Chatbot Assistant - Promtior’s Webpage

Documentation

Chiara Sosa

November 2024

First approach

When I first reviewed the technical test instructions and requirements, I encountered some unfamiliar concepts and technologies. To prepare, I started by familiarizing myself with them, specifically the concepts of RAG architectures, as well as LangChain and LangServe technologies.

Getting started

Next, i settled up the environment:

1. Created a public git repository
2. cloned it locally to my computer
3. Installed Jupyter Notebook, Langchain, Langgraph
4. Created an .env file with my OpenAI API key

Following the Langchain documentation, I first built and deployed a simple LLM application. Using a jupyter notebook, I learned how to invoke the OpenAI gpt-3.5-turbo model and chain the model with StrOutputParser and prompt templates. Lastly, I created a serve.py file containing all this logic and a route from which to serve the chain.

Building the Chatbot Assistant

With all this new knowledge, I started building a chatbot assistant which could answer questions about the content available on Promtiors website.

Initial Setup and Web Content Loading

After setting the environment variables, I loaded the entire content of the Promtior web page. While I initially followed the LangChain Web Loading guide https://python.langchain.com/docs/how_to/document_loader_web/ , I realized I needed a more comprehensive approach to crawl the entire website. This led me to use the RecursiveUrlLoader method, which I configured with:

- A maximum depth of 5 (appropriate for the website's structure)
- BeautifulSoup for text extraction
- Domain restriction to prevent crawling external links

Content Processing

To make the content of the page usable for the chatbot, i implemented the following steps:

Text splitting:
Used RecursiveCharacterTextSplitter to break down the content into manageable chunks

Creating embeddings and Vector Storage:
Utilized OpenAI's embedding model to convert text chunks into numerical representations with FAISS (Facebook AI Similarity Search) as the vector store. This combination allows for efficient semantic search and retrieval of relevant information.

To understand better this process, i followed this analogy:

> *An international restaurant has thousands of recipes. Chefs need to quickly find similar recipes. To understand which recipes are related, even if they use different words, each recipe gets converted into a special card with numbers. These numbers represent: main ingredients, cooking style, cuisine type, etc. For example:*
>
> *- A Mexican Taco recipe: [0.8, 0.3, 0.9, 0.2, 0.5]*
> *- A Mexican Burrito recipe: [0.7, 0.3, 0.9, 0.2, 0.6]*
> *- A Japanese Sushi recipe: [0.1, 0.8, 0.2, 0.9, 0.3]*
>
> *Taco and Burrito have similar numbers (they're related), while Sushi doesn't (different type of food).*

Promtior's web content needs to also be embedded with "number cards" where similar topics get similar numbers. Then, the chatbot will know where to find the information to answer questions.

Conversation Chain Setup

The model is created based on the 3.5-turbo version of OpenAI. The chain is configured through ConversationalRetrievalChain using the model and the vectorstore created before. The user asks a question and the retriever searches on the vectorstore relevant information. The model receives information from the vectorstore and the chat history and generates an answer.

Chatbot's Architecture

The **state** saves messages and chat context.
The **stateGraph** is configured so the new message follows the path: enters through the entry point (node model), gets processed by call_model and is saved in memory.
The **memory** is the chats long term memory.

Model call: obtains the last question and the chat history. Uses the chain (which includes the chat context) to get a response to the question made. Finally the response is formatted with StrOutputParser.

Deployment - Langserve

1. Fast API creation to serve the chatbot.
2. Creation of API endpoint for the chain. Added the conversational chain to the path: /chain.
3. The app launches through uvicorn, on localhost port 8000.

Deployment - AWS

Dockerfile → creates container image with Python environment and application code

AWS CLI → configure AWS credentials and region settings

AWS Copilot CLI → deploys application to AWS infrastructure
Commnds used:
1. copilot init --app chatbot-promtior --name api --type 'Load Balanced Web Service' --dockerfile './Dockerfile' --port 8000
2. copilot env deploy --name dev --force
3. copilot svc deploy

Conclusions

The chatbot responds effectively in the local environment, demonstrating good performance in processing and responding to queries. However, when deployed to AWS, the application encounters challenges with web scraping:

Local Environment:
- Successful webpage content extraction
- Reliable responses based on the content
- Direct access to the website

AWS Environment:
- Inconsistent webpage content extraction
- Issues accessing dynamic content

These limitations affect the quality of responses when the service runs in AWS, as the chatbot can't consistently access the full webpage content needed for accurate responses.

Chatbot URL:

http://chatbo-publi-pwmt21k72rbh-1880714462.sa-east-1.elb.amazonaws.com/chain/playground/