Brance <AI Applied Researcher intern>Task

Name:Arun Murali
Linkedin Profile: https://www.linkedin.com/in/pavan-kumar-c-n-375498205/
Date Challenge Received:29-07-23(29-07-23 work started)
Date Solution Delivered:30-07-23

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.

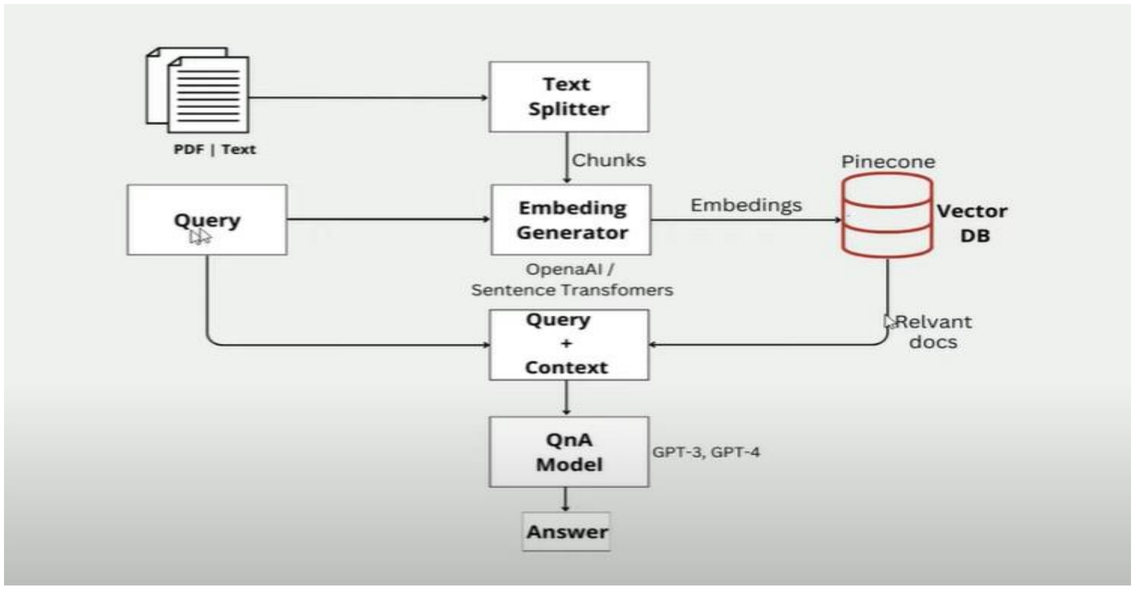
Deliverables:

The deliverables for the chatbot include:

- Working solution: This is the code for the chatbot that can be run on a local machine .
- 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.

Introduction

In the era of digital communication, chatbots have emerged as a powerful tool for businesses, organizations, and users alike. From handling customer service inquiries to providing interactive experiences, these AI-powered platforms are transforming the way we communicate and access information.

we will be delving into the technicalities of creating a highly efficient chatbot that can answer queries from its own documents or knowledge base. This chatbot, leveraging the power of advanced language models, can also respond to follow-up questions from the users, ensuring a seamless and interactive user experience.

Setting Up the Environment

Before diving into the actual process of building our chatbot, we first need to set up our development environment with the necessary libraries and tools. This ensures our code can execute successfully and our chatbot can function as intended.

The requirements.txt file contains a list of libraries and tools required for this project. Here's what it includes:

- **streamlit:** This library helps us to create interactive web apps for machine learning and data science projects.
- **streamlit_chat:** This Streamlit component is used for creating the chatbot user interface.
- **langchain:** This is a framework for developing applications powered by language models. It provides a standard interface for chains, lots of integrations with other tools, and end-to-end chains for common applications.
- **sentence_transformers:** This library allows us to use transformer models like BERT, RoBERTa, etc., for generating semantic representations of text (i.e., embeddings), which we'll use for our document indexing.
- **openai:** This is the official OpenAI library that allows us to use their language models, like GPT-3.5-turbo, for generating human-like text.
- **unstructured and unstructured[local-inference]:** These are used for document processing and managing unstructured data.
- **pinecone-client:** This is the client for Pinecone, a vector database service that enables us to perform similarity search on vector data.

To install all these libraries, you can run the following command in your terminal:

```
pip install -r requirements.txt
```

This command tells pip (Python's package installer) to install the libraries mentioned in the requirements.txt file.

With our environment set up, we can now proceed to the next step: indexing our documents in Pinecone.

Document Indexing

The next step in our journey to build the chatbot involves preparing and indexing the documents that our chatbot will utilize to answer queries. For this, we use the indexing.py script.

Loading documents from a directory with LangChain

The first step in the indexing.py script involves loading the documents from a directory. We use the DirectoryLoader class provided by LangChain to achieve this. This class accepts a directory as input and loads all the documents present in it.

```
from langchain.document_loaders import DirectoryLoader

directory = '/content/sample_data'

def load_docs(directory):
  loader = DirectoryLoader(directory)
  documents = loader.load()
  return documents

documents = load_docs(directory)
len(documents)
```

Splitting documents

After loading the documents, the script proceeds to split these documents into smaller chunks. The size of the chunks and the overlap between these chunks can be defined by the user. This is done to ensure that the size of the documents is manageable and that no relevant information is missed out due to the splitting. The RecursiveCharacterTextSplitter class from LangChain is used for this purpose.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

def split_docs(documents,chunk_size=500,chunk_overlap=20):
  text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size,
chunk_overlap=chunk_overlap)
  docs = text_splitter.split_documents(documents)
  return docs

docs = split_docs(documents)
print(len(docs))
Creating embeddings
```

Once the documents are split, we need to convert these chunks of text into a format that our AI model can understand. This is done by creating embeddings of the text using SentenceTransformerEmbeddings class provided by LangChain.

```
from langchain.embeddings import SentenceTransformerEmbeddings
embeddings = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")
```

Storing embeddings in Pinecone

After the embeddings are created, they need to be stored in a place from where they can be easily accessed and searched. Pinecone is a vector database service that is perfect for this task. The embeddings are stored in Pinecone using the Pinecone class from LangChain.

```
import pinecone
from langchain.vectorstores import Pinecone
pinecone.init(
    api_key="",  # find at app.pinecone.io
    environment="us-east-1-aws"  # next to api key in console
)
index_name = "langchain-chatbot"
index = Pinecone.from_documents(docs, embeddings, index_name=index_name)
```

The embeddings can then be accessed and searched using the similarity_search function provided by the Pinecone class.

```
def get_similiar_docs(query,k=1,score=False):
  if score:
    similar_docs = index.similarity_search_with_score(query,k=k)
  else:
    similar_docs = index.similarity_search(query,k=k)
  return similar_docs
```

This completes the process of document indexing, and we are now ready to move to the main application of our chatbot.

Building the Chatbot Application with Streamlit

With the indexed documents in place, the main part of our task is to build the chatbot application itself. We use Streamlit to create a seamless interactive interface for the chatbot. This involves constructing a user-friendly interface and ensuring the chatbot can process queries and provide responses. We accomplish this using the main.py file.

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import ConversationChain
from langchain.chains.conversation.memory import ConversationBufferWindowMemory
from langchain.prompts import (
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
    ChatPromptTemplate,
    MessagesPlaceholder
)
import streamlit as st
from streamlit_chat import message
from utils import *

st.subheader(" RAG_Chatbot ")

if 'responses' not in st.session_state:
    st.session_state['responses'] = ["How can I assist you?"]

if 'requests' not in st.session_state:
    st.session_state['requests'] = []

if 'buffer_memory' not in st.session_state:

st.session_state.buffer_memory=ConversationBufferWindowMemory(k=3,return_messages=Tru
e)


system_msg_template = SystemMessagePromptTemplate.from_template(template="""Answer
the question as truthfully as possible using the provided context, and if the answer is not
contained within the text below, say 'I don't know'""")


human_msg_template = HumanMessagePromptTemplate.from_template(template="{input}")

prompt_template = ChatPromptTemplate.from_messages([system_msg_template,
MessagesPlaceholder(variable_name="history"), human_msg_template])
```

1. **Session State Initialisation:** Firstly, we initialise two lists 'responses' and 'requests' within Streamlit's session state. These lists store the history of bot responses and user requests respectively.

2. **ConversationBufferWindowMemory:** This memory structure is instantiated with a size of 3, meaning that our chatbot would remember the last three interactions, keeping a manageable memory size for efficiency.

3. **PromptTemplate Construction:** We construct a PromptTemplate for our chatbot. The template contains instructions to the language model (LLM), providing structure and context to the input for the LLM to generate a response. Langchain provides different types of MessagePromptTemplate, which includes AIMessagePromptTemplate, SystemMessagePromptTemplate, and HumanMessagePromptTemplate, for creating different types of messages.

Creating the User Interface

The Streamlit library allows us to quickly build a user-friendly interface for our chatbot application. The st.title function is used to display the chatbot's title at the top of the interface. The user's queries and the chatbot's responses are displayed in a conversation format using the st.container and st.text_input functions.

```
st.title("RAG_Chatbot")
...
response_container = st.container()
textcontainer = st.container()
...
with textcontainer:
    query = st.text_input("Query: ", key="input")
    ...
with response_container:
    if st.session_state['responses']:
        for i in range(len(st.session_state['responses'])):
            message(st.session_state['responses'][i],key=str(i))
            if i < len(st.session_state['requests']):
                message(st.session_state["requests"][i], is_user=True,key=str(i)+ '_user')
```

Initializing the Language Model and Conversation

ChatOpenAI Initialisation: We then create an instance of ChatOpenAI, which utilizes the powerful gpt-3.5-turbo model from OpenAI for language understanding and generation.

ConversationChain Setup: Finally, we set up the ConversationChain using the memory, prompt template, and LLM we've prepared. The ConversationChain is essentially a workflow of how our chatbot would operate: it leverages user input, prompt template formatting, and the LLM to conduct an interactive chat.

```
llm = ChatOpenAI(model_name="gpt-3.5-turbo", openai_api_key="")
...
conversation = ConversationChain(memory=st.session_state.buffer_memory,
prompt=prompt_template, llm=llm, verbose=True)
```

Generating Responses

When the user inputs a query, the chatbot uses the predict method to generate a response. The response is then displayed in the chat interface.

```
if query:
    with st.spinner("typing..."):
        ...
        response = conversation.predict(input=f"Context:\n {context} \n\n Query:\n{query}")
    st.session_state.requests.append(query)
    st.session_state.responses.append(response)
```

This creates a seamless interaction where the user can ask questions and get responses from the chatbot. We will delve into the refining of queries and finding matches in the next section.

Refining Queries and Finding Matches with Utility Functions
Once our chatbot is operational, we need to ensure that it can effectively process user queries and find relevant responses. This is achieved through a set of utility functions defined in utils.py. Here, we describe the purpose of these functions and their roles in the application.

Refining Queries with OpenAI

The query_refiner function is used to take the user's query and refine it to ensure it's optimal for providing a relevant answer. It uses OpenAI's DaVinci model to refine the query based on the current conversation log.

```
def query_refiner(conversation, query):
    response = openai.Completion.create(
    model="text-davinci-003",
    prompt=f"Given the following user query and conversation log, formulate a question that
would be the most relevant to provide the user with an answer from a knowledge
base.\n\nCONVERSATION LOG: \n{conversation}\n\nQuery: {query}\n\nRefined Query:",
    temperature=0.7,
    max_tokens=256,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0
    )
    return response['choices'][0]['text']
```

Finding Matches in Pinecone Index

The find_match function is used to find the most relevant documents that match the user's query. It uses the Pinecone vector index to find matches and returns the most relevant text.

```
def find_match(input):
    input_em = model.encode(input).tolist()
    result = index.query(input_em, top_k=2, includeMetadata=True)
    return result['matches'][0]['metadata']['text']+"\n"+result['matches'][1]['metadata']['text']
```

Tracking the Conversation

The get_conversation_string function is used to keep track of the ongoing conversation. It generates a string of the conversation log, including both the user's queries and the chatbot's responses.

```
def get_conversation_string():
    conversation_string = ""
    for i in range(len(st.session_state['responses'])-1):
        conversation_string += "Human: "+st.session_state['requests'][i] + "\n"
        conversation_string += "Bot: "+ st.session_state['responses'][i+1] + "\n"
    return conversation_string
```

With these utility functions, the chatbot can not only generate responses but also refine the user's queries and find the most relevant answers. This ensures a more effective and user-friendly chatbot experience.

All that’s left is to launch the script:

4. Future Scope

Thoughts on how you could have improved the solution.

Due to time constraints iam 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.