Technical Test Documentation

Project Overview

This project's objective was to create a chatbot assistant capable of answering questions about **Promtior** using a combination of local and web-based data sources. The project employed the Retrieval-Augmented Generation (RAG) architecture, leveraging a Llama LLM and a vector database for context-based responses.

This project was built using Python with LangChain, as it is pretty simple to understand even with limited previous experience.

I built this project trying to make it cost-efficient, as it is supposed to be a test and not a production-level application, and decisions on tools to use were made in line with that idea. The impact of those decisions on the performance of the app is discussed further below in this overview.

Approach

The very first step in this project consisted of reading documentation, as this was the first time I had worked on an RAG application. I consulted the LangChain documentation pages, and I quickly grasped the basics of how a RAG app works conceptually and structurally.

The file structure for the Python project was decided as such:

```
rag/                        # Root project directory
├── app.py                  # Main Flask app entry point
├── src/                    # Source code directory
│   ├── main.py             # Main script used for testing functionality without running app
│   ├── generator.py        # Handles LLM (Ollama) integration and response generation
│   ├── retriever.py        # Vector index loading and document querying logic
│   ├── web_scraper.py      # Logic for scraping and processing web data
│   ├── pdf_loader.py       # Handles PDF file loading and processing
├── data/                   # Data directory for storing indices and raw data
│   ├── vector_index/       # Directory to store the FAISS vector index
│   └── pdf/                # Directory to read PDF from
├── templates/              # Flask HTML templates
│   └── chat.html           # Chatbot frontend UI
├── env/                    # Virtual environment directory (not included in Git)
├── requirements.txt        # List of Python dependencies
└── README.md               # Project documentation
```

To generate responses based on the indexed information, I recurred to using Ollama to run a model locally. Another option was using Open AI's API, but since I don't currently have a subscription, I decided to build this application using the Llama 3.2 model. As this model runs locally, performance is determined by the processing capabilities of the system hosting it. The ideal situation would be to run the model on a GPU-processing enabled computer, and so, not having access to this technology, the performance of the final application is not the best. Namely, it takes several seconds to generate a response. For the purposes of this test, I decided to focus on fulfilling the conditions and not spend many resources on improving the performance.

Data Collection:

- Scraped the website https://www.promtior.ai/ using WebBaseLoader from LangChain.
- Loaded the provided PDF using PyPDFLoader for indexing.

Vector Store Creation:

- Processed both data sources using embeddings generated by HuggingFaceEmbeddings.
- Stored the embeddings in a **FAISS vector database** for efficient similarity searches.

Query Processing:

- When a query is received, relevant documents are retrieved from the vector database using similarity search.
- The retrieved context is sent to the **LLaMA 3.2 model** via **Ollama** to generate the final answer.

Chatbot application:

- Wrapped the response generating functionality in a simple **Flask**-based application to interact with the chatbot.

Deployment:

- Hosted the Flask-based chatbot on an **AWS EC2 instance**, configured with **Gunicorn** and **Nginx**.
- Set up SSL for secure communication and used my custom domain.

Main Challenges

While building the RAG application itself was straightforward due to its structured nature and the available documentation, the main challenges for me arose from solving dependency conflicts, managing resources, and configuring the server:

- Dependency Conflicts: ensuring all dependencies worked seamlessly together was a bit tricky at the start of the project setup. I updated configurations based on error logs and consulted library documentation for guidance.
- Resource Management: running the Llama model locally could cause memory issues and occasionally incorrect answers due to resource constraints. After having trouble with a lower-specification instance, I deployed the optimal CPU-based virtual machine that AWS would allow for my account (I didn’t have access to GPU-based

instances). I optimized Gunicorn and Nginx configurations and implemented timeout handling in the Flask app as well.

- Server Configuration: this being the first time I deployed a Python application online, configuring the server with Nginx, Gunicorn and SSL required some troubleshooting.

Conclusion

This project serves as a practical example of how to build and deploy a scalable AI-powered assistant for specialized domains. The knowledge gained through this implementation paves the way for more powerful and better optimized applications when the needed resources are available.