Глава 14. Продвинутые техники RAG для улучшения результатов

Это продолжение перевода книги Кит Борн. Раскрытие потенциала данных с помощью генеративного ИИ и технологии RAG. В заключительной главе мы рассмотрим несколько продвинутых техник, которые помогут улучшить RAG-приложения. Эти методы выходят за рамки базовых подходов RAG, позволяя решать более сложные задачи и добиваться лучших результатов. Мы начнем с техник, которые уже использовали ранее, рассмотрим их ограничения, а затем добавим новые методы, которые позволят преодолеть эти недостатки и вывести RAG на новый уровень. В ходе этой главы вы получите практический опыт, реализуя продвинутые техники через лабораторные работы.

Темы главы:

- Наивный RAG и его ограничения
- Гибридный RAG / мультивекторный RAG для улучшенного поиска
- Переранжирование (re-ranking) в гибридном RAG
- Лаборатория кода 14.1 — Расширение запроса (Query Expansion)
- Лаборатория кода 14.2 — Декомпозиция запроса (Query Decomposition)
- Лаборатория кода 14.3 — Мультимодальный RAG (MM-RAG)
- Другие продвинутые техники RAG

Эти методы улучшают процесс поиска и генерации, расширяя запросы, разбивая вопросы на подзадачи и добавляя мультимодальные данные. Мы также рассмотрим индексацию, поиск, генерацию и весь конвейер RAG.

Предыдущая глава Содержание Следующая глава

Код для этой главы размещен в репозитории GitHub.

Начнем с обсуждения наивного RAG — основного подхода, который мы изучали в главе 2 и с которым вы уже должны быть хорошо знакомы.

Наивный RAG и его ограничения

До этого момента мы работали с тремя типами RAG-подходов: наивный RAG, гибридный RAG и переранжирование. Изначально мы использовали наивный RAG — базовый вариант RAG, который мы внедрили в главе 2 и использовали в нескольких лабораторных работах. Наивные RAG-модели стали первым шагом в развитии технологии RAG, предоставляя основную архитектуру для объединения поиска и генерации. Однако у них есть ограничения с точки зрения гибкости и масштабируемости.

Наивный RAG работает, извлекая фрагментированные части контекста — текстовые блоки, которые затем векторизуются и передаются в контекстное окно LLM. Если размер фрагментов слишком мал, это приводит к сильной фрагментации контекста, что ухудшает понимание текста и семантики. Это снижает эффективность поиска информации, делая генерацию менее точной. Наивный RAG обычно использует только семантический поиск, что делает его уязвимым перед этими ограничениями. Поэтому мы внедрили более продвинутый метод извлечения информации — гибридный поиск.

Гибридный RAG / мультивекторный RAG для улучшенного поиска

Гибридный RAG расширяет концепцию наивного RAG, используя несколько векторов вместо одного векторного представления для запросов и документов. Мы подробно изучили гибридный RAG в главе 8, реализовав его как с использованием встроенных механизмов LangChain, так и вручную, чтобы лучше понять его внутреннюю логику. Гибридный или мультивекторный RAG может сочетать семантический поиск, поиск по ключевым словам, другие векторные техники, подходящие для конкретного приложения

В нашей лабораторной работе гибридный RAG включал поиск по ключевым словам. Этот подход особенно полезен, когда контекст слабый (например, имена, коды, внутренние аббревиатуры). Мультивекторный RAG анализирует разные аспекты запроса, что повышает точность и релевантность извлеченной информации. Области применения мультивекторного RAG: техническая документация, научные исследования, внутренние корпоративные базы знаний с кодами и сущностями, сложные системы вопрос-ответ. Но мультивекторный RAG — не единственный продвинутый метод, который мы внедрили в главе 8. Мы также применили переранжирование (re-ranking).

Переранжирование в гибридном RAG

После выполнения семантического поиска и поиска по ключевым словам результаты ранжируются заново. Этот метод учитывает, какие результаты были найдены в обоих поисках, а также их первоначальный рейтинг. Таким образом, вы уже освоили три техники RAG, включая две продвинутые! Что дальше?

В этой главе мы добавим еще три продвинутых метода:

- Расширение запроса (Query Expansion)
- Декомпозиция запроса (Query Decomposition)
- Мультимодальный RAG (MM-RAG)

Мы также рассмотрим дополнительные подходы, но именно эти три техники мы выбрали, потому что они находят применение в самых разных RAG-сценариях. Первая лабораторная работа посвящена расширению запроса (Query Expansion).

Лаборатория кода 14.1 — Расширение запроса (Query Expansion)

Код для этой лабораторной работы находится в файле.

Многие техники улучшения RAG сосредоточены на одном аспекте — поиске или генерации. Расширение запроса (Query Expansion) может улучшить оба процесса одновременно. Мы уже рассматривали концепцию расширения в главе 13, но там она касалась выходных данных LLM. Здесь же мы применяем расширение к входным данным, добавляя к оригинальному запросу дополнительные ключевые слова или фразы.

Этот метод усиливает понимание запроса поисковой моделью, так как добавляет контекст, увеличивает вероятность нахождения релевантных документов, обеспечивает более качественный ответ от LLM, так как он получает лучшее исходное представление запроса.

Обычно расширение запроса включает немедленную отправку пользовательского запроса в LLM без стандартного контекста RAG. LLM генерирует начальный ответ, который позволяет расширить область поиска, сохраняя исходный смысл.

Создайте новую ячейку выше ячейки, где создается цепочка rag_chain_from_docs.

Мы добавим набор шаблонов промтов:

```
from langchain.prompts.chat import ChatPromptTemplate,
  HumanMessagePromptTemplate, SystemMessagePromptTemplate
```

Давайте разберем шаблоны промтов и их назначение.

- ChatPromptTemplate — шаблон для чатовых промтов, который позволяет объединять другие промты в единую структуру.
- HumanMessagePromptTemplate — шаблон для сообщений от пользователя. HumanMessage представляет сообщение, отправленное человеком. В данном случае LLM получает user_query, который подставляется в этот шаблон.
- SystemMessagePromptTemplate — шаблон для системных сообщений. В чатовых моделях системные промты играют отдельную роль от промтов пользователя. Этот шаблон определяет инструкцию для модели, задавая ей роль.

Теперь создадим функцию, которая будет управлять расширением запроса с использованием шаблонов промтов. Системное сообщение в этой функции будет выглядеть так (его можно адаптировать под тематику RAG-приложения — в данном случае экологические отчеты):

> *Вы — эксперт-ассистент по исследованиям в области экологии. Предоставьте пример ответа на заданный вопрос, который мог бы содержаться в документе, например в ежегодном экологическом отчете.*

Это будет первым шагом в функции:

```
def augment_query_generated(user_query):
  system_message_prompt = SystemMessagePromptTemplate.from_template(
    "You are a helpful expert environmental research assistant. Provide an example answer to the given question, that might be found in a document like an annual environmental report."
```

```
    )
    human_message_prompt = HumanMessagePromptTemplate.from_template("{query}")
    chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt,
human_message_prompt])
    response = chat_prompt.format_prompt(query=user_query).to_messages()
    result = llm(response)
    content = result.content
    return content
```

Здесь мы объединяем три типа шаблонов промтов, формируя общий набор сообщений, который отправляется в LLM. В результате LLM пытается самостоятельно ответить на вопрос, обогащая исходный запрос дополнительными деталями.

Добавим код, который вызывает эту функцию, чтобы проанализировать результат, представляющий расширенный запрос:

```
original_query = "What are Google's environmental initiatives?"
hypothetical_answer = augment_query_generated(original_query)
joint_query = f"{original_query} {hypothetical_answer}"
print(joint_query)
```

В этом фрагменте кода исходный запрос пользователя называется original_query, указывая, что он является основным источником, который затем пройдет процесс расширения. hypothetical_answer — это строка ответа, которую возвращает LLM. Затем исходный запрос объединяется с этим гипотетическим ответом в строку joint_query, которая и используется в качестве нового запроса.

Вывод будет выглядеть следующим образом (сокращен для краткости):

> Каковы экологические инициативы Google?
>
> В 2022 году Google продолжил развивать свои экологические инициативы, сосредоточившись на устойчивом развитии и сокращении углеродного следа. Ключевые инициативы включают:
>
> 1. Углеродная нейтральность и возобновляемая энергия: Google сохраняет статус углеродно-нейтральной компании с 2007 года и ставит цель работать на 100% безуглеродной энергии 24/7 к 2030 году. В 2022 году Google закупил более 7 гигаватт возобновляемой энергии, сделав значительный шаг к достижению этой цели.
> 2. Энергоэффективность дата-центров: Дата-центры Google — одни из самых энергоэффективных в мире. В 2022 году компания достигла среднего коэффициента эффективности энергопотребления (PUE) на уровне 1.10, что значительно ниже среднеотраслевого показателя. Это стало возможным благодаря передовым технологиям охлаждения и системам управления энергопотреблением на основе ИИ.
> 3. Устойчивые продукты и услуги...

Ответ стал намного длиннее, и LLM действительно постаралась дать развернутый ответ. Этот начальный ответ является гипотетическим или воображаемым ответом на исходный пользовательский запрос. Обычно RAG-методы избегают подобных ответов, но в этом случае они помогают извлечь из LLM ключевые концепции, соответствующие исходному запросу пользователя.

Теперь изменим исходный код: вместо передачи строки original_query, как раньше, мы передадим объединенный запрос (исходный + воображаемый ответ) в наш RAG-конвейер:

```
# Query expansion with an answer RAG results
result_alt = rag_chain_with_source.invoke(joint_query)
retrieved_docs_alt = result_alt['context']

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

```
print(f"Content:\n{doc.page_content}\n")
```

Видно, что запрос, передаваемый в RAG-пайплайн, теперь значительно длиннее (joint_query), и в результате мы получаем более богатый набор данных, объединяющий предоставленную информацию с расширенным контекстом, который добавил LLM. Так как LLM возвращает текст в формате Markdown, можно использовать IPython для красивого форматирования вывода. Этот код дает следующий результат:

```
from IPython.display import Markdown, display
markdown_text_alt = result_alt['answer']['final_answer']
display(Markdown(markdown_text_alt))
```

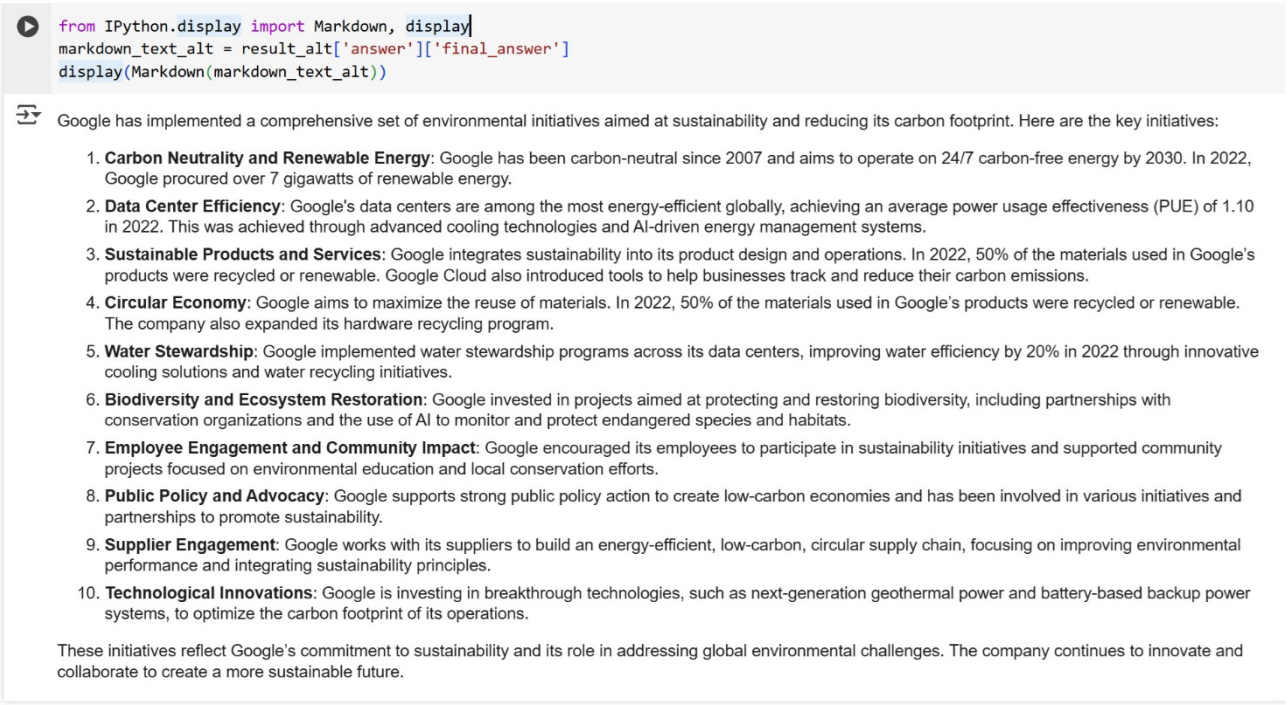

Рис. 14.1. Вывод обогащенного запроса

Сравните этот результат с ответами, которые мы получили с исходного запроса, и решите, улучшился ли ответ! Как видно, каждый раз получается немного другой ответ, и вы можете определить, какой вариант лучше подходит для вашего RAG-приложения.

Один из важных аспектов этого метода заключается в том, что теперь LLM используется не только на этапе генерации, но и на этапе извлечения данных. Ранее LLM применялся только в стадии генерации, но теперь инженерия промтов становится важной уже на этапе поиска. Однако этот процесс аналогичен тому, что мы обсуждали в главе 13, посвященной инженерии промтов, где речь шла об итерациях, необходимых для улучшения результатов работы модели.

Дополнительную информацию о расширении запроса можно найти в оригинальной научной статье.

Расширение запроса — это лишь один из множества подходов, которые позволяют улучшить исходный запрос и, как следствие, повысить качество RAG-выходных данных. В конце этой главы мы перечислим другие методы, но в следующей лабораторной работе рассмотрим технику, называемую декомпозицией запроса, которая особенно полезна в RAG-сценариях, ориентированных на поиск ответов на вопросы.

Лаборатория кода 14.2 — Декомпозиция запроса (Query Decomposition)

Код для этой лабораторной работы находится в файле.

Декомпозиция запроса — это стратегия, направленная на улучшение процессов поиска ответов в системах генеративного ИИ. Этот метод относится к категории преобразования запроса (query translation) — набора подходов, сосредоточенных на оптимизации первой стадии конвейера RAG, то есть извлечения данных. При декомпозиции запроса мы разбиваем исходный вопрос на несколько более мелких вопросов. Эти вопросы могут обрабатываться последовательно или параллельно, в зависимости от требований задачи, что делает метод гибким и адаптивным к различным сценариям

использования RAG. После обработки всех подзапросов выполняется этап консолидации, который формирует финальный ответ, часто обладающий более широким охватом, чем результат наивного RAG.

Существуют и другие методы преобразования запросов, такие как объединение RAG (RAG-Fusion) и мультизапрос (multi-query), которые также работают с подзапросами, но в этой лабораторной работе мы сосредоточимся на разбиении вопроса.

В статье, предложившей этот метод, написанной исследователями Google, он называется Least-to-Most (от простого к сложному), или декомпозиция. В документации LangChain этот метод описывается как query decomposition, так что мы опираемся на авторитетные источники, обсуждая этот подход.

Перед тем как перейти к коду, разберем две ключевые концепции, которые помогут понять реализацию декомпозиции запроса:

- Chain-of-thought (CoT) — метод проектирования промтов, в котором структура входного запроса имитирует человеческое логическое мышление. Цель — повысить качество работы модели в задачах, требующих логики, вычислений и принятия решений.
- Interleaving retrieval — метод, при котором процесс поиска и CoT-промты чередуются друг с другом. Такой подход позволяет получить более релевантные данные на каждом этапе обработки, по сравнению с прямым извлечением информации только по пользовательскому запросу. Эта комбинация называется Interleave Retrieval with CoT (IR-CoT).

Этот метод разбивает проблему на подзадачи и пошагово извлекает данные, формируя динамический процесс поиска. После разбиения пользовательского запроса на подвопросы выполняется поиск документов по первому вопросу, затем формируется ответ. Далее проводится новый поиск для второго вопроса, но теперь с учетом данных, полученных из первого шага. Этот процесс продолжается для всех подзапросов, пока не будет сформирован финальный ответ.

Теперь, когда мы разобрались с теорией, давайте перейдем к коду!

Сначала импортируем несколько новых пакетов:

from langchain.load import dumps, loads

Функции dumps и loads, импортируемые из langchain.load, используются для сериализации и десериализации объектов Python. В нашем коде они помогут конвертировать объекты Document в строковый формат перед удалением дубликатов, а затем преобразовывать их обратно.

Остаьвте определения retriever без изменения и добавьте новую ячейку, в которой мы создадим шаблон промта для декомпозиции, цепочку обработки и код для ее выполнения. Начнем с создания нового шаблона промта:

```
# Prompt LLM to decompose query for broader coverage
prompt_decompose = PromptTemplate.from_template(
   """You are an AI language model assistant.
   Your task is to generate five different versions of the given
   user query to retrieve relevant documents from a vector search.
   By generating multiple perspectives on the user question,
   your goal is to help the user overcome some of the limitations
   of the distance-based similarity search.
   Provide these alternative questions separated by newlines.
   Original question: {question}"""
)
```

```
# Prompt LLM to decompose query for broader coverage
prompt_decompose = PromptTemplate.from_template(
   """Ты — языковая модель искусственного интеллекта.
   Твоя задача — сгенерировать пять различных вариантов данного
   пользовательского запроса, чтобы наиболее эффективно находить
   релевантные документы в векторном поиске.
   Генерируя разные формулировки одного и того же вопроса,
```

```
    ты помогаешь пользователю обойти ограничения поиска,
    основанного на метрике расстояния. Представь альтернативные
    варианты запросов, разделяя их переносами строк.
    Original question: {question}"""
)
```

Читая строку в этом объекте PromptTemplate, мы получаем версию промта, объясняющую LLM, как выполнить требуемую нами декомпозицию. Это прозрачный запрос к модели, в котором мы четко формулируем проблему, которую пытаемся решить, и указываем, что нам нужно от нее. Мы также просим модель вернуть результат в определенном формате. Это может быть рискованно, так как LLM иногда выдает неожиданные ответы, даже если явно задан определенный формат. В более надежных приложениях на этом этапе стоит добавить проверку, чтобы убедиться, что ответ имеет правильную структуру. Однако в данном примере модель ChatGPT-4o-mini хорошо справляется с заданием и возвращает данные в нужном формате.

Далее мы настраиваем цепочку обработки, используя стандартные элементы, но применяя наш промт для декомпозиции:

```
decompose_queries_chain = (
    prompt_decompose
    | llm
    | str_output_parser
    | (lambda x: x.split("\n"))
)
```

Эта цепочка вполне очевидна: она использует шаблон промта, LLM, определенный ранее в коде, парсер вывода, а затем применяет форматирование для более удобного чтения результата. Чтобы вызвать эту цепочку, выполняем следующий код:

```
# Invoke decompose_queries_chain and print the five different versions
decomposed_queries = decompose_queries_chain.invoke({"question": user_query})
print("Five different versions of the user query:")
print(f"Original: {user_query}")
for i, question in enumerate(decomposed_queries, start=1):
    print(f"{question.strip()}")
```

Этот вызов запускает цепочку и возвращает исходный запрос пользователя, а также пять новых вариантов запроса, сгенерированных нашим промтом для декомпозиции и моделью:

> Оригинал: *Какие экологические инициативы реализует Google?*
>
> Пять различных версий пользовательского запроса:
>
> 1. *Какие шаги предпринимает Google для решения экологических проблем?*
> 2. *Как Google способствует устойчивому развитию окружающей среды?*
> 3. *Можете перечислить экологические программы и проекты, в которых участвует Google?*
> 4. *Какие меры Google внедрил для сокращения своего воздействия на окружающую среду?*
> 5. *Каковы основные экологические стратегии и цели Google?*

LLM отлично справляется с преобразованием исходного запроса, разбивая его на несколько связанных вопросов, охватывающих различные аспекты проблемы. Но это лишь половина процесса декомпозиции! Теперь мы будем использовать все сгенерированные вопросы для поиска документов, что обеспечит гораздо более широкий набор релевантного контекста по сравнению с тем, что мы получали в предыдущих лабораторных работах.

Сначала создадим функцию для форматирования документов, полученных в результате поиска по этим новым запросам:

```
def format_retrieved_docs(documents: list[list]):
    flattened_docs = [dumps(doc) for sublist in documents for doc in sublist]
    print(f"FLATTENED DOCS: {len(flattened_docs)}")
    deduped_docs = list(set(flattened_docs))
```

```
    print(f"DEDUPED DOCS: {len(deduped_docs)}")
    return [loads(doc) for doc in deduped_docs]
```

Эта функция возвращает список списков, где каждый вложенный список содержит найденные документы. Затем мы объединяем их в один общий список, сериализуем объекты Document в строки с помощью dumps, удаляем дубликаты, а затем восстанавливаем их в виде списка. Также мы выводим количество документов до и после удаления дубликатов. В данном примере из 100 документов после обработки осталось 67:

```
FLATTENED DOCS: 100
DEDUPED DOCS: 67
```

Теперь создадим цепочку, которая объединит нашу предыдущую цепочку decompose_queries_chain, выполнит поиск по всем новым запросам и отформатирует полученные результаты:

```
retrieval_chain = (
    decompose_queries_chain
    | ensemble_retriever.map()
    | format_retrieved_docs
)
```

Несмотря на свою лаконичность, эта строка кода выполняет огромный объем работы! В итоге у нас получается 67 документов, относящихся ко всем вопросам, которые были сгенерированы на основе оригинального запроса и его декомпозиции. Обратите внимание, что decompose_queries_chain теперь встроена в эту цепочку, поэтому нет необходимости запускать ее отдельно.

Мы сохраняем результаты работы этой цепочки в переменную docs с помощью следующего кода:

```
# We retrieve a significant number of documents compared to previous methods
docs = retrieval_chain.invoke({"question":user_query})
```

После выполнения этой цепочки мы получаем значительно больше документов, чем при предыдущих методах. Однако нам все еще нужно выполнить финальные шаги RAG с расширенным набором извлеченных данных. Большая часть кода остается неизменной, но мы заменяем ensemble chain на новую цепочку retrieval_chain, которую только что построили:

```
# replace ensemble_retriever with retrieval_chain
rag_chain_with_source = RunnableParallel(
    {"context": retrieval_chain, "question": RunnablePassthrough()}
).assign(answer=rag_chain_from_docs)
```

Этот код интегрирует нашу новую логику в предыдущее RAG-приложение. Выполнив эту строку, мы запускаем все созданные цепочки одновременно, без необходимости запускать их по отдельности, как это было в примерах раньше. Теперь у нас есть единый, связный процесс, объединяющий ранние методы с этой новой мощной техникой RAG.

Сравните текущие результаты, полученные с этим методом, с результатами предыдущих лабораторных работ, чтобы оценить, насколько лучше покрываются детали и расширяется охват запрашиваемых тем.

> Google реализует широкий спектр экологических инициатив, направленных на устойчивое развитие и снижение воздействия на окружающую среду.
> Вот ключевые инициативы, основанные на предоставленном контексте:
>
> 1. Восстановление кампусов и природных зон:
>   Google восстановил более 40 акров природных территорий в кампусах и городских ландшафтах, в основном в районе залива Сан-Франциско. Это включает высадку около 4000 деревьев и восстановление экосистем, таких как дубовые леса, ивовые рощи и водно-болотные угодья.
> 2. Использование безуглеродной энергии:
>   Google стремится достичь нулевых выбросов углерода и перейти на 100% безуглеродную энергетику (CFE) к 2030 году. Для этого компания инвестирует в закупку чистой энергии, технологические инновации и политику регулирования. В

2022 году Google разработал дорожную карту по CFE и активно продвигает инициативы по декарбонизации электросетей по всему миру.

3. Управление водными ресурсами...

Использование продвинутых техник RAG, таких как этот метод, может значительно улучшить качество извлеченных данных в зависимости от целей вашего приложения. Дополнительную информацию о данном подходе можно найти в оригинальной научной статье.

В следующей лабораторной работе мы выйдем за пределы текстовых данных и добавил работу с изображениями и видео с помощью техники, называемой MM-RAG.

Лаборатория кода 14.3 – Мультимодальный RAG

Код для этой лабораторной работы можно найти в файле.

Это отличный пример того, как сокращения могут значительно упростить обсуждение сложных понятий. Попробуйте произнести вслух *мультимодальная генерация дополненная поиском* (multi-modal retrieval augmented generation) хотя бы раз, и вы, скорее всего, захотите использовать MM-RAG вместо полного названия! Это новаторский подход, который, вероятно, получит широкое распространение в ближайшем будущем. Он лучше отражает то, как люди воспринимают и обрабатывают информацию, а значит, должен быть очень эффективным. Давайте начнем с рассмотрения концепции работы с несколькими режимами данных.

Мультимодальность

До этого момента мы работали исключительно с текстом: принимали текст в качестве ввода, выполняли поиск по текстовым данным и передавали найденный текст в LLM, который затем генерировал окончательный текстовый вывод. Но что насчет других типов данных? Компании, разрабатывающие LLM, уже начали предлагать мощные мультимодальные возможности. Как мы можем интегрировать эти возможности в RAG-приложение?

Мультимодальность означает работу с несколькими формами представления информации — текстом, изображениями, видео, аудио и другими типами данных. Мультимодальность может присутствовать как во входных данных, так и в выходных. Например, если вы передаете текст и получаете изображение, это мультимодальный процесс. Если вы передаете изображение и получаете текст (так называемая «генерация подписей»), это тоже мультимодальность.

Более сложные подходы могут включать передачу текста *«преобразуй это изображение в видео, продолжив движение по водопаду, добавив звук падающей воды»* и самого изображения водопада, а на выходе получить видео, которое погружает пользователя в этот водопад, со звуковым сопровождением. Это будет включать четыре разных режима: текст, изображение, видео и аудио. Поскольку модели с такими возможностями уже существуют и предлагают API, похожие на те, что мы использовали в этой книге, логично рассмотреть, как их можно применить в RAG, используя этот подход для работы с различными видами контента, хранящегося в корпоративных базах данных. Рассмотрим преимущества мультимодального подхода.

Преимущества мультимодальности

Этот подход усиливает возможности RAG в обработке мультимодальных данных, позволяя создавать более содержательные, информативные и контекстно насыщенные результаты. Интеграция мультимодальных данных делает RAG-системы более точными, расширяет контекст, а также повышает уровень взаимодействия с пользователями. Области применения включают улучшенные диалоговые агенты, которые могут понимать и генерировать мультимедийные ответы, а также продвинутые инструменты создания контента, способные разрабатывать сложные документы и презентации. MM-RAG делает RAG-приложения более универсальными и приближает их к тому, как люди воспринимают окружающий мир через разные сенсорные каналы.

Как и в обсуждениях о векторных представлениях данных в главах 7 и 8, здесь тоже важно понимать роль векторных эмбеддингов в MM-RAG.

Мультимодальные векторные эмбеддинги

MM-RAG возможен благодаря тому, что векторные эмбеддинги могут представлять не только текст, но и другие виды данных. Некоторые данные требуют предварительной обработки перед векторизацией, но в целом любой тип информации может быть преобразован в векторную форму и

использован в RAG-приложении. Если вспомнить, векторизация сводится к превращению данных в математическое представление, а именно математика и векторы являются основным языком глубинного обучения (deep learning, DL), на котором базируются все RAG-модели.

Еще одна важная идея, связанная с векторами, — векторное пространство, где похожие концепции располагаются ближе друг к другу, а несвязанные — дальше. В мультимодальном контексте это правило остается в силе. Например, понятие *чайка* должно быть представлено схожим образом, независимо от того, идет ли речь о слове *чайка*, изображении чайки, видео с ней или аудиозаписи ее крика. Этот мультимодальный принцип сохранения смысловой близости называется модальной независимостью (modality independence). Эта концепция формирует основу MM-RAG, который выполняет те же задачи, что и обычный RAG, но с разными типами данных.

Изображения — это не просто «картинки»

Изображения в корпоративной среде могут представлять намного больше, чем просто красивые пейзажи или фотографии с отдыха. В бизнес-контексте изображения — это диаграммы, схемы, инфографика, текстовые документы, представленные в графическом формате, и многое другое. Они являются важным источником данных.

Если вы еще не открывали PDF-файл с Google Environmental Report 2023, который мы использовали в лабораторных работах, вам могло показаться, что он содержит только текст. Однако, если заглянуть в него, можно увидеть, что документ полон иллюстраций. Многие из диаграмм, особенно сложные и детализированные, представлены в виде изображений. Как можно создать RAG-приложение, которое также использует эти данные? Давайте разберемся!

Введение в MM-RAG на практике

В этой лабораторной работе мы реализуем следующие шаги:

1. Извлечение текста и изображений из PDF с помощью мощного open-source инструмента unstructured.
2. Использование мультимодального LLM для генерации текстовых описаний на основе извлеченных изображений.
3. Векторизация и поиск по этим описаниям вместе с текстами, которые мы уже обрабатывали ранее.
4. Сохранение в Chroma текстов, изображений и их описаний, используя мультимодальный векторный поиск.
5. Объединение всех данных — передача текстов и изображений в мультимодальную LLM для формирования финального ответа.

Начнем с установки новых пакетов, необходимых для оптического распознавания символов (OCR) в unstructured:

```
# new packages to download for this code lab!
%pip install "unstructured[pdf]"
%pip install pillow==10.4.0
%pip install pydantic==2.9.2
%pip install lxml==5.3.0
%pip install matplotlib==3.9.2
%pip install tiktoken==0.7.0
!sudo apt-get -y install poppler-utils
!sudo apt-get -y install tesseract-ocr
```

Подробнее о том, какие функции выполняют эти библиотеки в нашем коде:

- unstructured[pdf]: предназначена для извлечения структурированной информации из неструктурированных данных, таких как PDF, изображения и HTML-страницы. В данном случае устанавливается только поддержка PDF. Если требуется обработка других типов документов, можно добавить соответствующие модули или установить unstructured[all], чтобы поддерживать все доступные форматы.
- pillow: ответвление библиотеки Python Imaging Library (PIL). Pillow используется для открытия, редактирования и сохранения изображений в различных форматах. В коде мы работаем с изображениями при использовании unstructured, а pillow помогает управлять этими процессами.

- pydantic: выполняет валидацию данных и управляет настройками, используя аннотации типов Python. Pydantic широко применяется для определения моделей данных и проверки входных данных.
- lxml: обрабатывает XML и HTML. Используется вместе с unstructured для парсинга и извлечения информации из структурированных документов.
- matplotlib: одна из самых популярных библиотек для визуализации данных в Python.
- tiktoken: токенизатор Byte-Pair Encoding (BPE), разработанный OpenAI. BPE изначально создавался как алгоритм сжатия текста, а затем был адаптирован для токенизации данных при предобучении GPT-моделей.
- poppler-utils: набор инструментов для работы с PDF через командную строку. В коде poppler используется unstructured для извлечения элементов из PDF-файла.
- tesseract-ocr: открытый OCR-движок для распознавания и извлечения текста из изображений. Tesseract также требуется unstructured для обработки PDF-документов с изображениями, содержащими текст.

Эти пакеты обеспечивают необходимые функции и зависимости для langchain, unstructured и других используемых библиотек. Они позволяют парсить PDF, обрабатывать изображения, проверять данные, выполнять токенизацию и OCR, что критически важно для анализа документов и генерации ответов на пользовательские запросы.

Теперь добавим импорт этих и других пакетов, чтобы использовать их в коде.

```
# new
from langchain.retrievers.multi_vector import MultiVectorRetriever
from langchain_community.document_loaders import UnstructuredPDFLoader
from langchain_core.runnables import RunnableLambda
from langchain.storage import InMemoryStore
from langchain_core.messages import HumanMessage
import base64
import uuid
from IPython.display import HTML, display
from PIL import Image
import matplotlib.pyplot as plt
```

Это длинный список Python-пакетов, поэтому разберем их по порядку:

MultiVectorRetriever – ретривер, который объединяет несколько векторных хранилищ и позволяет эффективно извлекать документы на основе поиска по схожести. Мы используем его для создания ретривера, комбинирующего vectorstore и docstore, чтобы находить релевантные документы на основе запроса пользователя.

UnstructuredPDFLoader – загрузчик документов, который извлекает элементы, включая текст и изображения, из PDF-файла с помощью библиотеки unstructured.

RunnableLambda – утилитарный класс, позволяющий обернуть функцию в компонент, который можно выполнить в конвейере LangChain. Используется для оборачивания функций split_image_text_types и img_prompt_func в виде компонентов RAG-конвейера.

InMemoryStore – хранилище в оперативной памяти, которое хранит пары ключ-значение. Используется как хранилище документов, в котором сохраняется фактическое содержимое документа, связанное с его идентификатором (document ID).

HumanMessage – сообщение, отправленное пользователем в диалоге с языковой моделью. Используется для создания сообщений в промптах для описания изображений.

base64 – используется для кодирования изображений в base64-строки, чтобы их можно было хранить и извлекать.

uuid – предоставляет функции для генерации уникальных идентификаторов (UUIDs). Используется для создания уникальных идентификаторов документов, которые добавляются в vectorstore и docstore.

HTML и display – HTML используется для создания HTML-представлений объектов, а display – для их отображения в IPython Notebook. В коде HTML и display используются в plt_img_base64, чтобы показывать изображения, закодированные в base64.

Image – предоставляет функции для открытия, обработки и сохранения изображений в различных форматах.

matplotlib – библиотека для визуализации данных. В коде plt напрямую не используется, но может применяться другими библиотеками или функциями.

Эти пакеты и модули обеспечивают загрузку, извлечение, хранение документов, обмен сообщениями, обработку изображений и визуализацию данных. Они используются в коде для анализа PDF-файлов и генерации ответов на пользовательские запросы.

После импорта мы создаем несколько переменных, которые используются в коде:

GPT-4o-mini: Мы будем использовать GPT-4o-mini, где буква o в названии означает omni, что указывает на его мультимодальность.

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

Укороченная версия PDF: Обратите внимание, что мы используем другой файл:

short_pdf_path = "google-2023-environmental-report-short.pdf"

Полный файл слишком большой, а его использование увеличило бы стоимость обработки без значительных преимуществ для демонстрации. Поэтому мы рекомендуем использовать этот укороченный файл, который позволит нам протестировать MM-RAG, снизив при этом затраты на вычисления.

OpenAI embeddings: Здесь есть важный нюанс, который стоит отметить:

embedding_function = OpenAIEmbeddings()

Эта модель эмбеддингов не поддерживает мультимодальные эмбеддинги, то есть не сможет сопоставить изображение чайки и слово чайка как мультимодальная модель эмбеддингов. Чтобы обойти этот недостаток, мы создаем эмбеддинги описания изображения, а не самого изображения. Это все еще считается мультимодальным подходом, но стоит следить за развитием мультимодальных эмбеддингов, которые позволят решать эту задачу уже на уровне эмбеддингов.

Теперь загрузим PDF с помощью UnstructuredPDFLoader:

```
# Extract elements from PDF using LangChain and Unstructured - can take a little time to load!
pdfloader = UnstructuredPDFLoader(
   short_pdf_path,
   mode="elements",
   strategy="hi_res",
   extract_image_block_types=["Image","Table"],
   extract_image_block_to_payload=True, # converts images to base64 format
)
pdf_data = pdfloader.load()
```

Здесь мы извлекаем элементы из PDF-файла, используя LangChain и unstructured. Этот процесс может занять от 1 до 5 минут в зависимости от мощности вашей среды разработки. Это хороший момент, чтобы сделать перерыв и почитать про параметры, которые позволяют этому пакету работать так, как нам нужно!

Давайте разберем параметры, которые мы использовали в загрузчике документов, и как они помогают нам в этой лабораторной работе:

short_pdf_path: Это переменная, содержащая путь к файлу, который мы определили ранее. Она указывает на укороченную версию нашего PDF-файла.

mode="elements": Этот аргумент задает режим извлечения данных в UnstructuredPDFLoader. Использование mode="elements" заставляет загрузчик извлекать отдельные элементы из PDF, такие

как текстовые блоки и изображения. Этот режим обеспечивает более детальный контроль над извлеченным контентом по сравнению с другими режимами.

strategy="hi_res": Этот аргумент определяет стратегию извлечения элементов из PDF. Доступны и другие варианты: auto, fast и ocr_only. Стратегия "hi_res" анализирует макет документа и использует его для получения дополнительной информации об элементах. Если вам нужно значительно ускорить процесс, попробуйте fast, но качество извлечения будет хуже, чем у "hi_res". Мы рекомендуем протестировать все стратегии, чтобы увидеть разницу.

extract_image_block_types=["Image", "Table"]: Этот параметр указывает, какие типы элементов следует извлекать при обработке изображений. Они сохраняются в виде base64-кодированных данных в метаданных. Здесь мы нацеливаемся на изображения и таблицы.

extract_image_block_to_payload=True: Этот параметр определяет, следует ли включать извлеченные изображения в payload в виде base64-кодированных данных. Это важно при обработке документов с изображениями, используя стратегию "hi_res". Мы устанавливаем True, чтобы не хранить изображения как файлы — загрузчик сам преобразует их в base64 и добавит в метаданные соответствующих элементов.

Когда процесс завершится, все данные из PDF будут загружены в pdf_data. Теперь добавим код, чтобы изучить загруженные данные:

```
texts = [doc for doc in pdf_data if doc.metadata["category"] == "NarrativeText"]
images = [doc for doc in pdf_data if doc.metadata["category"] == "Image"]
print(f"TOTAL DOCS USED BEFORE REDUCTION: texts: {len(texts)} images: {len(images)}")
categories = set(doc.metadata["category"] for doc in pdf_data)
print(f"CATEGORIES REPRESENTED: {categories}")
```

Здесь мы выбираем две наиболее важные категории элементов для этой лабораторной работы: NarrativeText и Image. С помощью генераторов списков мы извлекаем эти элементы и сохраняем их в отдельных переменных. Перед тем как уменьшить количество изображений для снижения затрат на обработку, мы выводим их начальное количество, чтобы убедиться, что все работает правильно! Мы также проверяем, сколько различных типов элементов представлено в данных. Вот вывод:

```
TOTAL DOCS USED BEFORE REDUCTION: texts: 78 images: 17
CATEGORIES REPRESENTED: {'ListItem', 'Title', 'Footer',
'Image', 'Table', 'NarrativeText', 'FigureCaption', 'Header',
'UncategorizedText'}
```

Сейчас у нас 17 изображений. Для демонстрации мы сократим их количество, поскольку будем использовать LLM для их суммаризации, а обработка трех изображений обходится примерно в шесть раз дешевле, чем 17! Мы также видим, что в данных присутствуют различные элементы, а не только NarrativeText. Если бы мы разрабатывали более сложное приложение, мы могли бы включить такие элементы, как Title, Footer, Header и другие в контекст, который передается LLM, а также указывать, на что делать акцент. Например, можно повысить значимость элементов Title. Библиотека unstructured отлично справляется с подготовкой PDF-данных в формате, удобном для LLM!

Как и обещали, уменьшаем количество изображений для снижения затрат на обработку:

```
# cost savings: keep only the first 3 images to save compute costs for summarization
if len(images) > 3:
    images = images[:3]
print(f"total documents after reduction: texts: {len(texts)} images: {len(images)}")
```

Мы просто отсекаем первые три изображения и используем этот сокращенный список в переменной images. Проверяем:

```
total documents after reduction: texts: 78 images: 3
```

Следующие блоки кода сосредоточены на суммаризации изображений, начиная с функции, которая применяет промпт к изображению и получает его описание:

```
def apply_prompt(img_base64):
    # Prompt
```

```
    prompt = """You are an assistant tasked with summarizing images for retrieval. \
    These summaries will be embedded and used to retrieve the raw image. \
    Give a concise summary of the image that is well optimized for retrieval."""

    return [HumanMessage(content=[
        {"type": "text", "text": prompt},
        {"type": "image_url","image_url": {"url": f"{img_base64}"},},
    ])]
```

Эта функция принимает параметр img_base64, представляющий base64-кодированное изображение. Внутри функции мы задаем переменную prompt, содержащую инструкцию для ассистента с просьбой создать краткое описание изображения. Функция возвращает список с одним объектом HumanMessage, представляющим суммарное описание изображения.

Объект HumanMessage содержит параметр content, который представляет собой список из двух словарей. Первый содержит текстовое сообщение с промптом в качестве значения. Второй представляет изображение в формате URL, где ключ image_url содержит словарь с ключом url, в который передается base64-кодированное изображение, дополненное префиксом data:image/jpeg;base64 для корректного отображения.

Помните, что мы установили extract_image_block_to_payload=True в загрузчике UnstructuredPDFLoader? Благодаря этому изображения уже находятся в метаданных в формате base64, поэтому нам просто нужно передать их в эту функцию! Если вы применяете этот метод в других приложениях и работаете с обычными изображениями (.jpg или .png), вам нужно сначала конвертировать их в base64, прежде чем использовать эту функцию.

В этом приложении LLM работает именно с base64-изображениями, поэтому нам не нужно физически загружать файлы изображений! Вы разочарованы, что не увидите изображения? Не переживайте! Вскоре мы создадим вспомогательную функцию, используя HTML, чтобы отобразить изображения в ноутбуке в удобном формате!

Но сначала мы подготавливаем текстовые и графические данные и создаем списки для сбора их суммарных описаний:

```
# Just using the existing text as text summaries to save money,
# but you can add summaries here too in more robust applications
text_summaries = [doc.page_content for doc in texts]

# Store base64 encoded images, image summaries
img_base64_list = []
image_summaries = []

# Apply to images
for img_doc in images:
    base64_image = img_doc.metadata["image_base64"]
    img_base64_list.append(base64_image)
    message = llm.invoke(apply_prompt(base64_image))
    image_summaries.append(message.content)
```

Обратите внимание, что мы не выполняем суммаризацию текстов — мы просто используем их напрямую в качестве резюме. Вы также можете суммаризировать тексты, что может улучшить результаты поиска, так как это распространенный метод оптимизации извлечения в RAG. Однако, чтобы сэкономить на вычислительных затратах, мы сосредоточились только на суммаризации изображений. Ваш кошелек нас поблагодарит!

Но что касается изображений — вот оно, мультимодальное использование! Вы только что использовали и текст, и изображения в своем LLM-процессе! Пока еще нельзя сказать, что это MM-RAG, так как мы не выполняем мультимодальный поиск, но мы скоро это исправим!

Завершаем подготовку данных — теперь можно возвращаться к RAG-компонентам, таким как векторные хранилища и ретриверы. Настроим векторное хранилище:

```
vectorstore = Chroma(
   collection_name="mm_rag_google_environmental",
   embedding_function=embedding_function
)
```

Здесь мы создаем новую коллекцию с именем mm_rag_google_environment, указывая на мультимодальную природу ее содержимого. Мы добавляем цепочку embedding_function, которая будет использоваться для встраивания (эмбеддинга) данных, аналогично тому, что мы уже делали в предыдущих лабораторных работах. Однако раньше мы обычно добавляли документы в векторное хранилище при его создании.

В этот раз мы ждем с добавлением документов не только до завершения настройки векторного хранилища, но и до настройки ретривера! Но как можно добавить их в ретривер, если ретривер — это механизм поиска документов? Как мы уже говорили ранее, ретривер в LangChain — это всего лишь оболочка над векторным хранилищем, так что векторное хранилище по-прежнему присутствует внутри, и мы можем добавлять документы в ретривер практически так же, как и напрямую в векторное хранилище.

Но сначала нам нужно настроить мультимодальный ретривер:

```
# mult-vector retriever - initialize the storage layer
store = InMemoryStore()
id_key = "doc_id"

# Create the multi-vector retriever
retriever_multi_vector = MultiVectorRetriever(
   vectorstore=vectorstore,
   docstore=store,
   id_key=id_key,
)
```

Здесь мы обернули наш vectorstore с помощью MultiVectorRetriever. Но что такое InMemoryStore? Это класс для хранения данных в памяти, который сохраняет пары ключ-значение. Он используется в качестве docstore — хранилища документов, где хранятся фактические данные каждого документа, связанные с его идентификатором (doc_id).

Далее мы передаем все это в MultiVectorRetriever(...), который объединяет несколько векторных хранилищ и обеспечивает поиск по нескольким типам данных. Мы уже использовали vectorstore, но теперь мы также подключаем docstore для хранения и поиска содержимого документа. Переменная store (экземпляр InMemoryStore) передается в retriever с параметром id_key, который выступает в роли внешнего ключа, связывая два хранилища по аналогии с реляционной базой данных.

Но пока в наших хранилищах нет данных! Давайте создадим функцию для их добавления:

```
# Helper function to add documents to the vectorstore and docstore
def add_documents(retriever, doc_summaries, doc_contents):
   doc_ids = [str(uuid.uuid4()) for _ in doc_contents]
   summary_docs = [
      Document(page_content=s, metadata={id_key: doc_ids[i]})
      for i, s in enumerate(doc_summaries)
   ]
   content_docs = [
      Document(page_content=doc.page_content, metadata={id_key: doc_ids[i]})
      for i, doc in enumerate(doc_contents)
   ]
   retriever.vectorstore.add_documents(summary_docs)
   retriever.docstore.mset(list(zip(doc_ids, content_docs)))
```

Эта функция является вспомогательной и используется для добавления документов в vectorstore (векторное хранилище) и docstore (хранилище документов) внутри ретривера. Она принимает в качестве аргументов retriever, список doc_summaries и список doc_contents. Как мы уже обсуждали, у

нас есть резюме (summaries) и содержимое (contents) для каждой категории — тексты и изображения, которые мы передаем в эту функцию.

Функция генерирует уникальные идентификаторы документов (doc_id) с помощью str(uuid.uuid4()), затем создает список summary_docs, проходя по doc_summaries и формируя объекты Document, где резюме используется как содержимое страницы (page_content), а идентификатор добавляется в метаданные. Аналогичным образом создается список content_docs, содержащий полный текст документа.

После этого summary_docs добавляется в vectorstore через retriever.vectorstore.add_documents, content_docs добавляется в docstore через retriever.docstore.mset, связывая идентификатор документа с его содержимым

Теперь необходимо вызвать функцию add_document, чтобы заполнить хранилище данными:

```
# Add texts and images to vectorstore, vectorization is handled automatically
if text_summaries:
   add_documents(retriever_multi_vector, text_summaries, texts)
if image_summaries:
   add_documents(retriever_multi_vector, image_summaries, images)
```

Этот шаг добавит документы и их резюме в нужные хранилища, а также создаст векторные представления для текстов и описаний изображений, которые будут использоваться в MM-RAG.

Следующий этап — добавление вспомогательных функций, которые понадобятся для финального MM-RAG конвейера, начиная с функции разделения base64-изображений и текстов:

```
# Split base64-encoded images and texts
def split_image_text_types(docs):
   b64_images = []
   texts = []
   for doc in docs:
      # Check if the document is of type Document
      if isinstance(doc, Document):
         if doc.metadata.get("category") == "Image":
            base64_image = doc.metadata["image_base64"]
            b64_images.append(base64_image)
         else:
            texts.append(doc.page_content)
      else:
         # Handle the case when doc is a string
         if isinstance(doc, str):
            texts.append(doc)
   return {"images": b64_images, "texts": texts}
```

Эта функция принимает список docs (изображения и тексты) и разделяет его на изображения в формате base64 и текстовые данные. Сначала создаются два пустых списка: b64_images и texts. Потом осуществляется проход по каждому документу в docs. Если doc является объектом Document и его категория в metadata — "Image", извлекается base64-изображение (doc.metadata["image_base64"]) и добавляется в b64_images. Если doc является Document, но не является изображением, его текстовое содержимое (doc.page_content) добавляется в texts. Если doc — просто строка, она добавляется в texts.

В результате функция возвращает словарь, где ключ *images* содержит список изображений в формате base64, а ключ *texts* – список текстов.

Также у нас есть функция для создания промта для изображений:

```
def img_prompt_func(data_dict):
   formatted_texts = "\n".join(data_dict["context"]["texts"])
   messages = []
```

```
    # Adding image(s) to the messages if present
    if data_dict["context"]["images"]:
        for image in data_dict["context"]["images"]:
            image_message = {"type": "image_url","image_url": {"url": f"{image}"}}
            messages.append(image_message)

    # Adding the text for analysis
    text_message = {
        "type": "text",
        "text": (
            "You are are a helpful assistant tasked with describing what is in an image.\n"
            "The user will ask for a picture of something.  Provide text that supports what was asked for.\n"
            "Use this information to provide an in-depth description of the aesthetics of the image. \n"
            "Be clear and concise and don't offer any additional commentary. \n"
            f"User-provided question: {data_dict['question']}\n\n"
            "Text and / or images:\n"
            f"{formatted_texts}"
        ),
    }
    messages.append(text_message)
    return [HumanMessage(content=messages)]
```

Эта функция принимает data_dict в качестве входных данных и создает промт-сообщение для анализа изображений. Она извлекает текстовые данные из data_dict["context"], объединяя их в одну строку formatted_texts с помощью "\n".join. Затем инициализируется пустой список messages. Если в data_dict["context"] есть ключ "images", функция перебирает все изображения из списка. Для каждого изображения создается словарь image_message, где:

- "type" указывается как "image_url"
- "image_url" содержит base64-кодированное изображение

Каждое созданное image_message добавляется в список messages.

Теперь добавляем финальный штрих – прежде чем запустить MM-RAG, создадим конвейер MM-RAG, используя только что настроенные функции:

```
# Create RAG chain
chain_multimodal_rag = (
    {"context": retriever_multi_vector | RunnableLambda(split_image_text_types), "question":
RunnablePassthrough()}
    | RunnableLambda(img_prompt_func)
    | llm
    | str_output_parser
)
```

Этот MM-RAG конвейер включает следующие компоненты:

- {"context": retriever_multi_vector | RunnableLambda(split_image_text_types), "question": RunnablePassthrough()} – это аналогично другим компонентам ретривера, которые мы использовали ранее. "context" получает результат retriever_multi_vector | RunnableLambda(split_image_text_types), где retriever_multi_vector выполняет поиск релевантных документов. Полученные результаты передаются в RunnableLambda(split_image_text_types), который вызывает split_image_text_types – функцию разделения данных на base64-изображения и текст. "question" использует RunnablePassthrough, который просто передает вопрос без изменений.
- RunnableLambda(img_prompt_func) – выходные данные предыдущего шага (разделенные изображения, текст + вопрос) передаются через RunnableLambda(img_prompt_func). img_prompt_func формирует промт-сообщение для анализа изображений на основе контекста и вопроса. Этот сформированный промт передается на следующий шаг – LLM.

- llm – обработанный промт, включающий изображение в формате base64, передается в LLM. LLM обрабатывает этот мультимодальный промт и передает результат на следующий этап – парсер вывода.
- str_output_parser – знакомый нам StrOutputParser, который разбирает сгенерированный ответ в виде строки.

Этот конвейер MM-RAG извлекает релевантные документы, разделяет их на изображения и текст, генерирует промт, обрабатывает его с помощью LLM и преобразует результат в строку.

Теперь вызываем конвейер и реализуем полнофункциональный мультимодальный поиск:

```
# Question - relevant question
user_query = "Picture of multiple wind turbines in the ocean."
chain_multimodal_rag.invoke(user_query)
```

Обратите внимание, что в этот раз мы используем новый запрос пользователя (user_query), который соответствует изображениям, доступным в нашей базе.

Вот выходные данные, полученные в результате работы MM-RAG с этим пользовательским запросом:

```
The image shows a vast array of wind turbines situated in the ocean,
extending towards the horizon. The turbines are evenly spaced and
stand tall above the water, with their large blades capturing the
wind to generate clean energy. The ocean is calm and blue, providing
a serene backdrop to the white turbines. The sky above is clear with a
few scattered clouds, adding to the tranquil and expansive feel of the
scene. The overall aesthetic is one of modernity and sustainability,
highlighting the use of renewable energy sources in a natural
setting.
```

Ответ соответствует строке user_query, а также промту, который мы использовали для объяснения LLM, как описывать изображение, которое он «видит». Поскольку у нас всего три изображения, легко определить, о каком идет речь – это изображение №2, которое мы можем извлечь следующим образом:

```
# Display a base64 image by rendering it with HTML
def plt_img_base64(img_base64):
   image_html = f'<img src="{img_base64}" />'
   display(HTML(image_html))
plt_img_base64(img_base64_list[1])
```

Эта функция – вспомогательная, как мы и обещали, она позволяет увидеть изображение. Она принимает base64-кодированное изображение (img_base64) в качестве входных данных и отображает его с помощью HTML. Она создает строку HTML (image_html), содержащую тег , где атрибут src принимает base64-кодированный URL изображения. Затем использует функцию display() из IPython, чтобы отрендерить HTML-строку и вывести изображение.

Запустите этот код, и вы увидите изображение, которое было извлечено из PDF, став основой для ответа MM-RAG!

А теперь, для справки, выведем сгенерированное описание этого изображения. Так как индексы в списке image_summaries и img_base64_list совпадают, мы можем обратиться к…

```
image_summaries[1]
```

Выведенное описание должно выглядеть так:

```
Offshore wind farm with multiple wind turbines in the ocean, text
"What\'s inside" on the left side.
```

Сравнив это с итоговым описанием, сгенерированным MM-RAG, которое более подробное и информативное, можно увидеть, что LLM действительно "видит" изображение и может его описывать. Поздравляем – теперь у вас есть полноценный мультимодальный RAG!

Мы выбрали три лабораторные работы в этой главе, потому что они, на наш взгляд, наиболее широко представляют возможные улучшения для большинства RAG-приложений. Однако это лишь вершина айсберга. В следующем разделе мы рассмотрим другие техники, которые можно включить в ваш RAG-конвейер, в зависимости от ваших конкретных задач.

Другие продвинутые методы RAG, которые стоит изучить

Как и в случае с большинством аспектов RAG и генеративного ИИ, количество доступных продвинутых техник для улучшения вашей RAG-системы слишком велико, чтобы их можно было перечислить или даже отследить. Мы выбрали методы, специализированные для RAG, распределив их по этапам конвейера, на которых они окажут наибольшее влияние.

Рассмотрим их в том же порядке, в котором работает RAG-конвейер, начиная с индексирования.

Улучшение индексирования

Эти продвинутые методы улучшают этап индексирования в RAG-конвейере:

- Глубокое разбиение (Deep chunking): Качество поиска зависит от того, как данные разбиваются на фрагменты перед сохранением в системе поиска. Глубокое разбиение использует глубинные нейросети, включая трансформеры, для оптимального и интеллектуального разбиения.
- Обучение и использование адаптеров эмбеддингов: Адаптеры эмбеддингов – это облегченные модули, адаптирующие уже обученные эмбеддинги под конкретные задачи или домены без необходимости полного дообучения. В RAG-системах адаптеры помогают лучше интерпретировать запросы и контекст, обеспечивая более точный поиск.
- Многоаспектное индексирование (Multi-representation indexing): Индексирование утверждений использует LLM для создания резюме документов, оптимизированных под поиск.
- Рекурсивная абстрактивная обработка для древовидного поиска (RAPTOR): RAG-системы должны обрабатывать как низкоуровневые вопросы, основанные на конкретных фактах из одного документа, так и высокоуровневые вопросы, требующие обобщения информации из нескольких источников. Однако обычный поиск kNN ограничен, так как извлекает только фиксированное количество фрагментов. RAPTOR решает эту проблему, создавая древовидные структуры обобщений. Он кластеризует и суммаризирует документы, а затем повторяет процесс на более высоком уровне, создавая иерархию смысловых представлений. В итоге поиск учитывает и конкретные факты, и обобщенные идеи.
- Контекстуализированное позднее взаимодействие на BERT (ColBERT): Эмбеддинговые модели кодируют текст в векторы, чтобы сжато передавать смысл документа. Однако такое представление может терять нюансы и искажать истинное значение текста. ColBERT решает эту проблему, формируя более детализированные эмбеддинги, где каждое слово документа сравнивается с запросом по отдельности, обеспечивая более точное семантическое соответствие.

Извлечение (Retrieval)

Извлечение — самая обширная категория продвинутых RAG-методов, что подчеркивает его ключевую роль в RAG-конвейере. Рассмотрим основные подходы, которые стоит учитывать при создании RAG-приложений:

- Гипотетические эмбеддинги документов (HyDE): Метод улучшения поиска, при котором LLM сначала генерирует гипотетический документ на основе запроса пользователя. Этот документ затем эмбеддируется и используется для поиска по индексу. Идея заключается в том, что гипотетический документ может лучше соответствовать индексированным документам, чем сам пользовательский запрос.
- Извлечение по оконному методу (Sentence-window retrieval): Вместо поиска по большим фрагментам выполняется поиск по отдельным предложениям, которые наиболее соответствуют контексту. Затем окно вокруг найденного предложения расширяется, чтобы создать более полный контекст.
- Автоматическое объединение фрагментов (Auto-merging retrieval): Решает проблему фрагментации, которая часто возникает при использовании наивного RAG с мелкими блоками текста. Используется эвристика автоматического объединения, которая собирает мелкие фрагменты в более крупные единицы, улучшая связность контекста.

- Мультизапросное переформулирование (Multi-query rewriting): Метод, в котором один вопрос переформулируется разными способами, затем поиск выполняется по каждому переформулированному варианту, а результаты объединяются, исключая дубли.
- Метод "шага назад" (Query translation step-back): Способ улучшения поиска, основанный на цепочке рассуждений (CoT). Генерируется "шаг назад" — более общий, абстрактный вопрос, который создает основу для корректного ответа на исходный вопрос. Это особенно полезно, когда необходимо предварительное знание или глубокое понимание предмета.
- Структурирование запросов (Query structuring): Метод преобразования текста в DSL (Domain-Specific Language), где DSL — это специализированный язык запросов для работы с конкретными базами данных. Автоматически переводит пользовательский запрос в структурированный SQL- или DSL-запрос.

Методы постобработки поиска и генерации (Post-retrieval/generation)

Эти продвинутые техники применяются на этапе генерации ответа в RAG-конвейере:

- Переранжирование с кросс-энкодером (Cross-encoder re-ranking): Мы уже видели в лабораторной работе по гибридному RAG, что переранжирование улучшает поиск. Однако кросс-энкодерный метод идет дальше, применяя более сложную модель для детального анализа и пересортировки найденных документов по их релевантности к исходному запросу. Это повышает качество итогового ответа.
- Переформулирование запросов (RAG-fusion query rewriting): Метод RAG-fusion переформулирует запросы, выполняет поиск по каждому из вариантов, объединяя и ранжируя результаты для повышения точности поиска.

Конвейер RAG в целом

Эти продвинутые методы RAG охватывают весь конвейер RAG, а не только отдельные его этапы:

- Саморефлексивный RAG с использованием LangGraph улучшает наивные модели RAG за счет механизма саморефлексии, совмещенного с лингвистической графовой структурой LangGraph. Этот подход позволяет LangGraph глубже понимать контекст и семантику, что дает системе RAG возможность уточнять свои ответы на основе более детального анализа содержимого и его взаимосвязей. Это особенно полезно в таких приложениях, как создание контента, системы вопросов и ответов и диалоговые агенты, поскольку приводит к более точным, релевантным и осмысленным ответам, значительно улучшая качество генерируемого текста.
- Модульный RAG: Модульный RAG использует сменные компоненты, обеспечивая более гибкую архитектуру, которая может адаптироваться к различным задачам разработки RAG. Такая модульность позволяет исследователям и разработчикам экспериментировать с различными механизмами поиска, генеративными моделями и стратегиями оптимизации, настраивая систему RAG под конкретные задачи и приложения. Как вы уже видели в лабораторных работах этой книги, LangChain предоставляет инструменты, поддерживающие этот подход, позволяя легко заменять и переключать LLM, ретриверы, векторные хранилища и другие компоненты. Цель модульного RAG — создать более настраиваемую, эффективную и мощную систему RAG, способную справляться с широким спектром задач с большей точностью.

С каждым днем появляется все больше исследований, поэтому этот список методов стремительно расширяется. Один из лучших источников новых методик — сайт Arxiv.org. Посетите этот сайт и ищите ключевые термины, относящиеся к вашему RAG-приложению, включая RAG, генерацию, дополненную поиском, векторный поиск и другие смежные темы.

Саммари

В заключительной главе мы рассмотрели несколько продвинутых методов для улучшения RAG-приложений, включая расширение запросов, разбиение запросов и MM-RAG. Эти методы улучшают поиск и генерацию, дополняя запросы, разбивая вопросы на подзадачи и интегрируя несколько типов данных. Мы также обсудили ряд других передовых методов RAG, охватывающих индексацию, поиск, генерацию и весь конвейер RAG.

Было приятно пройти с вами этот путь изучения RAG и его огромного потенциала. Теперь у вас есть знания и практический опыт для реализации собственных RAG-проектов. Удачи в ваших будущих разработках RAG!