**Documentation for RAG-based**

**Chatbot Assistant**

Objective of the Test

Develop and deploy a chatbot assistant using Retrieval-Augmented Generation (RAG) architecture to answer questions about the content of the Promtior website. The chatbot is built using the LangChain library.

Proyect Overview

Implementation Logic

* **Import Necessary Libraries:** The project begins by importing essential modules and classes from FastAPI, LangChain, and other libraries. These imports facilitate loading webpage information, generating embeddings, creating a retriever, defining the agent, and running the FastAPI server.
* **Data Preparation:**
  + **Web Scraping:** The content from the Promtior website is gathered using the ‘*WebBaseLoader’* from the LangChain library. The URLs to be scraped include various pages such as the homepage, service page, use-cases page, and contact page.
  + **Text Splitting:** The gathered content is divided into smaller, manageable chunks using the ‘*RecursiveCharacterTextSplitter’*. This step is needed for efficient processing and retrieval of information.
* **Embedding and Vector Store Creation:**
  + **Embedding:** The text chunks are converted into vector representations using the ‘*HuggingFaceEmbeddings’* model. This transformation is important for enabling the chatbot to understand and process the data.
  + **Vector Store:** The embeddings are stored in a FAISS vector store, which allows for efficient retrieval of relevant information based on user queries. This vector store acts as a retriever, fetching the most relevant chunks of data to answer a specific query.
* **Prompt and Language Model Integration:**
  + **Prompt Template:** A template is defined for generating answers. The template takes the context retrieved by the vector store and the user’s question as inputs, ensuring that the responses are coherent and relevant.
  + **Language Model:** The LlaMA2 model is integrated using the Ollama library, with the temperature set to 0 for deterministic responses. This language model generates the final answers based on the retrieved context and the input question.
* **RAG Chain Setup:** The final processing chain combines the retrieval of context, formatting through the prompt template, answer generation by the language model, and output parsing. This chain is crucial to handle user questions and generate appropriate responses.
* **FastAPI Integration:** FastAPI is used to deploy the chatbot as a web service. The ‘*Server.py’* file sets up the FastAPI application.
* **Deployment:** The FastAPI application is deployed using Uvicorn. It serves the application on the specified host and port. This setup makes the chatbot accessible to users through a web interface.

Main Challenges

Integrating the LlaMA2 model with the LangChain library was the most exhausting problem I had. It seemed to be everything perfect, every necessary line of code was there, but I couldn’t detect the error. I read the LangChain and Ollama documentation, debugged the integration process step-by-step, watched videos, but couldn’t fixed it. The server was working, I could run LlaMA2 from the command processor. I tried a lot of changes, thinking that the problem was the way I was accessing the server in the coding syntaxis part. However, the problem was that I was trying to access from Google Collab to the local server Ollama. I then used Jupyter Notebook and the problem was solved.

Apart from this, I run the Ollama server in my 8RAM laptop working extremely slowly. However, I maintained using LlaMA2 since it was the one suggested for the test, and according to the documentation it should work although it was the limit (*“You should have at least 8 GB of RAM available to run the 7B models”).*

Moreover, another important problem I had was ensuring it interacted correctly with the retriever. The documentation showed how to do it, but with a paid license of OpenAI. Therefore, I needed to find how to do it with Ollama but there were many documents, and each one used some resource I didn´t have. It first showed me the following error:

***“ValueError****: Ollama call failed with status code 500. Details: {"error":"llama runner process has terminated: exit status 0xc0000005 "}”*

I didn´t know why, but I tried many things and believe that when adding the OpenAI Key it changed the error and printed the following one:

As I couldn´t fix this problem I chose to follow the steps from another website.

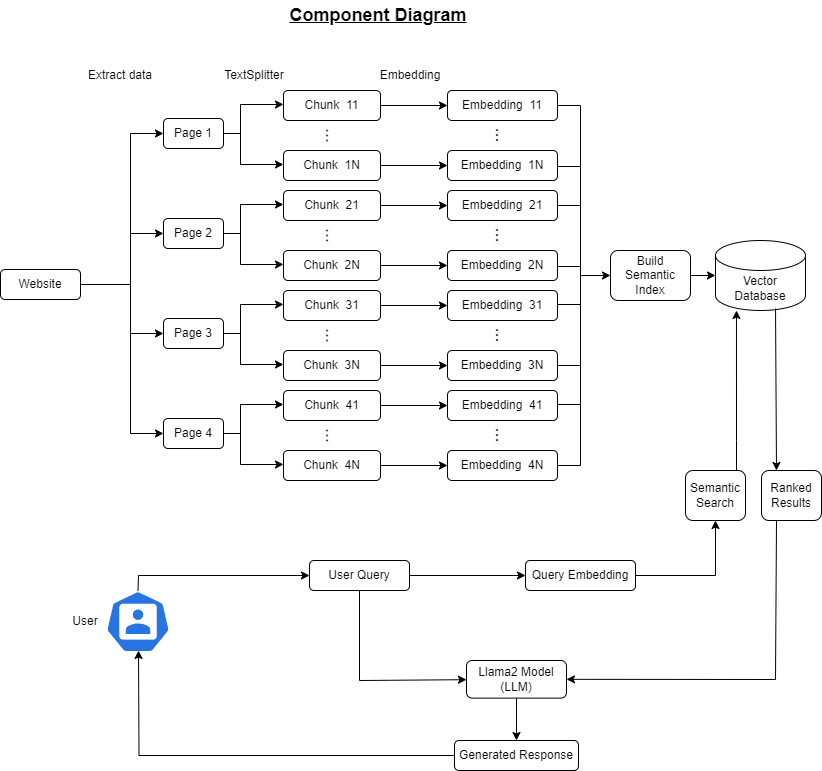
It was then deployed with Langserve, a deployment tool provided by LangChain, on Railway management. However, the failed status code 500 showed up again.

Accessing the chatbot just in python worked correctly but the problem was when accessing it through the web interface. The answer was only seen when looking into the ‘*Intermediate steps’*. However, I don´t know why it was not being printed inside the box and the previous ‘failed status code 500’ showed up in the Visual Studio Code terminal after each question.

Therefore, I decided to start again from zero, creating a simpler processing chain for the RAG system. Creating a ‘final\_chain’ and adding it as the RAG chain endpoint for the FastAPI application, appeared to be a simpler way. The error disappeared and the answer was correctly showed in the web box of the chatbot.

Component Diagram

The following diagram shows the components involved in the solution and their interactions from the time the question is received by the chatbot until the response is given.



Made with Draw.io

The solution is divided into two parts. One is the information gathered from the website and the other the user interaction part.

Firstly, all the pages from the Promtior website are collected. Then we extract all the information from them and split the data into small chunks. Then the embeddings for each chunk are created. They are vectors which contain floating point numbers. All these embeddings are going to be stored in the vector database after building the sematic indexes. Therefore, we have all the information available on the website in this knowledge base.

Apart from this, we have user interaction. When he asks a question, embeddings are created for that question. Then a semantic search is done in the knowledge base. All the website information was stored here in the form of embeddings and organized with semantic indexes. Therefore, it is possible to do a semantic search comparing the query embeddings with the embeddings of the Promtior website data. After finding several answers they are passed to the Large Language Model (Llama2) together with the user question. Finally, the LLM will generate a natural response using the retrieved data as context. The generated answer is sent back to the user.

User Interaction: the user interacts with the chatbot through a web interface or API.

Query: the user’s question is received by the chatbot.

Retriever Component: the question is passed to the FAISS retriever, which searches the vector store for relevant documents.

Document Retrieval: the retriever fetches the most relevant documents based on the user’s question.

LLM Component: retrieved documents and the user’s question are passed to the LLaMA2 model.

Answer Generation: the LLaMA2 model generates an answer using the retrieved documents as context.

Response Delivery: the generated answer is sent back to the user through the web interface or API.

Bibliography

<https://python.langchain.com/v0.1/docs/get_started/quickstart/>

<https://towardsdatascience.com/can-a-llama-2-powered-chatbot-be-trained-on-a-cpu-ce9ec6ebe3c2>

<https://medium.com/credera-engineering/build-a-simple-rag-chatbot-with-langchain-b96b233e1b2a>

<https://www.youtube.com/watch?v=M4G2zk4X93Y>

<https://www.youtube.com/watch?v=VXAVI1p0L4E>