Глава 11. Использование LangChain для улучшения RAG

Это продолжение перевода книги Кит Борн. Раскрытие потенциала данных с помощью генеративного ИИ и технологии RAG. Мы уже неоднократно упоминали LangChain и рассмотрели множество примеров кода, включая использование LangChain Expression Language (LCEL) — специфичного языка выражений LangChain. Теперь, когда вы знакомы с различными способами генерации, дополненной поиском (RAG) с LangChain, самое время углубиться в его дополнительные возможности, которые помогут сделать ваш RAG-конвейер еще более эффективным. В этой главе мы рассмотрим менее известные, но крайне важные компоненты LangChain, которые могут улучшить RAG-приложение. Мы затронем следующие темы:

- Загрузчики документов для получения и обработки данных из различных источников
- Сплиттеры текста для разбиения документов на фрагменты, удобные для поиска
- Парсеры вывода для структурирования ответов языковой моделью

Предыдущая глава Содержание Следующая глава

Код для этой главы размещен в репозитории GitHub.

Лаборатория кода 11.1 — Загрузчики документов

Файл с кодом.

Загрузчики документов играют ключевую роль в доступе к данным, их извлечении и загрузке для работы RAG-приложения. Они позволяют загружать и обрабатывать документы из различных источников, таких как текстовые файлы, PDF, веб-страницы или базы данных. Эти документы затем конвертируются в формат, подходящий для индексации и поиска.

Давайте установим несколько новых пакетов, необходимых для работы загрузчиков документов, включая библиотеки для обработки различных форматов файлов.

```
# New installs for document loaders
%pip install bs4==0.0.2
%pip install python-docx==1.1.2
%pip install docx2txt==0.8
%pip install jq==1.8.0
```

Первый из необходимых пакетов вам уже знаком – bs4 (Beautiful Soup 4), который мы использовали во второй главе для парсинга HTML. Также понадобятся несколько библиотек для работы с Microsoft Word, включая python-docx (позволяет создавать и редактировать файлы .docx) и docx2txt (извлекает текст и изображения из .docx файлов). Пакет jq является легковесным процессором JSON.

Далее мы выполним дополнительный шаг, который в реальных условиях, скорее всего, не потребуется: преобразуем наш PDF-документ в несколько других форматов, чтобы протестировать их обработку. Мы добавим новый раздел загрузчиков документов сразу после настройки OpenAI.

В этом разделе будет представлен код для генерации файлов, а также загрузчики документов и связанные с ними библиотеки для извлечения данных из этих форматов. В данный момент у нас есть PDF-версия документа, но нам также потребуются HTML/веб, Microsoft Word и JSON.

В новой ячейке, расположенной под настройкой OpenAI, мы импортируем библиотеки, необходимые для конвертации форматов:

```
# set up our file to be available in all formats:
from bs4 import BeautifulSoup
import docx
import json
```

BeautifulSoup помогает парсить HTML-страницы. python-docx используется для работы с Microsoft Word .docx. json предназначен для обработки JSON-файлов.

Далее зададим пути к файлам, в которых будут сохраняться документы в различных форматах. Эти файлы мы будем использовать при загрузке документов с помощью загрузчиков. Это итоговые файлы, сгенерированные из исходного PDF-документа:

```
# Document paths
pdf_path = "google-2023-environmental-report.pdf"
html_path = "google-2023-environmental-report.html"
word_path = "google-2023-environmental-report.docx"
json_path = "google-2023-environmental-report.json"
```

Ключевая часть нового кода заключается в извлечении текста из PDF-файла и его преобразовании в различные форматы, включая HTML, Word и JSON.

```
with open(pdf_path, "rb") as pdf_file:
    pdf_reader = PdfReader(pdf_file)
    pdf_text = "".join(page.extract_text() for page in pdf_reader.pages)

    # Text to HTML
    soup = BeautifulSoup("<html><body></body></html>", "html.parser")
    soup.body.append(pdf_text)
    with open(html_path, "w", encoding="utf-8") as html_file:
        html_file.write(str(soup))

    # Text to Word
    doc = docx.Document()
    doc.add_paragraph(pdf_text)
    doc.save(word_path)

    # Text to JSON
    with open(json_path, "w") as json_file:
        json.dump({"text": pdf_text}, json_file)
```

Мы создаем версии нашего документа в формате HTML, Word и JSON в самом базовом виде. Если бы эти документы предназначались для использования в конвейере, мы бы рекомендовали применять дополнительное форматирование и извлечение данных, но для целей этой демонстрации нам будет достаточно полученных данных. Далее мы добавим загрузчики документов на этапе индексирования в нашем коде. Мы уже работали с первыми двумя загрузчиками документов, которые покажем в этой лабораторной работе, но обновленными так, чтобы их можно было использовать взаимозаменяемо. Для каждого загрузчика мы укажем, какие пакеты необходимы для его работы.

В первых главах мы использовали веб-загрузчик, который загружал данные напрямую с веб-сайта, поэтому, если вам нужен такой вариант, обратитесь к соответствующему загрузчику. А пока мы рассмотрим немного другой загрузчик, ориентированный на использование локальных HTML-файлов, таких как тот, который мы только что сгенерировали.

Вот код этого HTML-загрузчика:

```
# HTML Loader
from langchain_community.document_loaders import BSHTMLLoader
loader = BSHTMLLoader(html_path)
docs = loader.load()
```

Здесь мы используем ранее определенный HTML-файл для загрузки кода из HTML-документа. Итоговая переменная docs может использоваться взаимозаменяемо с любыми другими документами, которые мы определим в следующих загрузчиках. В этом коде можно использовать только один загрузчик за раз, и он заменит docs своей версией документа (включая метаданные с указанием источника). Если выполнить этот код, а затем перейти к выполнению кода сплиттера, можно запустить оставшуюся часть лабораторной работы и получить схожие результаты, поскольку данные поступают из разных типов файлов. Нам пришлось внести небольшое изменение в код, о чем мы сообщим далее. На сайте LangChain перечислены альтернативные HTML-загрузчики.

Следующий тип файла, о котором мы поговорим, — это PDF.

```
# PDF Loader
from PyPDF2 import PdfReader
```

```
docs = []
with open(pdf_path, "rb") as pdf_file:
    pdf_reader = PdfReader(pdf_file)
    pdf_text = "".join(page.extract_text() for page in pdf_reader.pages)
    docs = [Document(page_content=page) for page in pdf_text.split("\n\n")]
```

Здесь представлена немного более упрощенная версия кода, который мы использовали ранее для извлечения данных из PDF. Этот новый подход демонстрирует альтернативный способ доступа к данным, но оба варианта одинаково подходят для загрузки документов с данными из PDF, используя PdfReader из PyPDF2.

Следует отметить, что существует множество мощных способов загрузки PDF-документов в LangChain, поддерживающего интеграции с различными популярными инструментами для обработки PDF. Вот некоторые из них: PyPDF2 (используемый здесь), PyPDF, PyMuPDF, MathPix, Unstructured, AzureAIDocumentIntelligenceLoader и UpstageLayoutAnalysisLoader.

Рекомендуем ознакомиться с актуальным списком загрузчиков PDF-документов. LangChain предоставляет полезные руководства по многим из них.

Далее загрузим данные из документа Microsoft Word:

```
# Microsoft Word Loader
from langchain_community.document_loaders import Docx2txtLoader
loader = Docx2txtLoader(word_path)
docs = loader.load()
```

Этот код использует загрузчик Docx2txtLoader из LangChain, чтобы преобразовать ранее сгенерированный Word-документ в текст и загрузить его в переменную docs, которая затем может быть использована сплиттером. Как и в случае с HTML или PDF-документами, дальнейшее выполнение кода будет работать с этими данными аналогичным образом. Существует множество вариантов загрузки Word-документов, которые можно найти здесь.

Наконец, аналогичный подход используется для загрузки JSON:

```
# JSON Loader
from langchain_community.document_loaders import JSONLoader
loader = JSONLoader(
    file_path=json_path,
    jq_schema='.text',
)
docs = loader.load()
```

Здесь мы применяем загрузчик JSON для загрузки данных, сохраненных в формате JSON-объекта. Результат остается таким же: переменная docs, которая передается в сплиттер и преобразуется в используемый далее формат. Другие варианты загрузчиков JSON можно найти здесь.

Обратите внимание, что некоторые загрузчики добавляют дополнительные метаданные в словарь metadata внутри объектов Document, создаваемых в процессе загрузки. Это вызывает проблемы в нашем коде при добавлении собственных метаданных. Чтобы исправить это, мы обновляем соответствующие строки кода при индексировании и создании векторного хранилища.

```
dense_documents = [Document(page_content=doc.page_content, metadata=
    {"id": str(i), "search_source": "dense"}) for i, doc in enumerate(splits)]
sparse_documents = [Document(page_content=doc.page_content, metadata=
    {"id": str(i), "search_source": "sparse"}) for i, doc in enumerate(splits)]
```

Мы также обновляем код в финальном выводе, чтобы протестировать ответ, изменяя вторую строку кода для обработки обновленного тега метаданных:

```
for i, doc in enumerate(retrieved_docs, start=1):
    print(f"Document {i}: Document ID: {doc.metadata['id']}
    source: {doc.metadata['source']}")
    print(f"Content:\n{doc.page_content}\n")
```

Запустите каждый загрузчик, а затем выполните оставшуюся часть кода, чтобы увидеть, как обрабатывается каждый документ! Доступны интеграции со сторонними сервисами, позволяющие получать данные из любого источника и форматировать их для к более эффективного использования компонент LangChain. Дополнительные примеры можно найти на сайте LangChain.

Загрузчики документов играют вспомогательную, но очень важную роль в вашем RAG-приложении. Однако для приложений, работающих с Генерацией, дополненной поиском (RAG), которые обычно используют фрагменты данных, загрузчики документов становятся полезными только после их обработки текстовым сплиттером. Далее мы рассмотрим текстовые сплиттеры и их применение для улучшения вашего RAG-приложения.

Лаборатория кода 11.2 — Текстовые сплиттеры

Файл с кодом.

Текстовые сплиттеры разбивают документ на фрагменты, которые затем можно использовать для поиска. Обработка больших документов создает сложности для многих частей RAG-приложения, и сплиттер — это первая линия защиты. Если векторизовать очень большой документ, при увеличении его размера будет теряться больше информации при векторном представлении. Но даже это не самая главная проблема: зачастую векторизовать слишком большой документ просто невозможно!

Большинство моделей встраивания имеют ограничения на размер обрабатываемого. Например, контекстная длина модели OpenAI, используемой для генерации эмбеддингов, составляет 8191 токен. Если передать документ, превышающий этот лимит, модель выдаст ошибку. Именно поэтому существуют сплиттеры, но их применение сопровождается дополнительными сложностями.

Ключевой аспект, который стоит учитывать при работе со сплиттерами, — это способ разбиения текста. Допустим, у вас есть 100 абзацев, которые нужно разделить. В некоторых случаях два или три абзаца логически связаны друг с другом, например, в рамках одной смысловой секции. Иногда встречаются заголовки разделов, URL-адреса или другие специфические текстовые элементы. В идеале необходимо сохранять связанные по смыслу фрагменты текста вместе, но это гораздо сложнее, чем кажется!

Для наглядного примера можно перейти на сайт ChunkViz и вставить туда большой текст. ChunkViz — это инструмент, созданный Грегом Камрадтом, который помогает визуализировать работу текстового сплиттера. Измените параметры сплиттера, установив размер фрагмента (чанка, chunk) на 1000 и перекрытие фрагментов на 200. Попробуйте сравнить сплиттер, разбивающий по символам, с рекурсивным сплиттером по символам. Обратите внимание, что в примере, представленном на рисунке 11.1, рекурсивный сплиттер по символам захватывает все абзацы отдельно, формируя фрагменты размером около 434 токенов.

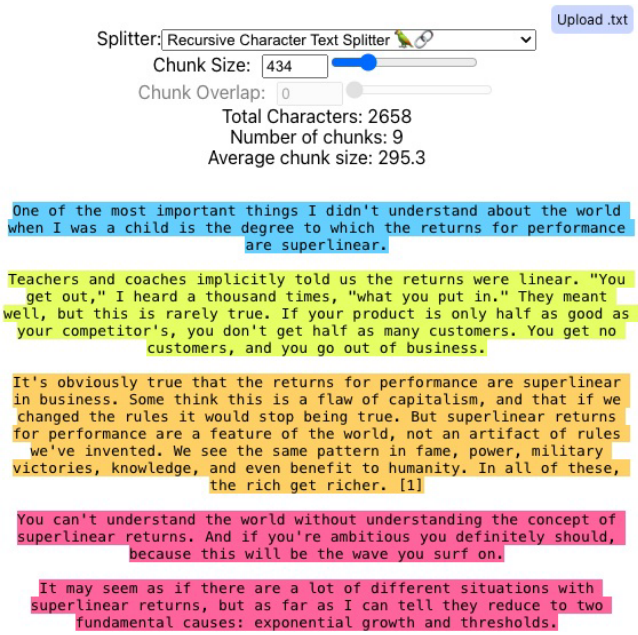

Рис. 11.1. Рекурсивный символьный сплиттер захватывает целые абзацы при размере фрагмента 434 символа

По мере увеличения размера фрагмента текст остается разделенным по абзацам, но со временем в одном фрагменте оказывается все больше абзацев. Это зависит от структуры текста. Если в тексте очень длинные абзацы, потребуется установить большее значение размера фрагмента, чтобы сохранить целые абзацы.

С другой стороны, если использовать обычный символьный сплиттер, он будет обрезать текст прямо посреди предложения, независимо от настройки.

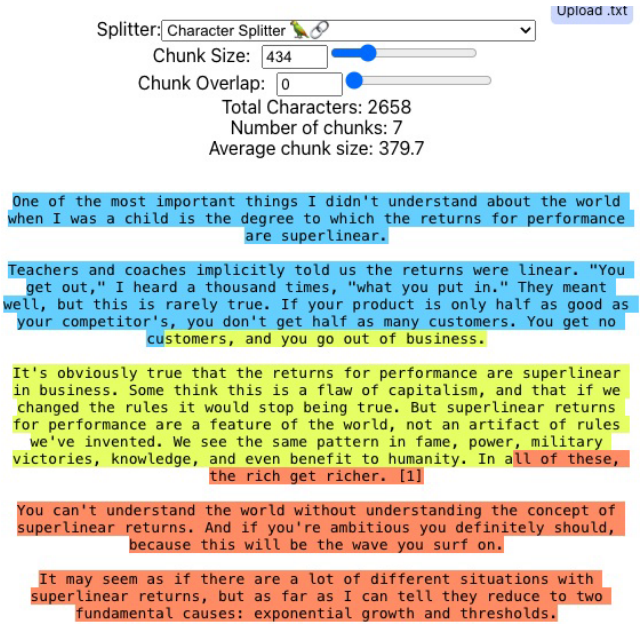

Рис. 11.2. Символьный сплиттер разделяет абзацы на части при размере фрагмента 434 символа

Такое разбиение предложений может существенно повлиять на способность фрагментов передавать важные смысловые элементы текста. Это можно компенсировать увеличением перекрытия фрагментов, но частично обрезанные абзацы все равно будут представлять собой шум для LLM, снижая точность ответов.

Рассмотрим примеры кода для каждого метода, чтобы понять, какие возможности у нас есть.

Символьный сплиттер

Это самый простой способ разбиения документа. Текстовый сплиттер позволяет разделить текст на фрагменты произвольного размера в N символов. Можно слегка улучшить этот метод, добавив параметр-разделитель, например \n. Это хороший начальный подход для понимания принципа разбиения на фрагменты, после чего можно перейти к более сложным методам, которые обеспечивают лучшую обработку текста, но требуют дополнительных настроек.

Вот код, использующий объект CharacterTextSplitter для работы с нашими документами. Его можно использовать взаимозаменяемо с другими выходными данными сплиттера:

```
from langchain_text_splitters import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
   separator="\n",
   chunk_size=1000,
   chunk_overlap=200,
   is_separator_regex=False,
)
splits = text_splitter.split_documents(docs)
```

Выходные данные после первого разбиения (split[0]) выглядят следующим образом:

```
Document(page_content='Environmental \nReport\n2023What's \ninside\ nAbout this
report\nGoogle's 2023 Environmental Report provides an overview of our environmental
\nsustainability strategy and targets and our annual progress towards them.\u20091 \nThis
report features data, performance highlights, and progress against our targets from our 2022
fiscal year (January 1 to December 31, 2022). It also mentions some notable achievements
from the first half of 2023. After two years of condensed reporting, we're sharing a deeper
dive into our approach in one place.\nADDITIONAL RESOURCES\n^ 2023 Environmental Report:
```

```
Executive Summary\n^ Sustainability.google\n^ Sustainability reports\n^ Sustainability
blog\n^ Our commitments^ Alphabet environmental, social, and governance (ESG)\n^ About
GoogleIntroduction 3\nExecutive letters 4\nHighlights 6\nOur sustainability strategy
7\nTargets and progress summary 8\nEmerging opportunities 9\nEmpowering individuals 12\nOur
ambition 13\nOur appr\noach 13\nHelp in\ng people make 14')
```

В тексте присутствует множество символов \n (перенос строки), а также некоторые \u. Мы видим, что сплиттер отсчитывает примерно 1000 символов, находит ближайший символ \n и использует его как границу первого фрагмента. Однако разбиение происходит прямо посреди предложения, что может вызвать проблемы!

Следующий фрагмент выглядит так:

```
Document(page_content='Highlights 6\nOur sustainability strategy 7\ nTargets and progress
summary 8\nEmerging opportunities 9\nEmpowering individuals 12\nOur ambition 13\nOur
appr\noach 13\nHelp in\ng people make 14 \nmore sustainable choices \nReducing home energy
use 14\nProviding sustainable \ntrans\nportation options 17 \ nShari\nng other actionable
information 19\nThe journey ahead 19\ nWorking together 20\nOur ambition 21\nOur approach
21\nSupporting partners 22\nInvesting in breakthrough innovation 28\nCreating ecosystems for
collaboration 29\nThe journey ahead 30Operating sustainably 31\nOur ambiti\non 32\nOur oper
a\ntions 32\nNet-\nzero c\narbon 33\nWater stewardship 49\nCircular econom\ny 55\nNature and
biodiversity 67\nSpotlight: Building a more sustainable \ncam\npus in Mountain View73
\nGovernance and engagement 75\nAbout Google\n 76\ nSustainab i\nlity governance 76\nRisk
management 77\nStakeholder engagement 78\nPublic policy and advocacy 79\nPartnerships 83\
nAwards and recognition 84\nAppendix 85')
```

Как видно, он немного отступил назад — это связано с тем, что мы задали перекрытие фрагментов в 200 символов. Затем он снова двигается вперед на 1000 символов и делает разбиение на следующем символе \n.

Рассмотрим параметры этого процесса:

- Разделители — В зависимости от выбранного разделителя результаты могут сильно отличаться. Здесь используется \n, и он хорошо подходит для этого документа. Однако если использовать \n\n (двойной перенос строки) в документе, где таких символов нет, разбиение не произойдет! По умолчанию используется \n\n, поэтому важно выбирать разделитель, подходящий для вашего контента.
- Размер фрагмента — Определяет количество символов, к которому мы стремимся при разбиении. Фактический размер может немного отличаться, например, в конце текста, но в целом фрагменты будут иметь заданный размер.
- Перекрытие фрагментов — Количество символов, которые будут дублироваться между соседними фрагментами. Это помогает сохранить контекст: если не использовать перекрытие, предложение может быть разрезано пополам, и смысл будет утрачен. Перекрытие позволяет захватывать дополнительный контекст на границах фрагментов.
- Использование регулярных выражений — Параметр, указывающий, является ли разделитель регулярным выражением.

В данном случае мы задаем размер фрагмента 1000 символов и перекрытие 200 символов. Это означает, что мы используем фрагменты размером до 1000 символов, но при этом каждый следующий фрагмент будет перекрываться с предыдущим на 200 символов. Этот метод перекрытия напоминает технику скользящего окна, используемую в сверточных нейронных сетях (CNN), где окно перемещается по изображению с небольшим перекрытием, чтобы сохранить контекст между фрагментами. Здесь мы пытаемся сохранить контекст внутри текстовых фрагментов.

Другие важные моменты:

- Объекты документа — Мы используем объект Document из LangChain для хранения текста, поэтому вызываем create_documents, чтобы подготовить данные для следующего этапа — векторизации. Если нужно просто получить текстовые фрагменты, можно использовать split_text.
- create_documents требует список — Функция create_documents принимает список строк, поэтому если у вас есть единственная строка, ее нужно обернуть в []. В нашем случае docs уже является списком, поэтому дополнительных изменений не требуется.
- Разделение и фрагментация (Splitting and chunking) — Эти термины можно использовать как синонимы.

Дополнительная информация о CharacterTextSplitter. Документация API доступна по адресу.

Но мы можем сделать лучше! Рассмотрим более продвинутый метод — рекурсивное символьное разбиение текста.

Рекурсивный символьный сплиттер

Мы уже встречались с этим методом! В наших лабораторных работах мы использовали его чаще всего, поскольку LangChain рекомендует применять его для разбиения обычного текста. Именно этим мы и занимаемся.

Как следует из названия, этот сплиттер рекурсивно разбивает текст, стараясь сохранить связанные фрагменты рядом друг с другом. В качестве параметра можно передать список разделителей, и сплиттер будет поочередно пытаться разбить текст по ним, пока фрагменты не станут достаточно малыми. По умолчанию используется список ["\n\n", "\n", " ", ""], который работает хорошо, но мы добавим в него ". ". Это поможет сохранять целые абзацы, предложения (разделенные как "\n", так и ". "), а также слова настолько долго, насколько это возможно.

Вот наш код:

```
# use the same splitter for all of them:
character_splitter = RecursiveCharacterTextSplitter(
   separators=["\n\n", "\n", ". ", " ", ""],
   chunk_size=1000,
   chunk_overlap=200
)
splits = character_splitter.split_documents(docs)
```

Внутри этот сплиттер сначала пытается разбить текст по "\n\n", что соответствует разделению на абзацы. Однако на этом процесс не останавливается: если размер фрагмента оказывается больше установленного нами значения (1000 символов), он использует следующий разделитель ("\n"), и так далее.

Как работает рекурсия в этом методе? Этот алгоритм применяется только в том случае, если переданный текст длиннее заданного размера фрагмента. Он выполняет следующие шаги:

- Определяет последний пробел или символ новой строки в диапазоне [chunk_size - chunk_overlap, chunk_size], чтобы разбиение происходило на границах слов или строк.
- Если подходящая точка разбиения найдена, текст делится на две части: фрагмент до точки разбиения и оставшийся текст после нее.
- Алгоритм рекурсивно применяет этот же процесс к оставшемуся тексту, пока все фрагменты не окажутся в пределах chunk_size.

Почему это лучше обычного символьного разбиения? Подобно символьному сплиттеру, рекурсивный метод зависит от размера фрагмента, но в отличие от простого разбиения по символам, он использует более логичный и последовательный способ сохранения контекста.

Этот метод особенно полезен при работе с большими текстами, которые должны обрабатываться языковыми моделями с ограниченным размером входных данных. Разбивая текст на небольшие фрагменты, можно передавать их модели по отдельности, а затем при необходимости объединять результаты.

Хотя рекурсивный сплиттер является шагом вперед по сравнению с символьным сплиттером, он все же не учитывает семантические связи между частями текста. Он ориентируется только на общие разделители, такие как абзацы и предложения, но не объединяет два абзаца, если они являются частью одной логической мысли и должны быть представлены в векторном пространстве вместе. Давайте посмотрим, сможем ли мы добиться лучшего результата с семантическим сплиттером.

Семантический сплиттер

Мы использовали этот метод в первой лабораторной работе. SemanticChunker — это инструмент, который пока считается экспериментальным. На сайте LangChain он описан так: Сначала разбивает текст на предложения. Затем объединяет соседние предложения, если они достаточно семантически похожи. Другими словами, цель метода — избежать необходимости вручную задавать размер фрагмента, который является ключевым параметром для символьного и рекурсивного сплиттеров, и вместо этого ориентироваться на смысл текста. Подробности можно найти здесь.

Как это работает? SemanticChunker сначала разбивает текст на отдельные предложения, затем группирует их по три и объединяет те группы, которые являются семантически схожими в эмбеддинговом пространстве.

Когда этот метод работает хуже? Если текст сложно анализировать с точки зрения семантики. Например, если в документе много кода, адресов, имен, внутренних идентификаторов и других данных, которые имеют низкую семантическую значимость для модели эмбеддингов, SemanticChunker может не справиться с правильным разбиением текста. Однако в большинстве случаев этот метод показывает хорошие результаты. Пример кода:

```
# SemanticChunker
from langchain_experimental.text_splitter import SemanticChunker
semantic_splitter = SemanticChunker(embedding_function, number_of_chunks=200)
splits = semantic_splitter.split_documents(docs)
```

Здесь мы импортируем SemanticChunker из модуля langchain_experimental.text_splitter. Для разбиения текста мы используем ту же модель эмбеддингов, которая применялась для векторизации документов, и передаем ее в SemanticChunker. Обратите внимание, что этот процесс требует затрат, так как использует тот же API-ключ OpenAI, который мы применяли для генерации эмбеддингов.

SemanticChunker использует эти эмбеддинги, чтобы определить, как именно следует разбить документ на фрагменты. Мы также задаем переменную number_of_chunks = 200, которая определяет, на сколько частей будет разбит документ. Чем больше это значение, тем мельче будут фрагменты, а при меньшем значении будет создано меньшее количество крупных фрагментов.

Как протестировать разные методы разбиения? Лабораторная работа организована так, что можно запускать каждый из типов сплиттеров по отдельности. Попробуйте запустить каждый сплиттер, а затем оставшуюся часть кода, чтобы увидеть, как каждый из них влияет на результаты. Также попробуйте изменять параметры, такие как chunk_size, chunk_overlap и number_of_chunks, в зависимости от того, какой сплиттер используется. Исследование всех этих параметров поможет вам лучше понять, как применять их в своих проектах.

Последний вспомогательный компонент, который мы рассмотрим, — это **парсеры вывода**, отвечающие за формирование окончательного результата нашей RAG-архитектуры.

Лаборатория кода 11.3 – Парсеры выходных данных

Файл с кодом.

Конечный результат любого RAG-приложения — это текст, который может включать в себя форматирование, метаданные и другие связанные данные. Обычно этот вывод генерируется языковой моделью (LLM). Однако бывают случаи, когда необходимо получить более структурированный формат, а не просто текст. Парсеры выходных данных — это классы, которые помогают структурировать ответы LLM в любом месте RAG-приложения. Полученный результат затем передается следующему этапу в цепочке обработки или, как в наших лабораторных работах, используется в качестве финального вывода.

Мы рассмотрим два разных парсера выходных данных и будем использовать их в разных частях конвейера RAG. Начнем со знакомого нам — **парсера строкового вывода**.

Внутри функции **relevance_prompt** добавьте следующий код в новую ячейку:

```
from langchain_core.output_parsers import StrOutputParser
str_output_parser = StrOutputParser()
```

Обратите внимание, что мы уже использовали этот парсер в коде LangChain, который появится далее, но теперь мы присвоим его переменной **str_output_parser**. Давайте разберем этот тип парсера подробнее.

Парсер строкового вывода

Это базовый парсер выходных данных. В простых подходах, как в наших предыдущих лабораторных работах, можно напрямую использовать класс **StrOutputParser** в качестве парсера. Однако можно также присвоить его переменной, особенно если он будет использоваться в нескольких местах кода,

как в нашем случае. Мы уже много раз видели, как он работает: он берет ответ от LLM и передает строковый результат в следующий этап цепочки обработки. Документацию можно найти здесь.

Теперь рассмотрим новый тип парсера — **парсер JSON-вывода**.

Парсер JSON-вывода

Этот парсер принимает вывод от LLM и форматирует его в JSON. Этот парсер может не понадобиться, так как многие современные модели поддерживают встроенные способы возвращения структурированных данных JSON или XML. Однако этот метод полезен для моделей, которые такой поддержки не имеют. Начнем с новых импортов, которые мы возьмем из уже установленной библиотеки **LangChain (langchain_core)**:

```
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.outputs import Generation
import json
```

Эти строки импортируют необходимые классы и модули из **langchain_core**, а также модуль **json**: **JsonOutputParser** используется для парсинга JSON-вывода; **BaseModel** и **Field** позволяют определить структуру JSON-модели; **Generation** представляет сгенерированный вывод; также мы импортируем модуль **json**, чтобы удобнее работать с входными и выходными JSON-данными.

Далее создадим Pydantic-модель **FinalOutputModel**, которая будет представлять структуру JSON-вывода:

```
# Define FinalOutputModel for JSON output
class FinalOutputModel(BaseModel):
    relevance_score: float = Field(description="The relevance score of the retrieved context to the question")
    answer: str = Field(description="The final answer to the question")
```

Эта модель содержит два поля: **relevance_score** (float) — оценка релевантности и **answer** (string) — ответ. В реальных приложениях эта модель может быть гораздо сложнее, но данный пример дает общее представление о том, как можно определить структуру JSON-выходных данных.

Теперь создадим экземпляр парсера **JsonOutputParser**:

```
json_parser = JsonOutputParser(pydantic_model=FinalOutputModel)
```

Эта строка присваивает **JsonOutputParser** с моделью **FinalOutputModel** переменной **json_parser**. В дальнейшем мы будем использовать этот парсер, когда нам понадобится получить JSON-ответ.

Теперь добавим новую функцию между двумя нашими вспомогательными функциями, а затем обновим **conditional_answer**, чтобы использовать ее. Этот код добавляется сразу после функции **extract_score**, которая остается без изменений:

```
def format_json_output(x):
    json_output = {
        "relevance_score": extract_score(x['relevance_score']),
        "answer": x['answer'],
    }
    return json_parser.parse_result([Generation(text=json.dumps(json_output))])
```

Функция **format_json_output** принимает на вход словарь **x** и форматирует его в JSON. Она создает словарь **json_output** с двумя ключами: **"relevance_score"** — получает значение из **x** и передает его в **extract_score** и **"answer"** — берет значение напрямую из **x.** Затем используется **json.dumps**, чтобы преобразовать **json_output** в строку JSON. Создается объект **Generation**, содержащий этот JSON. В конце функция передает объект **Generation** в **json_parser** и возвращает результат парсинга.

Эту функцию нам нужно использовать в **conditional_answer**. Обновите ее следующим образом:

```
def conditional_answer(x):
    relevance_score = extract_score(x['relevance_score'])
    if relevance_score < 4:
        return "I don't know."
```

```
    else:
        return format_json_output(x)
```

Здесь мы обновили **conditional_answer**, чтобы применить **format_json_output**, если ответ считается релевантным, прежде чем возвращать результат.

Далее мы объединим две существовавшие ранее цепочки в коде в одну большую цепочку, которая обработает весь конвейер. В прошлом было полезно разделять их для лучшего понимания отдельных частей, но теперь у нас есть возможность упростить код и показать, как эти цепочки можно объединить в единую логику:

```
rag_chain = (
    RunnableParallel({"context": ensemble_retriever, "question": RunnablePassthrough()})
    | RunnablePassthrough.assign(context=(lambda x: format_docs(x["context"])))
    | RunnableParallel(
        {
            "relevance_score": (
                RunnablePassthrough()
                | (
                    lambda x: relevance_prompt_template.format(
                        question=x["question"], retrieved_context=x["context"]
                    )
                )
                | llm
                | str_output_parser
            ),
            "answer": (
                RunnablePassthrough()
                | prompt
                | llm
                | str_output_parser
            ),
        }
    )
    | RunnablePassthrough().assign(final_result=conditional_answer)
)
```

Здесь **str_output_parser** используется так же, как и раньше. Однако **JSON-парсер** здесь не виден, так как он применяется внутри **format_json_output**, которая вызывается внутри **conditional_answer** в последней строке. Это упрощение работает для нашего примера, в котором мы фокусируемся на парсинге вывода в JSON, но стоит отметить, что мы теряем контекст, который использовался в предыдущих лабораторных работах. Это всего лишь альтернативный подход к организации цепочки обработки.

Наконец, так как наш итоговый вывод теперь представлен в формате JSON, но при этом нам все же нужно учитывать контекст, необходимо обновить код тестового запуска:

```
result = rag_chain.invoke(user_query)
print(f"Original Question: {user_query}\n")
print(f"Relevance Score: {result['relevance_score']}\n")
print(f"Final Answer:\n{result['final_result']['answer']}\n\n")
print(f"Final JSON Output:\n{result}\n\n")
```

Когда мы выводим результат, он выглядит похожим на предыдущие примеры, но теперь мы видим, как выглядит итоговый вывод в формате JSON:

Original Question: What are Google's environmental initiatives?

Relevance Score: 5

Final Answer:

Google's environmental initiatives include empowering individuals to take action through sustainability features in products like Google Maps, Google Nest thermostats, and Google Flights. They aim to help individuals, cities, and partners collectively reduce 1 gigaton of carbon equivalent emissions annually by 2030. Google also works with suppliers to reduce energy consumption and greenhouse gas emissions, as well as engages in public policy advocacy for low-carbon economies. Additionally, Google is involved in initiatives like the iMasons Climate Accord, ReFED, and supporting projects with The Nature Conservancy to address environmental challenges. They also focus on operating sustainably in their own operations, such as promoting sustainable consumption of public goods and engaging with coalitions like the RE-Source Platform. Google is also working on renewable energy solutions and using data analytics tools to drive more intelligent supply chains.

Final JSON Output:
{'relevance_score': '5', 'answer': "Google's environmental initiatives include empowering individuals to take action through sustainability features in products like Google Maps, Google Nest thermostats, and Google Flights. They aim to help individuals, cities, and partners collectively reduce 1 gigaton of carbon equivalent emissions annually by 2030. Google also works with suppliers to reduce energy consumption and greenhouse gas emissions, as well as engages in public policy advocacy for low-carbon economies. Additionally, Google is involved in initiatives like the iMasons Climate Accord, ReFED, and supporting projects with The Nature Conservancy to address environmental challenges. They also focus on operating sustainably in their own operations, such as promoting sustainable consumption of public goods and engaging with coalitions like the RE-Source Platform. Google is also working on renewable energy solutions and using data analytics tools to drive more intelligent supply chains.",
'final_result': {
  'relevance_score': 5.0,
  'answer': "Google's environmental initiatives include empowering individuals to take action through sustainability features in products like Google Maps, Google Nest thermostats, and Google Flights. They aim to help individuals, cities, and partners collectively reduce 1 gigaton of carbon equivalent emissions annually by 2030. Google also works with suppliers to reduce energy consumption and greenhouse gas emissions, as well as engages in public policy advocacy for low-carbon economies. Additionally, Google is involved in initiatives like the iMasons Climate Accord, ReFED, and supporting projects with The Nature Conservancy to address environmental challenges. They also focus on operating sustainably in their own operations, such as promoting sustainable consumption of public goods and engaging with coalitions like the RE-Source Platform. Google is also working on renewable energy solutions and using data analytics tools to drive more intelligent supply chains."}}

Это простой пример JSON-вывода, но на его основе можно создать любую нужную структуру, используя класс **FinalOutputModel**, который мы определили и передали в наш парсер. Дополнительную информацию о JSON-парсере можно найти здесь.

Важно понимать, что сложно полагаться на LLM, чтобы она всегда выдавала результат в строго заданном формате. Более надежная система должна глубже интегрировать парсер в архитектуру, чтобы лучше использовать JSON-вывод. Это также потребует дополнительных проверок, чтобы убедиться, что формат соответствует требованиям следующего этапа обработки. В нашем коде мы реализовали очень легковесный слой для форматирования JSON, чтобы показать, как парсер выходных данных может быть встроен в RAG-приложение.

Саммари

В этой главе мы изучили различные компоненты LangChain, которые могут улучшить RAG-приложение. *Лабораторная работа 11.1* была посвящена загрузчикам документов, которые позволяют загружать и обрабатывать данные из различных источников, таких как текстовые файлы, PDF, веб-страницы или базы данных. Были приведены примеры загрузки документов в форматах HTML, PDF, Microsoft Word и JSON с помощью разных загрузчиков LangChain. Было отмечено, что некоторые загрузчики добавляют метаданные, которые могут потребовать корректировок в коде.

Лабораторная работа 11.2 рассматривала разделители текста (text splitters), которые разбивают документы на фрагменты, подходящие для поиска. Это позволяет работать с большими документами и корректно представлять контекст при поиске по векторам. В главе были рассмотрены:

- CharacterTextSplitter — разбивает текст на фрагменты заданного размера в N символов,
- RecursiveCharacterTextSplitter — рекурсивно разделяет текст, стараясь сохранить смысловые связи,
- SemanticChunker — экспериментальный метод, который объединяет семантически похожие предложения в осмысленные блоки.

Наконец, Лабораторная работа 11.3 была посвящена парсерам выходных данных, которые структурируют ответы языковой модели в RAG-приложении. Мы рассмотрели:

- Парсер строкового вывода — передает ответ LLM в виде строки,
- Парсер JSON-вывода — форматирует вывод в JSON согласно заранее заданной структуре.

Приведенный пример показал, как JSON-парсер можно встроить в RAG-приложение.

В следующей главе мы рассмотрим относительно сложную, но мощную тему — LangGraph и AI-агенты.