Transforming Data Interaction: An In-Depth Guide to RAG with LangChain

In today's data-driven world, the ability to extract relevant information efficiently and present it in a meaningful way is crucial. This is where Retrieval-Augmented Generation (RAG) steps in, combining the power of information retrieval with the generative capabilities of language models. Among the tools advancing this frontier is LangChain, a framework designed to streamline the integration of language models with various data sources.

This article delves into the intricacies of RAG, providing a comprehensive guide on how LangChain can be leveraged to enhance data retrieval and generation processes. We will explore the foundational concepts of RAG, its implementation using LangChain, and practical applications that highlight its potential to transform data interaction.

By the end of this article, you will have a thorough understanding of how to harness the capabilities of LangChain to build sophisticated RAG systems, making data retrieval more efficient.

History

Retrieval-Augmented Generation (RAG) was first introduced in the paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" by Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. The paper was published in 2020 by researchers at Facebook AI. (https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf)

Usecase

The introduction of RAG was motivated by the need to address several challenges in natural language processing (NLP), especially in knowledge-intensive tasks such as:

Handling Large-Scale Knowledge: Traditional models were limited by the fixed size of their input, making it challenging to incorporate large-scale knowledge directly into the model.

Improving Information Retrieval: RAG combines retrieval mechanisms with generative models to dynamically fetch relevant information from a large corpus, improving the ability to answer questions and generate text based on up-to-date information.

Enhancing Model Efficiency: By retrieving relevant passages from a large database, RAG reduces the burden on the generative model to memorize all the facts, leading to more efficient and accurate generation.

RAG - Retrieval-Augmented Generation

Most of us would have used a LLM for some use cases by now. Either to print a recipe or to get help with scientific concepts and so on. We know by now that if we try to ask a LLM chat tool on any of the recent events, we are more likely to get an out-of-date update or something similar to "i don't know". In some cases we also get a message that "based on my knowledge cut-off date xx-xx-xxxx" this is the information.

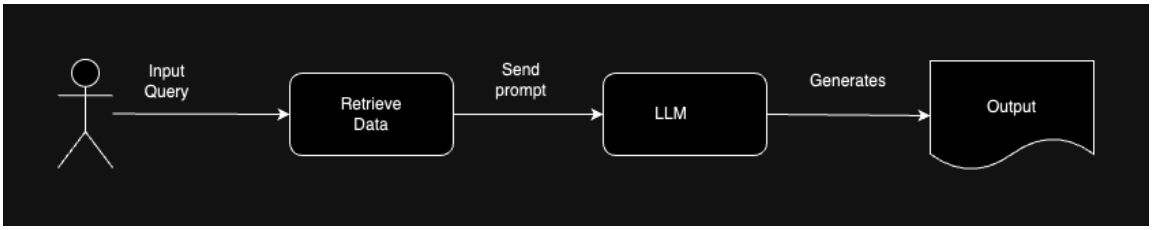
Consider a use case where a company has integrated their chat assistant with an LLM but the response a customer recieves is not updated with the recent information, that would be chaotic and won't reflect well on the business. This issue can be solved with a RAG approach.

Consider another use case where a chat assistant has to fetch information that is specific to a business and this information is not available in public repositories. We could make use of RAG in this case as well.

Components

The main components in a RAG framework are:

1. Retriever: Search component that searches a large corpus of data. This could be structured data like something stored in a database or unstructured data like html pages, pdf documents etc.
2. Generator: A generative model that uses the retrieved passages as context to produce the final output.



RAG frameworks

RAG (Retrieval-Augmented Generation) with LangChain is a powerful framework that combines the strengths of retrieval-based models and generative models in natural language processing.

LangChain is one of the frameworks that can be used to build a Retrieval-Augmented Generation (RAG) system, but it is not the only way to do so. There are several other methods and tools available to construct a RAG.

1. The Hugging Face Transformers library offers pre-trained models for both retrieval and generation tasks.
2. Haystack is an open-source framework designed for building NLP pipelines, including RAG systems.
3. You can build a RAG system from scratch by developing custom retrieval and generation components.

In this article though, we will be focussing on implementing RAG framework using LangChain.

What is LangChain

LangChain is a popular framework specifically designed for building applications that leverage language models. It offers extensive tools and integrations for building RAG systems, including support for retrieval models, APIs, and various data sources.

Official site -> https://python.langchain.com/v0.2/docs/introduction/

RAG with LangChain

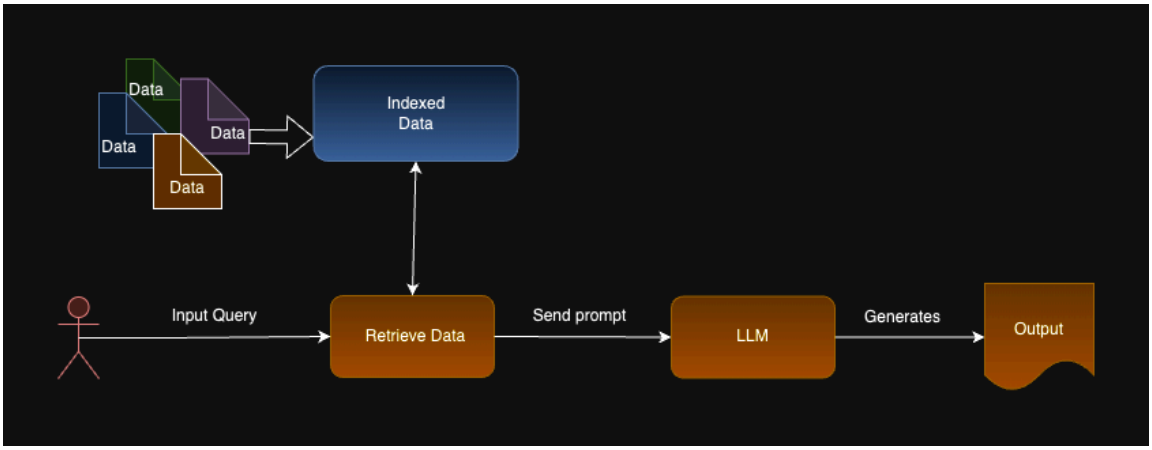
LangChain documentation walks us through two typical use cases of RAG -

1. over unstructured data
2. over structured data

In this article as well, we will implement both uses cases.

RAG over unstructured data

Other than retrieval and generation, another key step in a RAG is Data Indexing. Data indexing refers to the process of organizing and structuring the data in a way that allows for efficient data retrieval.



Pre-requisites

To run the code cells in Jupyter, make sure you have the following dependencies installed in the active python environment,

1. Python (If not installed)
2. jupyter
3. langchain
4. langchain_community
5. langchain_chroma
6. langchain-openai
7. langchainhub
8. bs4

Also, make sure you have retrieved the API keys from OpenAI and Langchain by signing up for both.

LangChain supports many LLMs, please refer official documentation for the complete list. Since we will be using OpenAI LLM in this article, we will import langchain_openai.

In [ ]:
```
import getpass
import os
from langchain_openai import ChatOpenAI


os.environ["LANGCHAIN_API_KEY"] = getpass.getpass()
os.environ["OPENAI_API_KEY"] = getpass.getpass()

model = ChatOpenAI(model="gpt-4")
```

We import a bunch of libraries in this example, most of which are from langchain core and community libraries. A few of them are explained below,

bs4 - The bs4 library, also known as Beautiful Soup, is a Python library used for web scraping purposes to pull the data out of HTML and XML files. It provides Pythonic

idioms for iterating, searching, and modifying the parse tree of these files. Beautiful Soup helps in navigating and searching the parse tree, and it can automatically convert incoming documents to Unicode and outgoing documents to UTF-8, making it a robust tool for web scraping tasks.

langchain_chroma - library that itegrates with Chroma. (official website - https://docs.trychroma.com/getting-started). Chroma is a open-source vector database that can store data in-memory.

In [ ]:
```
import bs4
from langchain import hub
from langchain_chroma import Chroma
from langchain_community.document_loaders import WebBaseLoader
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

Loading Data from Web

In this example, we will be loading data from the html pages that are available online. We make use of SoupStrainer class of beautiful soup library here to parse a part of a page. (Check official documentation of SoupStrainer to evaluate different ways of parsing a part of the document https://www.crummy.com/software/BeautifulSoup/bs4/doc/#bs4.SoupStrainer)

We also make use of DocumentLoader (WebBaseLoader) from LangChain to load the content from given Uris. Again, a wide range of documentloaders are supported by LangChain so please refer their official website for the complete list. (https://python.langchain.com/v0.2/docs/how_to/#document-loaders)

The bs_kwargs parameter is a dictionary that specifies additional keyword arguments for BeautifulSoup. In this case, it uses bs4.SoupStrainer to filter the HTML content, only parsing elements with the id "bodyContent". This means that only these parts of the HTML document will be processed, making the parsing more efficient by ignoring irrelevant parts of the page.

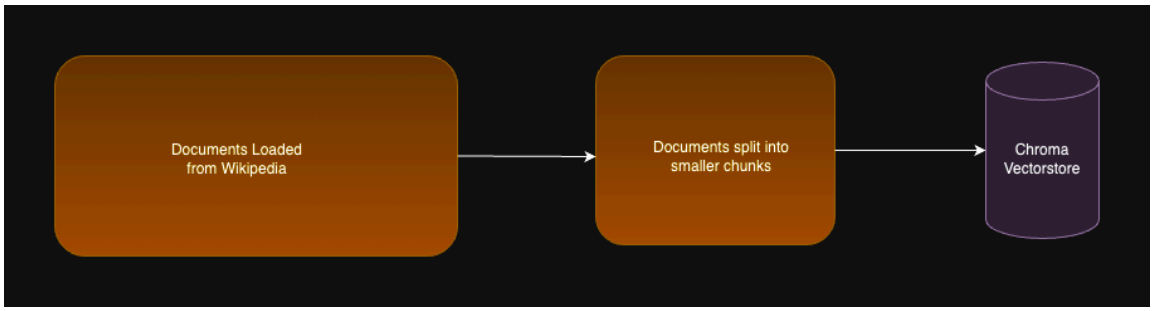
Finally, the docs variable is assigned the result of calling the load method on the loader object

In [ ]:
```
# Load, chunk and index the contents of the blog.
loader = WebBaseLoader(
    web_paths=(["https://en.wikipedia.org/wiki/Great_cuckoo-dove","https://e
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            id="bodyContent"
        )
    ),
```

```
)
docs = loader.load()
```

Splitting

Now that we have loaded the documents, its time to split the documents into smaller chunks.



While splitting a document, it's important to preserve the context of the document so that upon retrieval the most appropriate data can be fetched from the store.

Assume that the document we just loaded has 200 sentences, when we split the document using chunk_size=1000-it means each chunk will have 1000 tokens. Also, to preserve the context, the last 200 tokens of this first chunk will also be added as the first 200 tokens in the second chunk . This is achieved by setting chunk_overlap=200.

By enabling the add_start_index parameter, the system will keep track of the exact position (character index) where each segment or split of the document begins within the original document. This starting position is then stored as a metadata attribute called "start_index" for each split segment. This helps maintain the context and location of each segment relative to the original document, facilitating tasks such as text analysis, searching, and reconstruction of the document.

"splits" variable below holds the chunks of document that we just split.

Next step is to embed these chunks and insert them into a vectorstore which in our case is Chroma. (Check my article on Embeddings if you are new to this word https://medium.com/gitconnected/what-are-text-embeddings-and-how-do-they-work-6107bd52998b)

Using Chroma class - we can embed and store the chunks all in one go.

In [ ]:
```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overla
splits = text_splitter.split_documents(docs)
vectorstore = Chroma.from_documents(documents=splits, embedding=OpenAIEmbedd
```

Retrieval

Once document is loaded, split, embedded and stored in Chroma, its time to work on the retrieval.

retriever = vectorstore.as_retriever() initializes a VectorStoreRetriever object from an existing vectorstore instance. This is achieved by calling the as_retriever method on the vectorstore object. The as_retriever method is designed to convert the vectorstore into a retriever, which is a specialized class used for retrieving documents based on certain search criteria.

In [ ]:
```
# Retrieve and generate using the relevant snippets of the blog.
retriever = vectorstore.as_retriever()
```

Generation - without template

Using retriever object we can call invoke method to pass the query string. This would return a Document object. (https://api.python.langchain.com/en/latest/documents/langchain_core.documents.base.Doc

Note that - while the query string is a valid question (related to the documents that we loaded), we will get a valid response but for an invalid question, eg: "What is peacock"- we will get an incorrect response. (This issue is solved in the next example.)

In [ ]:
```
retrieved_docs = retriever.invoke("What is great cuckoo-dove")
print(retrieved_docs[0].metadata)
print(retrieved_docs[0].page_content)
```

```
{'source': 'https://en.wikipedia.org/wiki/Great_cuckoo-dove', 'start_index':
896}
```

Generation - with template

LangChain provides prompt templates that's stored in hub. In the below example we are using a prompt -> https://smith.langchain.com/hub/rlm/rag-prompt. This prompt instructs the LLM to give a standard response when it cannot find a meaningful response for the input query.

If we ask "What is Peacock" in input query string now, we would get a standard response - 'The context does not provide information on what Peacock is.'

From Official Documentation:

1. retriever | format_docs passes the question through the retriever, generating Document objects, and then to format_docs to generate strings;
2. RunnablePassthrough() passes through the input question unchanged.

In [ ]:
```
prompt = hub.pull("rlm/rag-prompt")

def format_docs(docs):
```

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

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

rag_chain.invoke("What is Peacock")
```

Out[ ]: 'The context does not provide information on what Peacock is.'

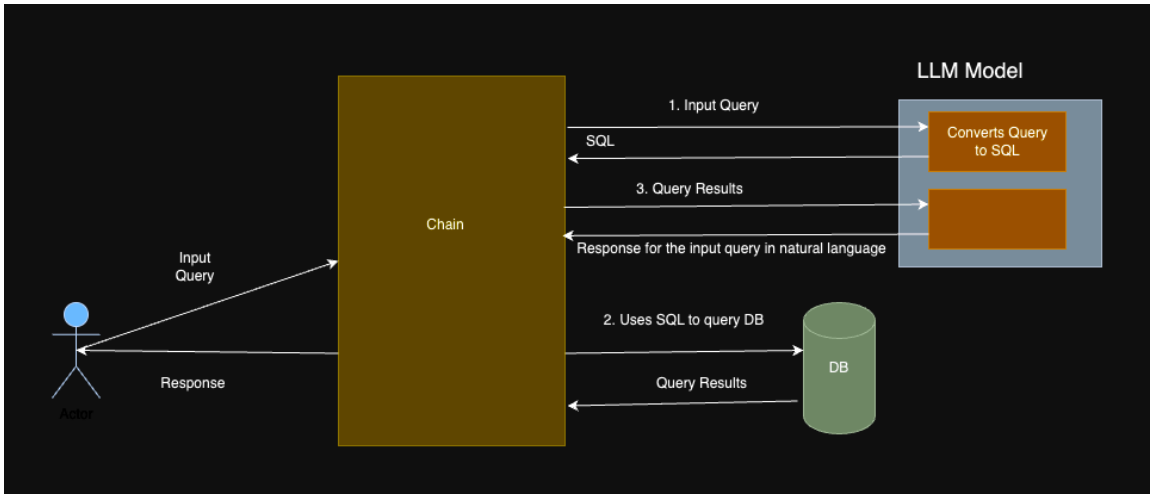
Clean up

Once, all the above cells are executed, lets delete the stored data from the vectorstore.

In [ ]:
```
vectorstore.delete_collection()
```

RAG over structured data

Implementing RAG over structured data means we will have to pull data from a database and to do that we would write SQL queries. Basic workflow of this task is shown below,



Make sure to install the following dependencies

1. <python-package..> -m pip install --upgrade --quiet langchain langchain-community langchain-openai faiss-cpu
2. <python-package..> -m pip install psycopg2-binary

There are a number of databases that we can use for this purpose. In this article, we are using PostgreSQL as our database so if you do not have it installed already, install ot from here -> https://www.postgresql.org/download/macosx/

In the below code cell, we are simply connecting to the database and print some statements to make sure our connection works.

We are also using SQLDatabase class - a SQLAlchemy(ORM library for Python) wrapper, it creates an engine from the given database uri.

Run the following scripts to create a table and populate it with some dummy data

```
CREATE TABLE employees (
postgres(#     id SERIAL PRIMARY KEY,
postgres(#     name VARCHAR(100),
postgres(#     age INTEGER
postgres(# );


INSERT INTO employees (id, name, age) VALUES (1, 'John Doe',
30);
INSERT INTO employees (id, name, age) VALUES (2, 'Jane Doe',
20);
INSERT INTO employees (id, name, age) VALUES (3, 'Lily', 20);
```

In [ ]:
```
from langchain_community.utilities import SQLDatabase

db = SQLDatabase.from_uri("postgresql://localhost:5432/postgres")
print(db.dialect)
print(db.get_usable_table_names())
```

```
postgresql
['employees']
```

In th next code cell, we are creating a chain that would first write a SQL query from the given input question and then pass the output to execute_query. We then execute this chain and print the response.

Let's analyse what each on of these steps do -

1. write query: we use the built-in chain from LangChain create_sql_query_chain to convert an input to a SQL query
2. execute_query: We make use of QuerySQLDataBaseTool class (Tool for querying a SQL database) to execute the generated SQL query.
3. create a chain using #1 and #2.

What is a chain?

In LangChain, a "chain" refers to a sequence of operations or steps that are linked together to accomplish a specific task using language models. Each step in the chain can involve different types of processing, such as retrieving information, transforming data, or generating responses.

Chains allow for the composition of multiple steps into a coherent process. This can involve sequential steps where the output of one step is passed to the next, or parallel steps that execute simultaneously.

In the below code, we are creating a chain named "chain" `chain = write_query | execute_query`

where, "|" is used to create a chain with step #1 and step #2.

In [ ]:
```
from langchain.chains import create_sql_query_chain
from langchain_community.tools.sql_database.tool import QuerySQLDataBaseTool


execute_query = QuerySQLDataBaseTool(db=db)
write_query = create_sql_query_chain(model, db)
chain = write_query | execute_query
response = chain.invoke({"question": "Name the employees who are in their tw
print(response)
```

```
[('Jane Doe',), ('Lily',)]
```

Another way of implementing RAG over structured data using LangChain is to use agents intead of chains. Please check the official documentation to go through the advantages of using one over the other.

In conclusion, implementing RAG using LangChain can enhance the efficiency and accuracy of information retrieval processes, providing more relevant and contextually appropriate responses. As we have explored throughout this article, the flexibility and power of LangChain make it a valuable tool for developers and researchers.

Please do remember to consider the threats with respect to giving a chain/agent the permission to query data directly from your databases. Make sure to set the permissions correctly.