Project Report on the Computer Hub AI Intern Assignment: “Jessup’s Helper” Chatbot

Soumyadeep Bose

Introduction:

"Jessup's Helper" is a Streamlit-based chatbot application made to provide assistance to potential customers through natural language understanding, and contextual memory-based responses. The chatbot leverages language processing units (LPUs) through Groq API for high-speed inference, open-source large language models (LLMs) for text generation and vector embeddings to deliver accurate and relevant answers to user queries. This project report details the approach adopted for the creation of the chatbot.

Objectives:

1. Developing a user-friendly interface for question answering conversations.
2. Implementing a language model capable of understanding and generating human-like responses.
3. Utilizing document vector embeddings for contextual understanding.
4. Maintaining conversational memory for coherent interactions.

Technologies Used:

1. **Streamlit**: For building the web-based interactive interface.
2. **Groq**: To integrate the ChatGroq model, for high-speed inference.
3. **FAISS**: To store vector embeddings and also for retrieval of relevant information.
4. **Google Generative AI Embeddings**: To generate document embeddings.
5. **PyPDFDirectoryLoader**: For loading PDF documents into the system.
6. **dotenv**: For managing environment variables securely.
7. **Langchain**: To facilitate natural language tasks.

Approach Details:

1. Environment Setup: The app starts by loading the required API keys from environment variables using `dotenv`. The required API keys are `GOOGLE_API_KEY` and the `GROQ_API_KEY`. I have used Google Gen AI Studio API keys to fetch text embeddings for the PDF document, and Groq for high speed inference, since it uses the Language Processing Units (LPUs). Both the API keys are securely stored in an `.env` file that is not uploaded in the GitHub Repo.

2. Initializing the Streamlit Session State Variables:

- **Model Initialization**: First the ChatGroq model is initialized with Groq's API key and the model used is Llama3, since it is open-source and also available to use with Groq. The state variable `llm` is initialized with this model.
- **Other state variables**: The `messages` variable is initialized to store the messages of the current session, so that after every query and response, only those two are not displayed but all of the conversation is displayed. The `store` variable stores the `ChatMessageHistory` by session ID, which is arbitrary for this project since this project only requires the current session's history.

3. Displaying the whole Conversation: Existing messages in the current conversation are rendered in the chat interface using the lines of code given below. I have not opted for

Chainlit for the chat interface since the `async` and `await` keywords were yielding errors that were proving too difficult to resolve. This was the first big problem I encountered.

```
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])
```

5. Getting Document Vector Embeddings: The function `vector_embedding` first ingests and loads all the PDF documents located in the `pdfs` directory, then splits the all the text into chunks of size 1500 for efficient information retrieval, using the `RecusiveCharacterTextSplitter` class. This method can be used to ingest and load multiple PDFs too in case there are. Next, Google Generative AI Embeddings model `embedding-001` is used to get vector embeddings for the document(s), and then `FAISS` is used to store the vectors, which are then used to retrieve relevant information based on the user's query. I used `Ollama` too to run the embeddings model initially, but since it was locally running, it was extremely slow and resource straining. I opted for Google Generative AI Embeddings since it is free for personal projects and the model runs on the cloud.

6. Contextualizing and Answering Questions: I have created two prompt templates, one to retrieve relevant query-based information from the vector store and create a document chain, and another to create a history aware retrieval chain. Next, a conversational retrieval chain is created to get responses from the language model. To add the memory functionality to the model, I first used the Buffer Memory creation approach but that did not quite fit into this project. Thereafter, I went with the current approach of creating a conversational RAG chain. After the response is obtained from the model, it is displayed in the interface, and also appended in the `messages` session state variable.

Conclusion and Future Scope:

"Jessup's Helper" is a RAG-based chatbot that successfully integrates multiple modern technologies to provide a seamless, informed, and intelligent Q&A experience for users. As of now, it is only capable of understanding textual data from the PDFs provided. However, in the future, it can be upgraded to understand multimodal data too, such as graphs and images. We could also use `GraphRAG` methods to make the chatbot even more accurate.