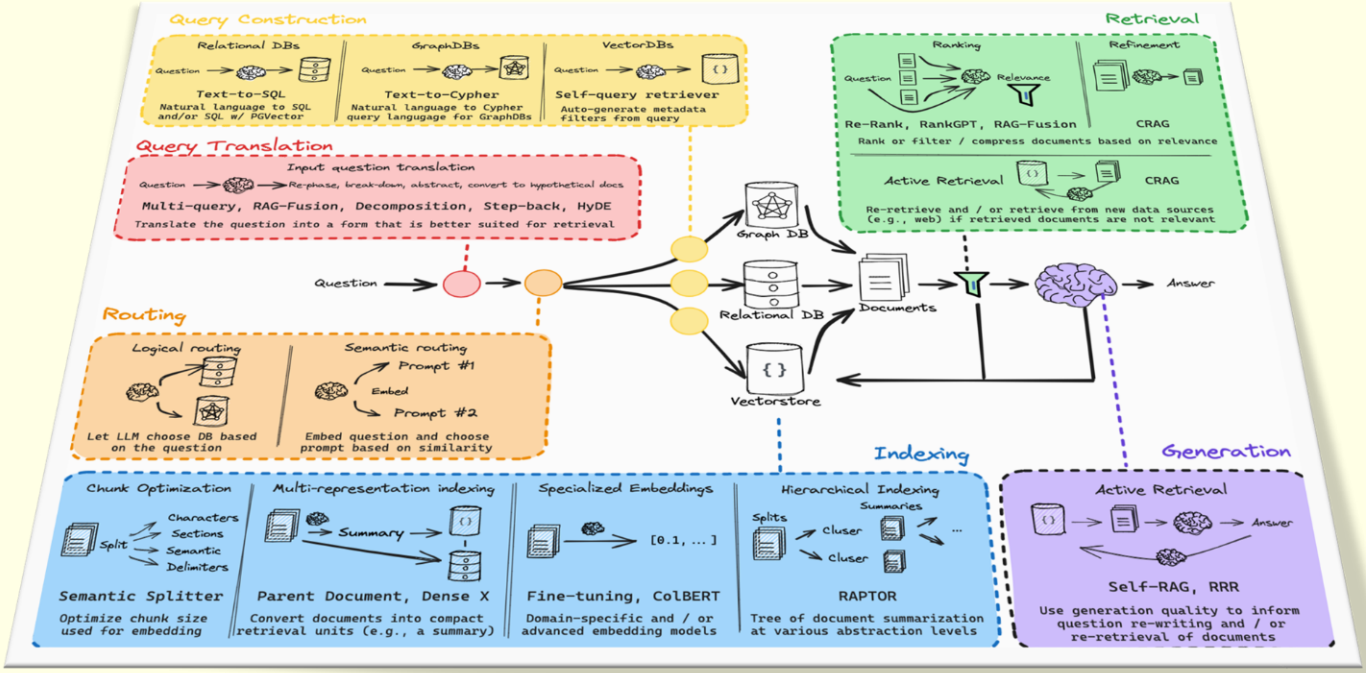
RAG Tutorial

(Retrieval Augmented Generation)

Retrieval and Generation: Retrieve



Day 7 of 7

Table of Contents

1. Introduction

In Welcome to Day 7 of our LangChain RAG tutorial series! Today, we'll bring together everything we've learned and build a chain that takes a question, retrieves relevant documents, constructs a prompt, passes it to a language model, and then parses the output to generate a concise and accurate answer.

This is the culmination of our efforts to create a sophisticated retrieval-augmented generation (RAG) pipeline.

We'll use the gpt-3.5-turbo OpenAI chat model, but the concepts can be applied with any LangChain-compatible LLM or ChatModel.

2. Setting Up the Environment

To use LangChain with OpenAI, you need to set up your environment:

- **Install langchain-openai:**
  - Use pip install *langchain-openai*
- **Get your OpenAI API key:**
  - Obtain it from the OpenAI website and store it securely, ideally as the environment variable *OPENAI_API_KEY*.
- **Import and configure the model:**
  - Import *OpenAI* from *langchain.llms* and initialize the *ChatOpenAI* model, choosing the desired model name (e.g., gpt-3.5-turbo) and temperature.
- **Once these steps are complete, you are ready to start building with LangChain and OpenAI!**

3. Install Packages & API Setup

This section explains how to install necessary packages and configure your OpenAI API key.

1. **Install `*langchain-openai*`:**
   Run the command `*pip install -qU langchain-openai*` to install the package.

2. **Set up API Key:**
   The provided Python code snippet securely prompts the user for their OpenAI API key and stores it in the `*OPENAI_API_KEY*` environment variable.

This ensures secure handling of your API key while making it accessible to your application.

4. Defining the RAG Prompt

This section introduces the use of pre-defined RAG (Retrieval Augmented Generation) prompts from LangChain's prompt hub.

Instead of writing your own prompts from scratch, you can leverage these readily available prompts for common RAG tasks.

The section also showcases an example output generated using such a pre-defined RAG prompt, illustrating its structure and capabilities.

5. Sample Prompt Invocation

This section delves into the practical aspect of using a RAG prompt.

It displays a sample prompt structure with clearly marked placeholders for injecting your specific context and the question you want to ask.

This visual representation helps understand how to effectively use the pre-defined prompt by plugging in your own data and queries.

6. Building the RAG Chain

This section outlines the construction of a RAG (Retrieval Augmented Generation) pipeline using LangChain's LCEL (LangChain Executable Logic) Runnable protocol. The pipeline consists of several key components chained together:

- **retriever**: Fetches relevant information based on the input query.
- **format_docs**: Structures the retrieved documents for optimal input to the language model.
- **RunnablePassthrough**: Allows passing data through the chain without modification.
- **prompt**: The pre-defined RAG prompt with placeholders for context and question.

- **llm**: The chosen large language model for generating the final output.

- **StrOutputParser**: Parses the output from the language model into a readable string format.

This chain effectively combines retrieval, prompting, and language generation to provide comprehensive answers based on provided context.

7. RAG Chain Code

This section presents the Python code used to build the RAG chain described in the previous section.

It demonstrates how to combine the individual components like the retriever, prompt, and language model (llm) using the LCEL pipe (|) operator.

The code snippet shows a clear and concise way to construct the complete RAG pipeline for execution.

To verify the retrieval process, you can check the number of documents retrieved:

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

8. Understanding the LCEL Chain

This section dives deeper into the RAG chain built with the LCEL (LangChain Executable Logic) protocol.

It provides a detailed explanation of each component's role and how they work together to deliver the final output.

Furthermore, it highlights the advantages of using the Runnable protocol, emphasizing its ability to create consistent and flexible chains for various RAG tasks.

9. Using Built-in Chains

This section introduces LangChain's built-in functions that simplify the process of creating RAG chains.

Instead of manually connecting individual components, you can leverage functions like `*create_retrieval_chain*` and `*create_stuff_documents_chain*` for a more streamlined approach.

These pre-built functions offer a higher-level interface for constructing RAG pipelines with less code.

10. Code for Built-in Chains

This section provides concrete Python code examples demonstrating the use of LangChain's built-in functions for creating RAG chains.

Specifically, it showcases how to utilize create_stuff_documents_chain to build a question-answering chain and then integrate it with a retriever using create_retrieval_chain.

This example highlights the simplicity and efficiency of leveraging built-in functionalities for RAG pipeline construction.

```
from langchain.chains import create_retrieval_chain
question_answer_chain = create_stuff_documents_chain(llm, prompt)
rag_chain = create_retrieval_chain(retriever, question_answer_chain)
```

11. Returning Sources

This section emphasizes the crucial aspect of transparency in RAG systems by highlighting the importance of revealing the sources used to generate answers.

It further provides a code snippet demonstrating how to retrieve and display the source documents alongside the generated response, enhancing the trustworthiness and verifiability of the system.

12. Customizing the Prompt

This section delves into the customization capabilities of RAG prompts.

It explains how to tailor the pre-defined prompts from LangChain's hub to meet specific needs.

The section includes a practical example, demonstrating how to add specific instructions to the prompt, such as concluding the generated response with "thanks for asking!"

This highlights the flexibility of RAG prompts in controlling the output format and style.

13. Customized Prompt Code

This section provides a practical demonstration of RAG prompt customization using Python code.

It shows how to define a custom prompt template using the `*PromptTemplate.from_template*` function.

The example code includes specific instructions, like always ending the response with "thanks for asking!", showcasing how to tailor the prompt to elicit desired output formatting and content.

```
template = """Use the following pieces of context to answer
the question at the end...
Always say "thanks for asking!" at the end of the
answer."""
custom_rag_prompt = PromptTemplate.from_template(template)
```

Link of collab Notebook

14. Conclusion

In today's tutorial, we've successfully integrated retrieval and generation into a cohesive chain.

By leveraging LangChain's Runnable protocol, we've built a flexible and powerful RAG pipeline that can be easily customized and extended.

This concludes our 7-day LangChain RAG tutorial series. We hope you've gained valuable insights and skills that you can apply to your own AI projects.

Stay tuned for more advanced tutorials and deep dives into the fascinating world of AI and machine learning!

Bonus Content Coming …