Brance AI Applied Researcher - Intern Hiring Task

Name: Saumay Dudeja
Linkedin Profile:https://www.linkedin.com/in/saumaydudeja
Date Challenge Received: 03-08-23 (12:48)
Date Solution Delivered: 05-08-23

Problem Statement :

The task was to build a RAG(Retrieval Augmented Generation) chatbot. For user questions, the RAG module would retrieve context from knowledge document and during the generation phase, the LLM would provide an answer personalised to the query and retrieved context.

Approach

The approach I used in order to build an RAG chatbot involves the following parts:

Reading, embedding, chunking and storing the knowledge document into a vector database

This is done in order to take the raw data/information present in the knowledge and transform them into embeddings, through which the LLM can actually understand the underlying knowledge, structure and semantic meaning of the text present in the knowledge doc. Hence, embeddings are an important decision factor affecting the performance. In this approach, Open AI embeddings are used.

Chunking and storing of the embeddings into a vector store is done so as to allow efficient retrieval of the relevant pieces of knowledge/context during query time. Different algorithms like k-means, PCA are used to look up the vectors closest to the query vector and the corresponding text is supplied to the LLM along with query. In this approach, FAISS (faiss-cpu) has been used for the same.

Technical side:

- `UnstructuredFileLoader` from Langchain is used to load the knowledge document

- `RecursiveCharacterTextSplitter` is used to split the file contents into chunks. Chunk sizes and overlaps also have considerable effect on question answering. Here, the chunk size was set to be 500 considering the nature of PAN card question answers.

- User is asked for the OpenAI API key

- `OpenAIEmbeddings()` and `FAISS` are used to create embeddings for the chunks and store them into the vector database

Querying, context retrieval and prompting to the LLM

This is the core part of the model. The LLMChain is responsible for taking the query from user, transforming them into embeddings, looking up relevant docs in the vector store and then combining the retrieved context, query, chat history etc. into a prompt that is sent to the LLM. Many different LLM Chains from Langchain can be used. The

`ConversationalRetrievalChain`

allows for a conversation-like experience with the bot since it stores chat history in a

`ConversationBufferMemory`

class but makes doesn't have an easy way to implement response evaluation since the underlying retrieval/prompting techniques are heavily abstracted.

In this approach, the

`RetrievalQA`

class was used as the LLM chain. It supports retrieval of the relevant context from a vector store through a retriever object. The LLM powering the RetrievalQA chain is the default OpenAI LLM ie `gpt-3.5-turbo`

```
RetrievalQA.from_chain_type(
    llm=OpenAI(temperature=0),
    chain_type='stuff',
    retriever=retriever,
    return_source_documents=False,
    verbose=True
    )
```

sets up a Retrieval based QA chain with verbose outputs in the terminal. Fortunately, LangChain provides async support for this LLMChain. The chain is run using the

```
qa.arun({"query":message},callbacks=[cl.AsyncLangchainCallbackHandler()])
```

method

Evaluation of the answers: Some metric to check how relevant the answers are to the questions.(Bonus)

Many different approaches can be used for evaluating the answers.

- **Comparison with ground truths**: Given question-answer pairs relevant to the knowledge doc for training, the model can use them as a reference to learn what correct answers look like and evaluate further answers based upon that

- **Dataset generation**: If there are no given question-answer pairs, we could generate the same using an LLM and then use these generated pairs as the training dataset

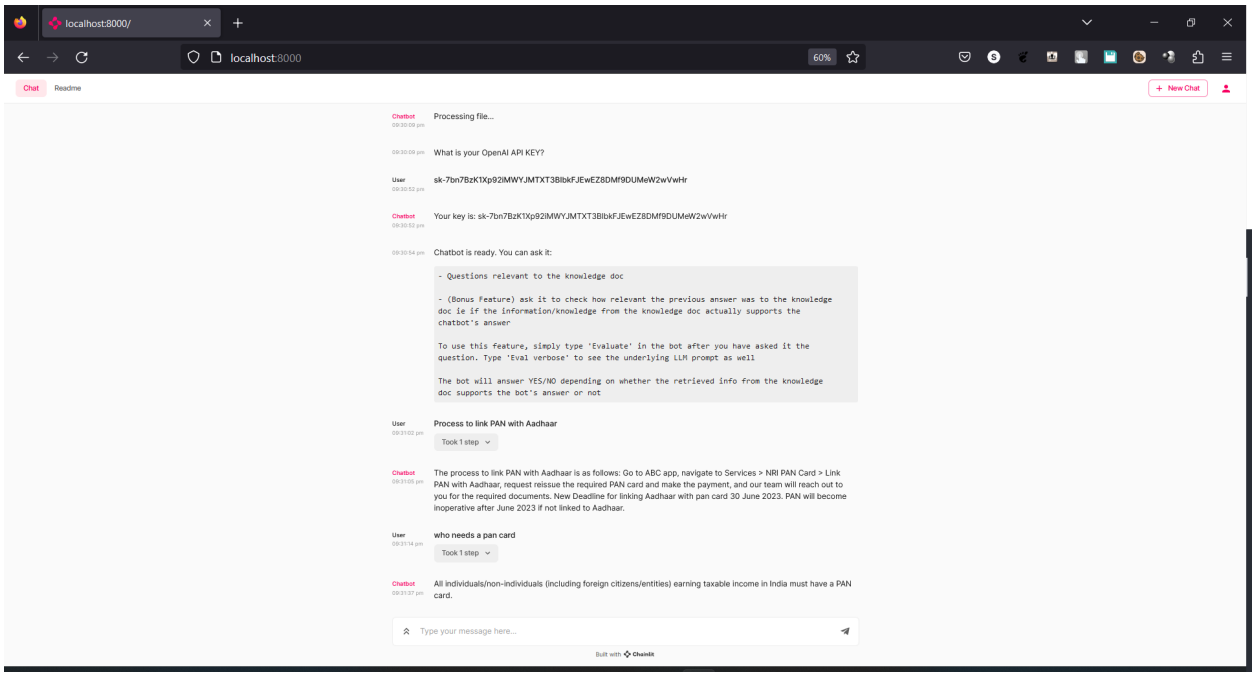
- **Evaluate response against query and context**: In this approach, we manually extract the relevant context was that retrieved from the vector store and provided to the LLM, then we check if the response supplied by the chatbot makes sense, given the supplied context and query. Under the hood, this approach uses few-shot learning on an LLM to rate the answer as correct or incorrect.

Upon inspection, one can see that the `RetrievalQA` chain uses `get_relevant_documents()` method of the retriever under the hood to retrieve the relevant context from the vector store. So, `get_relevant_documents()` of the FAISS retriever is used to is used to manually retrieve the relevant content and is then supplied to a simple LLMChain used for evaluation of the answer against the query and context by prompting the LLM in a few-shot learning context.

All of this is wrapped in a chainlit application to provide a frontend chatbot UI on localhost:8000

Solution

Bot in action:



(Space for diagrams)

Future Scope

- **Sequential chunk retrieval** : One of the problems visible in this model is that sometimes, context for a particular query is retrieved from many different locations since get_relevant_documents() loads multiple chunks under the hood and the LLMChain combines them to form the full context. This can be troublesome as there can be a sudden switch in context which can distort the semantic structure and underlying meaning of the retrieved knowledge entirely and can feed the LLM incorrect knowledge leading to hallucination. A workaround to this would be to implement a retrieval algorithm that selects the best sequence of chunks for the given query to form the context. This would also reduce hallucination since one of the sources of hallucination is actually broken/incorrect context
- **Optimising for document structure**: Since the given knowledge document has structure of it's own, one could fine tune the LLM by extracting question-answer pairs from the document and training the LLM on them.
- Better response evaluation techniques can be found