RAGanot

Name: Divyanshu Kashyap

LinkedIn Profile: <https://www.linkedin.com/in/divyanshu-kashyap-3bb0b5231/>

Date Challenge Received: 04-07-23 (05-08-23 work started)

Date Solution Delivered: 07-08-23



**1. Problem Statement**

What was the task and how you understood it?

The task was to build a chatbot using the Retrieval Augmented Generation (RAG) model. This chatbot was required to accept a user's question as input and return an answer based on a provided knowledge document. The RAG model was responsible for two main tasks: retrieving relevant context from the knowledge document based on the user's question and generating a personalized answer using this retrieved knowledge. For the generation phase, a Language Model (LLM), possibly orchestrated by Langchain, was to be used.

As I understood it, the goal was to create a system that could interact intelligibly with a user by leveraging the information in the knowledge document. The chatbot needed to be capable of understanding the user's question, finding the most relevant information in the document, and then formulating an appropriate and personalized response. This required an understanding of both retrieval-based models (to identify relevant sections of the document) and generative models (to create a suitable answer). Furthermore, the mention of Langchain suggested the possibility of implementing a more complex pipeline, with various components working together to process the user's input and generate the output.

**2. Approach**

Your approach to the problem. Mention any assumptions made.

**Approach**

My approach to the problem was to first thoroughly understand the requirements and constraints. I then formulated a plan based on the Retrieval-Augmented Generation (RAG) model, which is designed for tasks like this.

* Data Preparation: The first step was to process the provided knowledge document. This involved reading the document and converting it into a format suitable for the retrieval model.
* Question Processing: The next step was to handle the user's question. This included parsing the question and potentially preprocessing it to enhance retrieval and generation performance.
* Retrieval Phase: Using a retrieval model, the system would identify the parts of the knowledge document most relevant to the user's question. The model would be trained or fine-tuned on the knowledge document to optimize its performance.
* Generation Phase: With the relevant context retrieved, a generative model (LLM) would then generate a response. This model would be capable of using the retrieved context to produce a personalized, relevant, and coherent answer.
* Integration with Langchain: I planned to use Langchain to orchestrate the different components of the system. This could include managing the interaction between the retrieval and generation models, handling memory, and managing the dialogue history for context in a conversation.

Throughout this process, I made a few assumptions:

* The knowledge document is sufficiently comprehensive and accurate to answer the user's questions.
* The user's questions are relevant to the information contained in the knowledge document.
* The LLM can generate human-like, coherent responses when provided with relevant context.
* The retrieval model can accurately and reliably identify the most relevant context from the knowledge document based on a user's question.
* The sample questions and answers in SampleQuestions.xlsx file can be used to evaluate the model answers.

These assumptions were necessary to define the scope of the problem and to ensure that the problem was solvable with the RAG approach.

**To prevent hallucination, Contextual Prompt Engineering can be used.**

For LLMs, context is very important for increased accuracy and addressing hallucination.

Having a model stating, “I don’t know” is far better than having a model giving a wrong or hallucinated answer.

Using prompt engineering in the model with “Answer the question as truthfully as possible, and if you're unsure of the answer, say "Sorry, I don't know".” Can improve the results, drastically reducing hallucination.

**Bonus Features**

The following bonus features can be added to the chatbot:

* Multi-turn Conversations: Allow the chatbot to handle multi-turn conversations where the context of previous turns is used to understand and respond to the current user input. This makes the interaction with the chatbot more natural and conversational.
* Personalization: Personalize the chatbot's responses based on user characteristics or preferences. This could be achieved by maintaining a user profile and tailoring the chatbot's responses accordingly.
* Contextual Understanding: Improve the chatbot's ability to understand the context of a conversation. This could involve recognizing when the user changes the topic of conversation and adjusting the chatbot's responses accordingly.
* Sentiment Analysis: Enable the chatbot to recognize and respond to the user's emotional state. The chatbot could adjust its tone or provide empathetic responses based on the sentiment expressed in the user's input.
* Error Handling: Improve how the chatbot handles unrecognized inputs or errors. Instead of simply saying it doesn't understand, the chatbot could ask clarifying questions or suggest possible interpretations of the user's input.
* Active Learning: Implement an active learning mechanism where the chatbot learns from each interaction with a user. Over time, this could improve the chatbot's performance and ability to provide accurate and helpful responses.
* Multimedia Content: Allow the chatbot to handle multimedia content, such as images, audio, and video. The chatbot could provide responses or information based on this content.
* Integration with External Systems: Integrate the chatbot with external systems or databases to provide real-time information or perform actions based on the user's input.

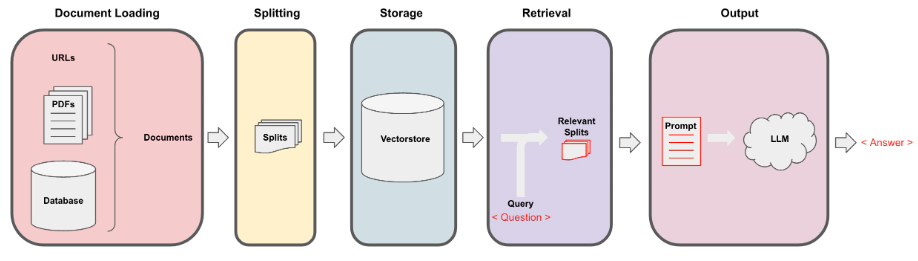
**Deliverables**

The deliverables for the chatbot include:

* Working Solution: Fully functional code for the chatbot, capable of running on local or cloud platforms, complete with necessary files and instructions.
* Clean, Efficient, and Maintainable Code: Code adhering to best practices for readability and maintainability, with detailed comments for complex areas.
* Approach, Assumptions, and Future Scope Document: A concise write-up detailing the development methods, assumptions made, and potential future improvements for the chatbot.

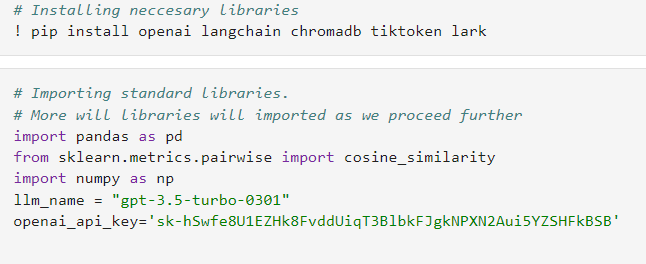
**3. Solution**

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

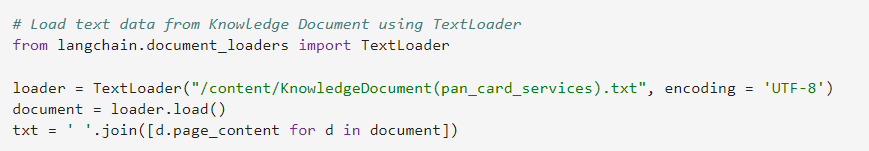


A diagram showing the process to make a chatbot

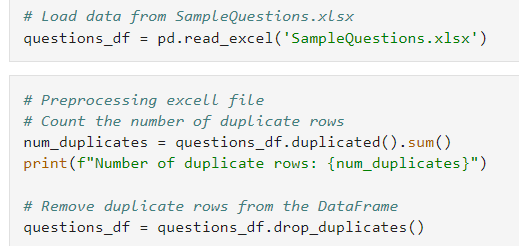
* Started by entering [OpenAI API key](https://platform.openai.com/account/api-keys) and uploading the document files on which the chatbot will be based. We import all the necessary libraries.



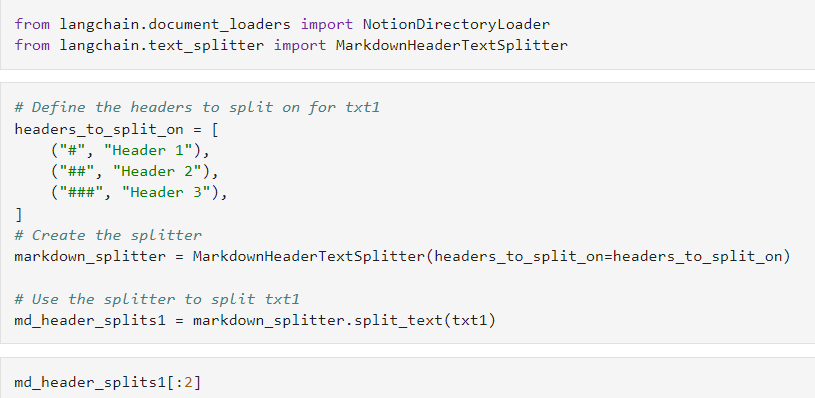
* The LangChain's Loader class is used to load knowledge document file. It also allows us to split any pdf / txt file into splits.



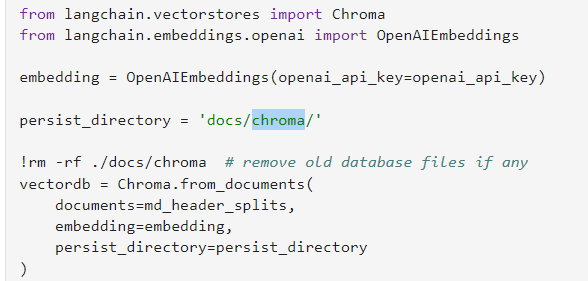
* Sample Questions file is loaded and preprocessed.



* Next, I split text data using a markdown splitter.



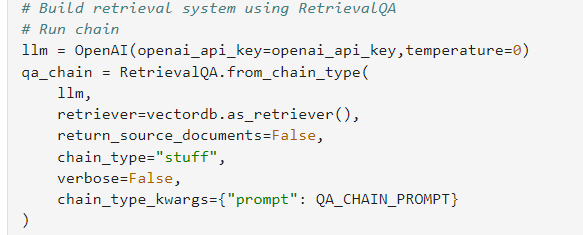
* Splitting the file now allows us to provide it to our vectorstore (chroma) using OpenAI embeddings.
* Embeddings allow transforming the parts cut by Loader into vectors, which then represent an index based on the content of each row of the given file.

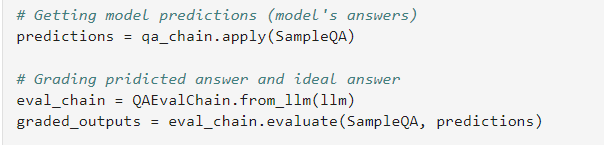


//Next, I retrieved data for a query using the data uploaded by 4 different ways to compare the retrieved data.

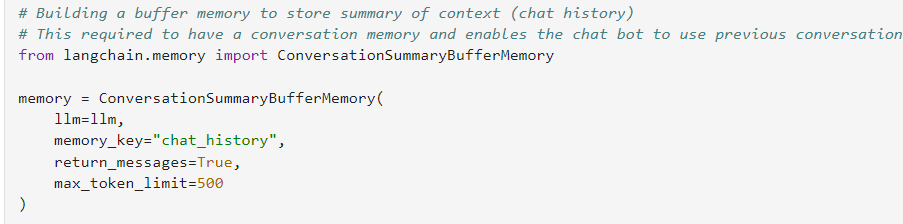
//Saw difference in retrieval when using similarity search, maximum marginal relevance search and using retrievers self-query retriever and compression retriever.

* Next, to build an evaluation system, we build a retrieval chain using RetrievalQA.
* Based on this chain **LLM assisted evaluation** can be done. We are analyzing if the predicted answers are similar to ideal answers or not.

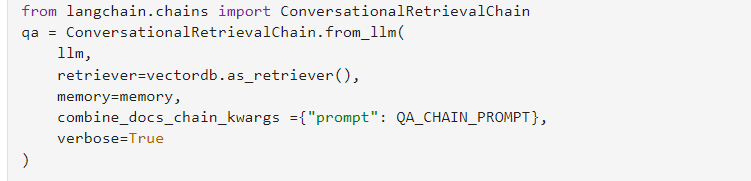




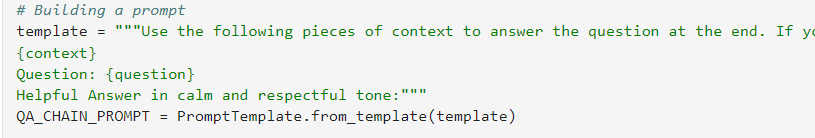
* The evaluation system can be run on different retrieval chain types like map reduce and refine chains to see if they improve accuracy.
* Building buffer memory to store summary of chat history.
* We use ConversationSummaryBufferMemory for its unique ability to store chat history as a summary saving on tokens while with-holding maximum important information.
* This makes **contextual understanding** possible



* Building a buffer conversational chain which can retrieve information from our given data and use previous chat history as context. We add the ConversationalRetrievalChain by providing it with the desired chat model gpt-3.5-turbo and the Choma vectorstore storing our file transformed into vectors by OpenAIEmbeddings().
* This makes **multi-turn conversations** possible



* Using prompt engineering in ConversationalRetrievalChain



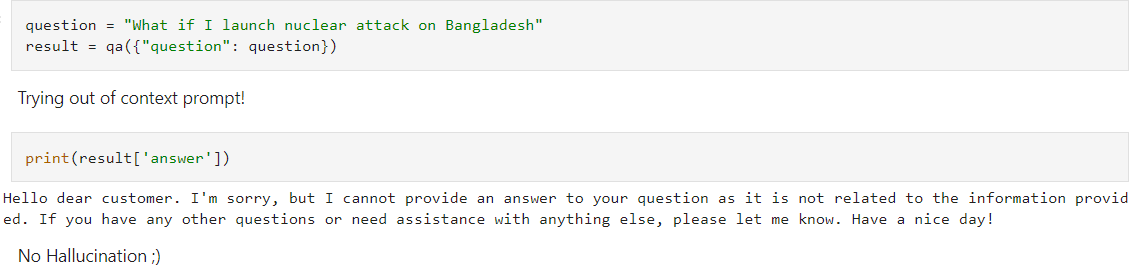
* Using prompt template ="""Use the following pieces of context to answer the question at the end. If you don't know the answer, just say that you don't know, don't try to make up an answer. If you don't know the answer you can ask the customer to clarify and explain his question in detail. Give a systematic answer in points in calm and polite tone. The answer should be consistent and coherent with the data provided. Keep the answer as concise as possible but avoid giving answer in one line. Always greet "Hello dear customer" in the beginning and always say "Have a nice day" at the end of the answer in separate sentences.

{context}

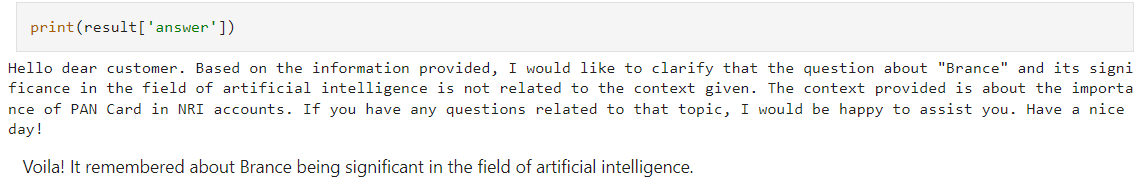
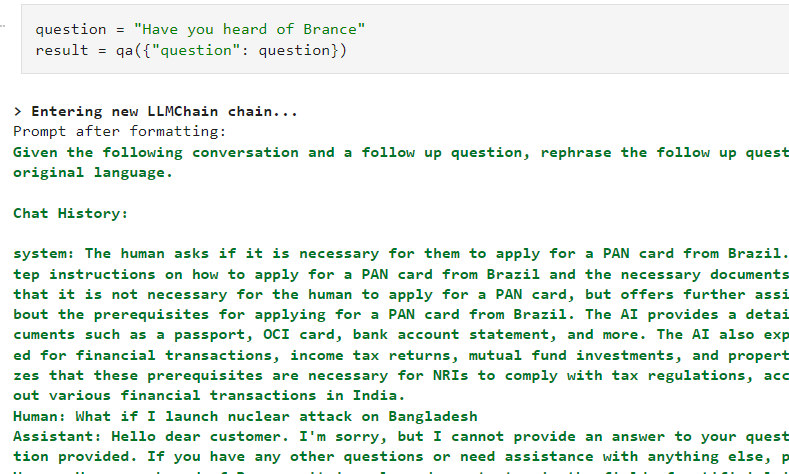
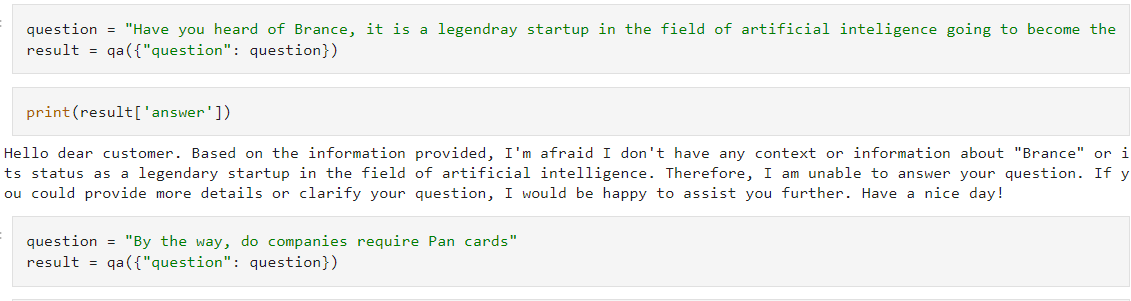
Question: {question}

Helpful Answer in calm and respectful tone:"""

* Using this prompt engineering we are
* **Preventing hallucination**
* Setting tone of answers according to need
* Bringing **personalization**
* **Error handling, i**nstead of simply saying it doesn't understand, the chatbot asks clarifying questions or suggests possible interpretations of the user's input.
* Techniques for **active learning** (mechanism where the chatbot learns from each interaction with a user, improving the chatbot's performance and ability to provide accurate and helpful responses over time) can be applied here in future.
* **Sentiment analysis** enabling the chatbot to recognize and respond to the user's emotional state can be applied here in future.
* Chatting with the bot:



* In the above example, prevention of hallucination due to prompt engineering can be seen.



* The above example demonstrates the ability to store chat history
* Using the last examples one can use Chatbot to get information related to that given in the knowledge document.
* To start a new chat re-run the ConversationSummaryBufferMemory celll

**4. Future Scope**

Thoughts on how you could have improved the solution.

* Designing the User Interface (UI), hosting and deployment.
* Adding more evaluation metrics: This can be done by using a variety of metrics, such as accuracy, relevance, and fluency.
* Support for 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.
* Multimedia content handling.
* Integration with External Systems
* Integration of Ray, a library for building scalable applications, into the RAG contextual document retrieval mechanism. This speeds up retrieval calls by 2x and improves the scalability of RAG distributed fine-tuning.
* Using RAG model from HuggingFace.
* RAG model from HuggingFace basically consists of a pre-trained DPR model. The DPR model consists of two BERT models. One model encodes questions (question encoder), and the other model encodes the documents (passage encoder) using the CLS token outputs. DPR model used in RAG has already been trained with passages and questions extracted from open domain Wikipedia-based datasets. RAG is known to have a neural retriever, and a reader combined in an end-to-end manner. However, in practice, we freeze the passage encoder and only train the question encoder.
* RAG authors illustrated it is ok not to fine-tune the Passage Encoder for tasks like question answering and fact-checking. But the authors have conducted their experiments mainly on open domain Wikipedia-like datasets. Since DPR also initially trained on Wiki-data, it really makes sense!!!

But what about other domains like finance, healthcare, and legislation? So, does the training of the entire RAG-retriever help domain adaptation?