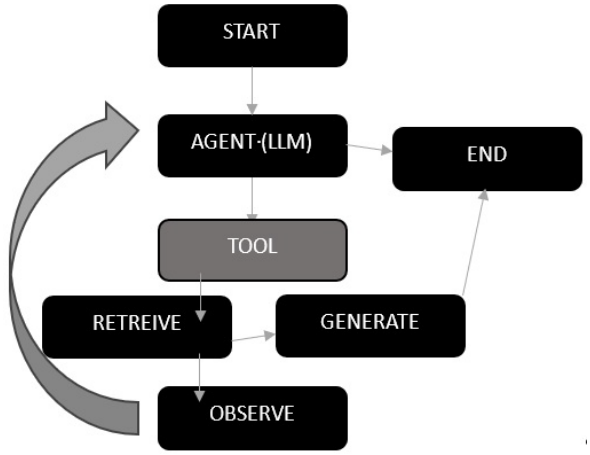

Рис. 12.4. Базовый граф, представляющий наше приложение RAG

В этом циклическом графе (Figure 12.3) узлы представляют: начало (start), агента (agent), инструмент поиска (retrieve tool), генерацию ответа (generation), наблюдение (observation), завершение процесса (end).

Ключевые ребра включают этапы, когда LLM принимает решение, какой инструмент использовать (в данном случае retrieve — единственный доступный инструмент), оценивает, достаточно ли найденных данных, и затем переходит к генерации.

Если агент решает, что извлеченных данных недостаточно, условное ребро направляет процесс обратно к агенту, чтобы он принял решение, нужно ли повторить попытку. Эти точки принятия решений представляют собой условные ребра (conditional edges).

Узлы и ребра в нашем агенте

Мы упоминали, что агентный RAG-граф включает три ключевых компонента: состояние, узлы, которые добавляют данные в состояние или обновляют его, и условные ребра, которые определяют, к какому узлу перейти дальше. Теперь мы можем рассмотреть каждый из этих компонентов в коде и увидеть, как они взаимодействуют. Исходя из этого, первым шагом будет добавление условного ребра, отвечающего за принятие решений. Мы определим ребро, которое решает, являются ли полученные документы релевантными. Эта функция выбирает, перейти ли к этапу генерации или попробовать снова.

Мы рассмотрим этот код пошагово, но помните, что это одна большая функция, начинающаяся с определения:

def score_documents(state) -> Literal["generate", "improve"]:

Функция score_documents анализирует, насколько полученные документы соответствуют запросу пользователя. Функция принимает state — это набор собранных сообщений. Таким образом, состояние становится доступным для этой функции условного ребра.

Теперь создаем модель данных:

```
class scoring(BaseModel):
   binary_score: str = Field(description="Relevance score 'yes' or 'no'")
```

Класс scoring с помощью Pydantic BaseModel содержит одно поле binary_score, которое принимает строковое значение: "yes" (релевантно) или "no" (нерелевантно).

Затем мы добавляем LLM, которая будет принимать решение:

```
llm_with_tool = llm.with_structured_output(scoring)
```

Объект llm_with_tool, объединяя LLM с моделью данных scoring для валидации выходных данных.

Как и раньше, нам нужно определить шаблон промта (PromptTemplate), который передаст информацию в LLM:

```
prompt = PromptTemplate(
   template="""You are assessing relevance of a retrieved document to a user question with a binary grade.
   Here is the retrieved document: {context}
   Here is the user question: {question}
   If the document contains keyword(s) or semantic meaning related to the user question, grade it as
relevant.
   Give a binary score 'yes' or 'no' to indicate whether the document is relevant to the question.""",
   input_variables=["context", "question"],
)
```

Этот промт объясняет LLM, что нужно оценить релевантность документа к запросу пользователя и присвоить оценку "yes" или "no".

Далее мы используем LCEL для создания цепочки обработки, комбинируя промт и LLM:

```
chain = prompt | llm_with_tool
```

chain определяет конвейер обработки данных для оценки документов, но пока не вызывает его.

Теперь извлекаем состояние агента (state) и получаем ключевые сообщения:

```
messages = state["messages"]
last_message = messages[-1]
question = messages[0].content
docs = last_message.content
```

Код извлекает данные из состояния и подготавливает контекст, который передается в LLM. Он включает: messages — список сообщений в диалоге, last_message — последнее сообщение в диалоге (предполагается, что это результат поиска), question — первое сообщение (предполагается, что это исходный вопрос пользователя), docs — текст документов, найденных при поиске.

Затем запускаем цепочку, передавая в нее вопрос и найденные документы:

```
scored_result = chain.invoke({"question": question, "context": docs})
score = scored_result.binary_score
```

Этот код извлекает оценку релевантности (binary_score) и присваивает ее переменной score.

Теперь принимаем окончательное решение:

```
if score == "yes":
   print("DECISION: DOCS RELEVANT")
   return "generate"
else:
```

```
    print("DECISION: DOCS NOT RELEVANT")
    print(score)
    return "improve"
```

Если оценка "yes", выводится сообщение "DECISION: DOCS RELEVANT", и функция возвращает "generate", что означает переход к генерации ответа. Если оценка "no", выводится сообщение "DECISION: DOCS NOT RELEVANT", и функция возвращает "improve", что означает улучшение запроса пользователя.

Эта функция определяет точку принятия решений в нашем рабочем процессе. Она оценивает, насколько найденные документы релевантны запросу, и либо переходит к генерации ответа, либо переформулирует запрос, если документы не подходят.

Теперь, когда мы определили условное ребро, переходим к созданию узлов, начиная с агента:

```
# Agent node - invokes the agent model to generate a response based on the current state.
# Decision choices (given the question): retrieve using the retriever tool, web_search tool, both, or end.
def agent(state):
    print("---CALL AGENT---")
    messages = state["messages"]
    llm = agent_llm.bind_tools(tools)
    response = llm.invoke(messages)
    return {"messages": [response]} # Return list, will get added to existing list
```

Эта функция представляет узел агента на нашем графе и вызывает модель агента, чтобы сгенерировать ответ на основе текущего состояния.

Функция agent принимает текущее состояние (state) как входной параметр, который содержит историю сообщений в диалоге. Она выводит сообщение, указывающее, что агент вызывается, извлекает сообщения из состояния, затем использует экземпляр agent_llm, который мы определили ранее с помощью ChatOpenAI. Это представляет "мозг" агента. Затем с помощью метода bind_tools мы связываем инструменты с моделью. После этого вызываем LLM агента, передавая ему сообщения, и сохраняем результат в переменной response.

Следующий узел, improve, отвечает за переформулирование user_query, если агент решает, что это необходимо:

```
def improve(state):
    print("    TRANSFORM QUERY    ")
    messages = state["messages"]
    question = messages[0].content
    msg = [
        HumanMessage(content=f"""\n
        Look at the input and try to reason about
        the underlying semantic intent / meaning.
        \n
        Here is the initial question:
        \n \n
        {question}
        \n \n
        Formulate an improved question:
        """)
    ]
    response = llm.invoke(msg)
    return {"messages": [response]}
```

Функция принимает текущее состояние (state) в качестве входных данных и возвращает словарь, в котором новый запрос добавляется в список сообщений. Она выводит сообщение о трансформации запроса, извлекает сообщения из состояния, получает первое сообщение (исходный запрос пользователя) и сохраняет его в переменную question. Затем с помощью класса HumanMessage создается новое сообщение, в котором LLM просят определить основной смысл запроса и

сформулировать улучшенный вопрос. Результат, полученный от LLM, сохраняется в response, и затем возвращается обновленный список сообщений.

Следующая функция — generate:

```
def generate(state):
   print("    GENERATE    ")
   messages = state["messages"]
   question = messages[0].content
   last_message = messages[-1]
   docs = last_message.content
   rag_chain = generation_prompt | llm | str_output_parser
   response = rag_chain.invoke({"context": docs, "question": question})
   return {"messages": [response]}
```

Эта функция похожа на этап генерации, который мы использовали в предыдущих лабораторных работах, но упрощена для возвращения только ответа. Функция принимает состояние (state), извлекает историю сообщений, получает первое сообщение (вопрос пользователя) и содержимое последнего сообщения (предположительно, полученные документы). Затем создается цепочка rag_chain, комбинируя: generation_prompt (заранее подготовленный шаблон генерации), llm (модель для генерации ответа), str_output_parser (парсер для строки).

Как и в других случаях LLM-промтов, мы *гидратируем* (hydrate) промт generation_prompt, передавая в него контекст (документы) и вопрос, а затем вызываем цепочку и возвращаем результат в виде обновленного списка сообщений.

Теперь мы готовы к созданию циклического графа в LangGraph, добавляя в него наши узлы и ребра.

Настройка циклического графа

Следующий важный шаг в нашем коде — это создание графа с использованием LangGraph. Сначала импортируем необходимые пакеты:

```
from langgraph.graph import END, StateGraph
from langgraph.prebuilt import ToolNode
```

Этот код импортирует несколько ключевых классов и функций из библиотеки langgraph: END — специальный узел, обозначающий конец рабочего процесса, StateGraph — класс для определения графа состояния рабочего процесса, ToolNode — класс для создания узла, который представляет инструмент или действие.

Определяем граф состояния, передавая в него AgentState:

```
workflow = StateGraph(AgentState)
```

Код создает экземпляр StateGraph, называемый workflow, который представляет новый граф для данного экземпляра StateGraph.

Добавляем узлы, между которыми будет происходить цикл, и связываем их с функциями:

```
workflow.add_node("agent", agent)  # агент
retrieve = ToolNode(tools)
workflow.add_node("retrieve", retrieve)  # поиск через веб или retriever
workflow.add_node("improve", improve)  # улучшение вопроса для лучшего поиска
workflow.add_node("generate", generate)  # генерация ответа, когда документы релевантны
```

Код добавляет несколько узлов в workflow, используя метод add_node:

- "agent" — узел, представляющий агента, вызывает функцию agent.
- "retrieve" — узел, представляющий поиск информации. Это специальный узел ToolNode, содержащий список инструментов, которые мы определили ранее (web_search и retriever_tool). Чтобы код был более читаемым, мы сначала создаем объект ToolNode (retrieve = ToolNode(tools)) и затем передаем его в add_node.
- "improve" — узел, отвечающий за улучшение запроса, вызывает функцию improve.
- "generate" — узел для генерации ответа, вызывает функцию generate.

Определяем стартовый узел графа:

```
workflow.set_entry_point("agent")
```

Этот код устанавливает начальный узел ("agent") в качестве входной точки графа.

Добавляем условное ребро для узла "agent", чтобы определить, следует ли выполнять поиск:

```
workflow.add_conditional_edges(
   "agent",
   tools_condition,
   {
      "tools": "retrieve",
      END: END,
   },
)
```

В этом коде tools_condition используется как условное ребро в графе. Оно определяет, должен ли агент выполнить поиск ("tools": "retrieve") или завершить процесс (END: END).

Шаг "retrieve" включает оба инструмента, которые мы сделали доступными для агента (web_search и retriever_tool). Если агент решает, что поиск не нужен, то workflow просто завершает работу.

Добавляем дополнительные ребра, которые используются после вызова узла "action":

```
workflow.add_conditional_edges("retrieve", score_documents)
workflow.add_edge("generate", END)
workflow.add_edge("improve", "agent")
```

После вызова узла "retrieve" добавляются условные ребра с использованием workflow.add_conditional_edges ("retrieve", score_documents). Это позволяет оценить найденные документы с помощью функции score_documents и определить следующий узел на основе полученного результата.

Далее создается ребро от узла "generate" к узлу END, что означает, что после генерации ответа рабочий процесс завершается.

Последним добавляется ребро от "improve" обратно к "agent", создавая цикл, в котором улучшенный вопрос возвращается агенту для повторной обработки.

Теперь мы готовы скомпилировать граф:

```
graph = workflow.compile()
```

Этот код компилирует граф с помощью workflow.compile() и сохраняет скомпилированную версию графа в переменной graph.

Мы уже визуализировали этот граф на рис. 12.1, но если вы хотите создать визуализацию самостоятельно, используйте код:

```
from IPython.display import Image, display
try:
   display(Image(graph.get_graph(xray=True).draw_mermaid_png()))
except:
   pass
```

Этот код использует IPython для генерации и отображения визуализации графа.

Теперь запускаем нашего агента:

```
import pprint
inputs = {
   "messages": [
      ("user", user_query),
   ]
}
```

Код импортирует модуль pprint, который позволяет форматировать и выводить данные в удобочитаемом виде.

Затем создается словарь inputs, который представляет начальные входные данные для графа: "messages" — ключ, содержащий список кортежей, кортеж ("user", user_query) представляет сообщение, где "user" указывает роль отправителя, а user_query содержит запрос пользователя.

Создаем переменную final_answer, в которой будет храниться окончательный ответ:

final_answer = ''

Запускаем цикл агента, используя экземпляр графа в качестве основы:

```
for output in graph.stream(inputs):
   for key, value in output.items():
      pprint.pprint(f"Output from node '{key}':")
      pprint.pprint("    ")
      pprint.pprint(value, indent=2, width=80, depth=None)
      final_answer = value
```

Этот код запускает двойной цикл, который потоково обрабатывает входные данные с помощью graph.stream(inputs). Внешний цикл перебирает выходные данные, которые генерируются графом во время обработки inputs. Внутренний цикл перебирает пары ключ-значение (key, value) внутри output.items(): key — это название узла, который сгенерировал вывод, value — это результат работы этого узла.

Код выводит название узла, который сгенерировал выходные данные:

pprint.pprint(f"Output from node '{key}':")

Далее выводит сам результат (value) в отформатированном виде с помощью pprint.pprint():

pprint.pprint(value, indent=2, width=80, depth=None)

indent=2 — задает уровень отступа, width=80 — ограничивает ширину вывода, depth=None — не ограничивает глубину вложенности данных.

Переменная final_answer обновляется на каждом шаге, записывая в себя последний результат (выходные данные последнего узла в процессе обработки).

Этот код позволяет наблюдать промежуточные результаты, которые генерируются каждым узлом в графе. Фактически, вывод представляет собой *мысли* агента, отображая процесс принятия решений.

Когда мы запускаем агента и начинаем получать вывод, видно, что происходит множество вычислений!

Пример первых строк вывода:

CALL AGENT

"Output from node 'agent':"

'---'

{ 'messages': [AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0, 'id': 'call_46NqZuz3gN2F9IR5jq0MRdVm', 'function': {'arguments': '{"query":"Google\'s environmental initiatives"}', 'name': 'retrieve_google_environmental_ question_answers'}, 'type': 'function'}]}, response_ metadata={'finish_reason': 'tool_calls'}, id='run-eba27f1e- 1c32-4ffc-a161-55a32d645498-0', tool_calls=[{'name': 'retrieve_google_environmental_question_answers', 'args': {'query': "Google's environmental initiatives"}, 'id': 'call_46NqZuz3gN2F9IR5jq0MRdVm'}])]}

Здесь агент решает использовать инструмент retrieve_google_environmental_question_answers. Это тот самый retriever, который мы определили ранее. Хороший выбор!

Агент оценивает релевантность найденных документов:

CHECK RELEVANCE

DECISION: DOCS RELEVANT

Агент принимает решение, что документы релевантны.

Вывод данных, полученных агентом из PDF-документа и ensemble_retriever (укорочено для удобства):

"Output from node 'retrieve':"

'---'

{ 'messages': [ToolMessage(content='iMasons Climate AccordGoogle is a founding member and part of the governing body of the iMasons Climate Accord, a coalition united on carbon reduction in digital infrastructure.\nReFEDIn 2022, to activate industry-wide change...[TRUNCATED]', tool_call_ id='call_46NqZuz3gN2F9IR5jq0MRdVm')] }

Когда мы просматриваем этот вывод, видно, что найденные данные объединены и предоставляют агенту объемную информацию, с которой он может работать.

Теперь агент, как и в нашем исходном RAG-приложении, берет вопрос, найденные данные и формирует ответ на основе заданного промта генерации:

GENERATE

"Output from node 'generate':"

'---'

{ 'messages': ['Google has a comprehensive and multifaceted approach to '

'environmental sustainability, encompassing various ' 'initiatives aimed at reducing carbon emissions, promoting' 'sustainable practices, and leveraging technology for ' "environmental benefits. Here are some key aspects of Google's " 'environmental initiatives:\n"\n'

'1. **Carbon Reduction and Renewable Energy**.'] }

'\n \n'

Мы добавили механизм для отдельного вывода итогового ответа для удобства чтения:

final_answer['messages'][0]

Результат вывода:

"Google has a comprehensive and multifaceted approach to environmental sustainability, encompassing various initiatives aimed at reducing carbon emissions, promoting sustainable practices, and leveraging technology for environmental benefits. Here are some key aspects of Google's environmental initiatives:\n\n1. **Carbon Reduction and Renewable Energy**:\n - **iMasons Climate Accord**: Google is a founding member and part of the governing body of this coalition focused on reducing carbon emissions in digital infrastructure.\n - **Net-Zero Carbon**: Google is committed to operating sustainably with a focus on achieving net-zero carbon emissions. This includes investments in carbon-free energy and energy-efficient facilities, such as their all-electric, net water-positive Bay View campus..."

Это полный выходной результат работы нашего агента!

Саммари

В этой главе мы рассмотрели, как AI-агенты и LangGraph могут улучшить RAG-приложения, делая их более мощными и сложными. Мы выяснили, что AI-агент — это, по сути, LLM с циклом, который позволяет ему разбивать сложные задачи на более простые шаги, повышая вероятность успешного выполнения RAG-задач.

LangGraph, построенный поверх LCEL, предоставляет возможности для создания настраиваемых агентных рабочих процессов, позволяя разработчикам управлять агентами с помощью графового подхода.

Мы погрузились в основы AI-агентов и их интеграции с RAG, рассмотрели инструменты, которые агенты могут использовать для выполнения задач, а также изучили класс AgentState в LangGraph, который отслеживает состояние агента во времени. Мы также разобрали основные концепции теории графов: узлы, ребра и условные ребра, которые являются ключевыми элементами LangGraph.

В практической части мы создали агента LangGraph для RAG-приложения, продемонстрировав, как разрабатывать инструменты, определять состояние агента, настраивать промты и создавать циклические графы. Мы увидели, как агент использует логические возможности, чтобы определить, какие инструменты использовать, как их применять и какие данные в них передавать, формируя более точный и полный ответ на запрос пользователя.

В следующей главе мы рассмотрим, как инженерия промтов может быть использована для улучшения RAG-приложений.