Brance <Position>Task

Name:Arun Murali

Linkedin Profile:[**https://www.linkedin.com/in/arun-murali-2ba3a382/**](https://www.linkedin.com/in/arun-murali-2ba3a382/)

Date Challenge Received:06-07-23(08-07-23 work started)

Date Solution Delivered:10-07-23



1. Problem Statement

What was the task and how you understood it.

The task was to build a RAG(Retrieval Augmented Generation) chatbot. For user question RAG module would retrieve context from knowledge document and generation phase llm would personalize answer using retrieval knowledge. I searched with llms, google, research papers about RAG which helped me to have good understanding of the problem.

2. Approach

Your approach to the problem. Mention any assumptions made.

Approach

The approach to building a chatbot using RAG is to first retrieve the relevant context from the knowledge document based on the user's question. This can be done using a variety of techniques, such as keyword matching, semantic similarity, and entity extraction. Once the relevant context has been retrieved, it can be used to generate a personalized answer using an LLM.

To prevent hallucination, it is important to ensure that the generated answer is consistent with the knowledge document. This can be done by using a variety of techniques, such as fact checking, consistency checking, and coherence checking.

Bonus Features

The following bonus features can be added to the chatbot:

* Evaluation of the answers: This can be done by using a variety of metrics, such as accuracy, relevance, and fluency.
* Supporting multi-linguality: This can be done by using an LLM that supports multiple languages.
* Adding speech capabilities: This can be done by using a speech-to-text engine and a text-to-speech engine.

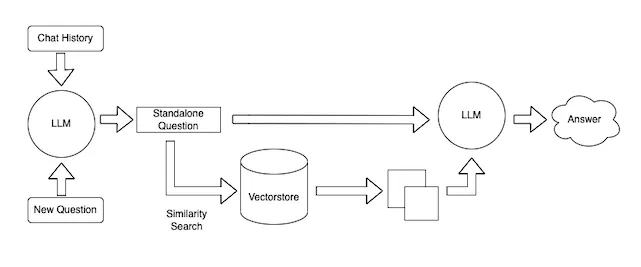
Deliverables

The deliverables for the chatbot include:

* Working solution: This is the code for the chatbot that can be run on a local machine or a cloud platform.
* A clean, efficient, explanatory, and maintainable code: This is the code for the chatbot that is well-organized, easy to read, and easy to maintain.
* A small writeup on approach, assumptions, and future scope: This is a document that describes the approach to building the chatbot, the assumptions that were made, and the future scope of the project.

3. Solution

Details about your solution. Illustrate performance and design with diagrams.



A diagram of the process used to create a chatbot on your data, from [LangChain Blog](https://blog.langchain.dev/tutorial-chatgpt-over-your-data/)

* We ask the user to enter their [OpenAI API key](https://platform.openai.com/account/api-keys) and upload the document file on which the chatbot will be based.
* If a document file is uploaded by the user, we load it using the [Loader](https://python.langchain.com/en/latest/modules/indexes/document_loaders/examples/csv.html?highlight=csvloader) class from LangChain
* The LangChain Loader class allows us to split a pdf file into splits.
* Cutting the pdf file now allows us to provide it to our vectorstore ([FAISS](https://github.com/facebookresearch/faiss)) using OpenAI embeddings.
* Embeddings allow transforming the parts cut by Loader into vectors, which then represent an index based on the content of each row of the given file.
* In practice, when the user makes a query, a search will be performed in the vectorstore, and the best matching index(es) will be returned to the LLM, which will rephrase the content of the found index to provide a formatted response to the user.
* We then add the [ConversationalRetrievalChain](https://python.langchain.com/en/latest/modules/chains/index_examples/chat_vector_db.html?highlight=conversationalretri) by providing it with the desired chat model gpt-3.5-turbo (or gpt-4) and the FAISS vectorstore storing our file transformed into vectors by OpenAIEmbeddings().
* This chain allows us to have a chatbot with memory while relying on a vectorstore to find relevant information from our document.
* chain = ConversationalRetrievalChain.from\_llm(
* llm = ChatOpenAI(temperature=0.0,model\_name='gpt-3.5-turbo'),
* retriever=vectorstore.as\_retriever())
* This function allows us to provide the user’s question and conversation history to ConversationalRetrievalChain to generate the chatbot’s response.
* st.session\_state[‘history’] stores the user’s conversation history when they are on the Streamlit site.

def conversational\_chat(query):

result = chain({"question": query,

"chat\_history": st.session\_state['history']})

st.session\_state['history'].append((query, result["answer"]))

return result["answer"]

* We initialize the chatbot session by creating st.session\_state[‘history’] and the first messages displayed in the chat.
* [‘generated’] corresponds to the chatbot’s responses.
* [‘past’] corresponds to the messages provided by the user.
* Containers are not essential but help improve the UI by placing the user’s question area below the chat messages.

if 'history' not in st.session\_state:

st.session\_state['history'] = []

if 'generated' not in st.session\_state:

st.session\_state['generated'] = ["Hello ! Ask me anything about " + uploaded\_file.name + " 🤗"]

if 'past' not in st.session\_state:

st.session\_state['past'] = ["Hey ! 👋"]

#container for the chat history

response\_container = st.container()

#container for the user's text input

container = st.container()

* Now that the session.state and containers are configured.
* We can set up the UI part that allows the user to enter and send their question to our conversational\_chat function with the user’s question as an argument.
* with container:
* with st.form(key='my\_form', clear\_on\_submit=True):
* user\_input = st.text\_input("Query:", placeholder="Talk about your csv data here (:", key='input')
* submit\_button = st.form\_submit\_button(label='Send')
* if submit\_button and user\_input:
* output = conversational\_chat(user\_input)
* st.session\_state['past'].append(user\_input)
* st.session\_state['generated'].append(output)
* This last part allows displaying the user’s and chatbot’s messages on the Streamlit site using the [streamlit\_chat](https://pypi.org/project/streamlit-chat/) module.
* if st.session\_state['generated']:
* with response\_container:
* for i in range(len(st.session\_state['generated'])):
* message(st.session\_state["past"][i], is\_user=True, key=str(i) + '\_user', avatar\_style="big-smile")
* message(st.session\_state["generated"][i], key=str(i), avatar\_style="thumbs")
* All that’s left is to launch the script:

4. Future Scope

Thoughts on how you could have improved the solution.

Although we have implemented document reading, csv-excel to insights analytics and youtube video summarization, due to time constraints we were not able to implement evaluation metric, multi lingual capability and speech capabilities to the chatbot.

* Evaluation of the answers: This can be done by using a variety of metrics, such as accuracy, relevance, and fluency.
* Supporting multi-linguality: This can be done by using an LLM that supports multiple languages.
* Adding speech capabilities: This can be done by using a speech-to-text engine and a text-to-speech engine.

Further more testing and resolving the negatives, we can further improve chatbot. Building Chatbot is an iterative process, provided more time we can add more functionalities and solve the pitfalls of the chatbot.