Manuel E. Valdés

Promtior AI Engineer Challenge

Description

This project is chatbot designed to answer questions about Promtior. The system uses RAG architecture to combine the power of Large Language Models (LLMs) with specific context.

Features

- **RAG Architecture:** Combines information retrieval with response generation
- **Automatic Web Scraping:** Obtains updated information from Promtior's website
- **PDF Processing:** Ingests PDF documents to enrich the knowledge base
- **REST API:** Implemented with FastAPI and LangServe
- **Interactive Playground:** Web interface to test the chatbot
- **Semantic Embeddings:** Uses OpenAI Embeddings for semantic search
- **Efficient Vector Store:** FAISS for fast vector storage and retrieval

Technologies Used

- **LangChain:** Framework for LLM applications
- **LangServe + FastAPI:** Deploy LangChain chains as REST APIs
- **OpenAI API:** GPT-3.5-turbo for response generation
- **FAISS:** Efficient similarity search
- **Railway:** Deployment platform

Approach to the solution

When I first read the challenge, the first thing I did was analyze the problem and break it down into smaller tasks to solve it more simply, focusing first on having an MVP and then adding more details or functionality to the app based on that. I identified 4 main tasks:

- **Creation of chatbot knowledge base.**
- **Creation of RAG chain with LangChain.**
- **Application deployment.**
- **Documentation.**

Task 1 - Knowledge base

In this task, two sub-tasks were identified: information ingestion and processing.

Information ingestion

Two possible solutions were determined: manually creating a .txt file with company information or performing web scraping of Promtior's website, and ingesting the PDF document.

On one hand, the first option allows us to have a simplified version for v0, enabling faster progress with the rest of the project's key components and allowing minimal app functionality.

On the other hand, the second option is more complex but more effective for real-world application. Therefore, it was chosen for the final version of the app.

After researching scraping a bit, it was done with **WebBaseLoader**, a LangChain class for automatically loading content from web pages as text documents, and **PyPDFLoader** for loading the PDF with information.

Processing
Tasks were performed so the LLM could interpret the information and use it to answer user questions. First, since the text obtained from these sources is long, it was subdivided into more manageable chunks for the LLM, this was achieved using **RecursiveCharacterTextSplitter**, chunks of 800 with an overlap of 200, also adding some separators so they could be better interpreted by the LLM. Then, we need to generate embeddings for the different chunks, which we did with **OpenAIEmbeddings**. This is used so that when receiving a question, we can compare its vector with those of the chunks and thus measure semantic similarity.

To store the embeddings, we used **FAISS**, which we also use when receiving a question to calculate semantic similarity with stored chunks and retrieve the most relevant ones.

Task 2 - RAG chain
In this step, we need to create the RAG chain, based on **LangChain** documentation.

For this, the **RetrievalQA** class offered by LangChain was used, which is a pre-built chain that implements the RAG pattern. It allows building the complete chain flow by specifying only the language model (LLM), in this case **OpenAI**, and the retrieval mechanism (retriever) as parameters.

Additionally, to help the LLM better perform the task we want, a prompt was written specifying its objective using only the context provided to it.

Task 3 - Deploy
After reading the documentation, **LangServe** was used to deploy LangChain runnables and chains as a REST API. The library is integrated with **FastAPI** and also provides a playground to try the chatbot. For the app deployment, Railway was used without major issues.

Challenges encountered and solutions
One challenge that I found was that I didn't like the way the input and the output of the LangServe playground looked. The input showed the fields question, context and output where I could write and the responses in the output were complex and contained a lot of metadata and context.
So for the input I defined a class ChatInput with the fields that I needed and implemented a wrapper (RunnableLambda) to show only the final response, simplifying the frontend and improving usability.

Another issue I encountered was accessing the /docs endpoint provided by FastAPI. The underlying cause appeared to be related to schema validation issues and limitations arising from the integration between LangServe and Pydantic.