Query Your PDF Application

(Chat with Multiple PDFs)

Group 5 :

Aditya Bommireddipalli (100892785),

Harpreet kaur (100901899),

Ayesha sarah (100907046))

Introduction:

Query Your PDF is a web application that allows you to chat with multiple PDF documents. You can ask questions about the PDFs using natural language, and the application will provide relevant responses based on the content of the documents. This app utilizes a language model to generate accurate answers to your queries. The app will only respond to questions related to the loaded PDFs.

1. Problem Identification:

The AI solution aims to address the issue of effectively interacting with and extracting pertinent data from PDF documents. Although PDFs are frequently used to share information, accessing, and comprehending their content can take some time. This issue is significant because it can enable users to interact with PDFs seamlessly, which will speed up information retrieval and analysis. By offering a user-friendly interface that enables natural language queries to be posted to PDF documents, AI can be useful in solving this problem by enabling quick and accurate access to information.

2. Language Model Selection:

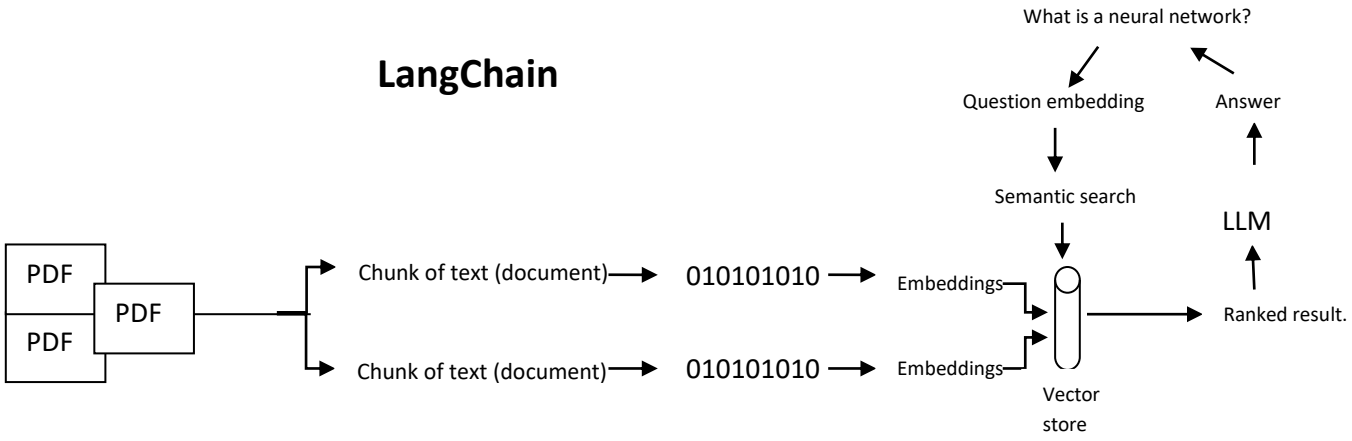
The "**text-davinci-003**" model was chosen for this AI product because it is well suited to the issue of understanding and interacting with natural language. Because ChatGPT models have a rate limit we preferred using this model from **OpenAI** to produce human-like responses based on input text, they are appropriate for use in conversational interfaces. The ChatGPT family of models (e.g., GPT-3.5, GPT-4), or from the language models available from **Hugging Face** can also be used for the same purpose and we have made the application in such a way that it is customizable to change the models when required. This model's capacity to produce coherent and contextually relevant text makes it an appropriate option given the emphasis on interaction and real-time responses.

```python
def get_conversation_chain(vector_store):
    llm= OpenAI(
            temperature=0.6,
            model_name="text-davinci-003"
        )

    memory=ConversationBufferMemory(memory_key='chat_history',return_messages=True)
    conversation_chain = ConversationalRetrievalChain.from_llm(
        llm=llm,
        retriever=vector_store.as_retriever(),
        memory=memory
    )
    return conversation_chain
```

3. AI Product Design:

The AI product is a integrated chatbot application that makes use of the **LangChain** framework. The chatbot will accurately and pertinently respond to any questions or queries users may enter regarding the information contained in their PDF documents. The front-end is created as an interactive web application using a tool called **Streamlit.io**, while the back end involves making API calls to the **OpenAI** model using the LangChain framework. The chatbot can be used by users to interact, upload PDFs, and view the outcomes.



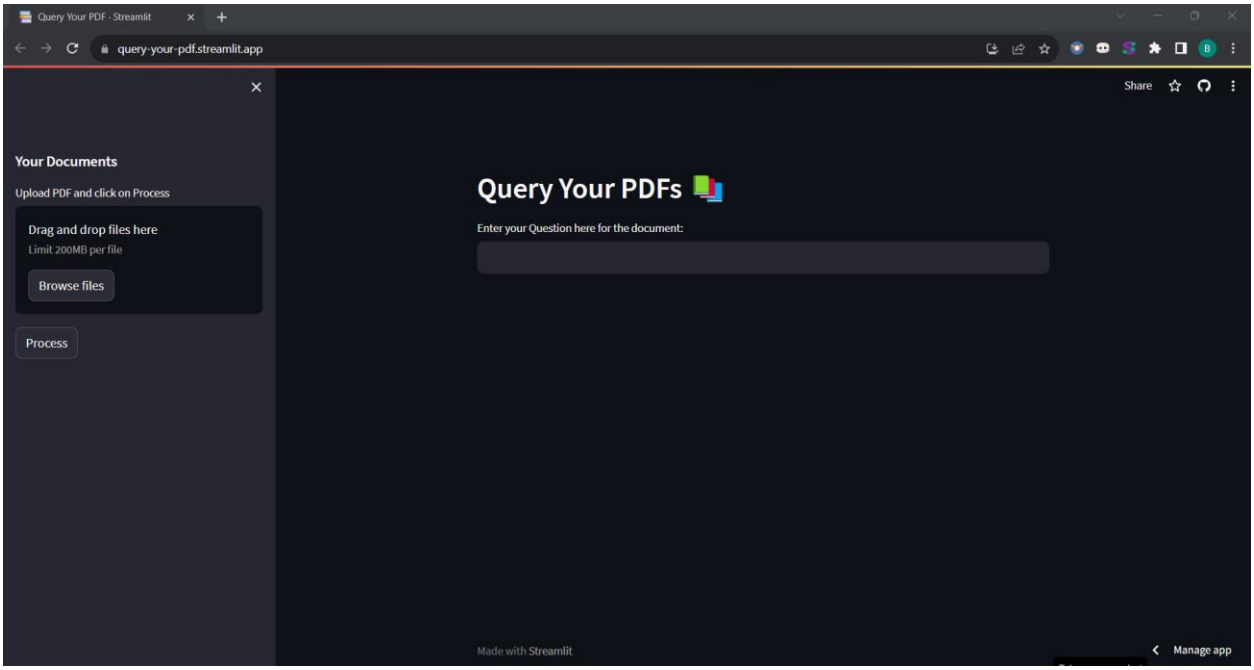
The application follows these steps to provide responses to your questions:

- **PDF Loading:** The app reads multiple PDF documents and extracts their text content.
- **Text Chunking:** The extracted text is divided into smaller chunks that can be processed effectively.

- **Language Model:** The application utilizes a language model to generate vector representations (embeddings) of the text chunks.
- **Similarity Matching:** When you ask a question, the app compares it with the text chunks and identifies the most semantically similar ones.
- **Response Generation:** The selected chunks are passed to the language model, which generates a response based on the relevant content of the PDFs.

4. Front-end Application:

Users can communicate with the AI chatbot through the user-friendly interface of the front-end application. Users can upload their PDF files, ask questions about the content, and get prompt answers from the chatbot. Users can interact with their PDFs and the AI chatbot in an aesthetically pleasing and simple manner thanks to the interface, which can be created using Streamlit.io.



5. Technical Explanation:

The AI product consists of several technical elements. Using libraries like PyPDF2, the PDF documents are first pre-processed to extract text. The front-end user interface and the API of the model are connected using the **LangChain** framework. According to the chatbot's architecture, user inquiries are sent to the **OpenAI** model, which then produces responses based on the input. To make the model more

suitable for the specific issue of responding to inquiries about PDF content, some fine-tuning may be done. Tokenization, model inference, and text generation are all part of the interaction flow.

5.1 Import Statements:

In this section, you're importing the required libraries and modules that are necessary for your application to function properly. These include **streamlit, dotenv, PdfReader** from **PyPDF2**, and various modules from the **Langchain** library for language processing tasks.

```
import streamlit as st
from dotenv import load_dotenv
from PyPDF2 import PdfReader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings,HuggingFaceInstructEmbeddings
from langchain.vectorstores import FAISS
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationalRetrievalChain
from langchain import OpenAI
from htmlTemplates import css,bot_template,user_template
```

5.2 Getting the PDF text:

This function takes a list of PDF files as input and extracts text from each page of each PDF file using the **PdfReader** class from **PyPDF2**.

It then concatenates the extracted text and returns it as a single string.

```
def get_text_from_pdf(pdf_files):
    text= ""
    for pdf in pdf_files:
        pdf_reader = PdfReader(pdf)
        for page in pdf_reader.pages:
            text += page.extract_text()
    return text
```

5.3 Get the Text Chunks from PDFs:

This function takes a text input and uses the **CharacterTextSplitter** class from the **Langchain** library to split the text into smaller chunks.

It returns a list of text chunks.

```
def get_text_chunks(text):
    text_chunk = CharacterTextSplitter(
        separator="\n",
        chunk_size=1000,
        chunk_overlap=200,
        length_function=len
        )
    chunks = text_chunk.split_text(text)
    return chunks
```

5.4 Creating Vector Store for these embeddings:

This function takes a text chunk as input and creates vector embeddings using the **OpenAIEmbeddings** class from the **Langchain** library.

It then creates a vector store using the **FAISS** library for efficient similarity search using these embeddings and returns the vector store.

```
def get_vector_embeddings(text_chunk):
    embeddings = OpenAIEmbeddings()
    vectorstore = FAISS.from_texts(texts=text_chunk, embedding=embeddings)
    return vectorstore
```

5.5 Creating a conversation chain:

This function sets up a conversational chain using the OpenAI language model (llm) and the vector store created using the get_vector_embeddings function.

It also initializes a memory buffer to keep track of the conversation history. The function returns the initialized conversation chain.

```
def get_conversation_chain(vector_store):
    llm= OpenAI(
            temperature=0.6,
            model_name="text-davinci-003"
        )

    memory=ConversationBufferMemory(memory_key='chat_history',return_messages=True)
    conversation_chain = ConversationalRetrievalChain.from_llm(
        llm=llm,
        retriever=vector_store.as_retriever(),
        memory=memory
    )
    return conversation_chain
```

5.6 Handling the User Input:

This function handles the user's input question and generates a response using the conversation chain.

It uses the **st.session_state** to store and retrieve conversation history and the conversation chain. The Latest user input is used to query the conversation chain and get a response.

The function then extracts the latest messages from the conversation history, including the user's question and the bot's answer, and displays them using HTML templates.

```
def handle_userInput(user_question):
    response = st.session_state.conversation({'question': user_question})
    st.session_state.chat_history = response['chat_history']

    # Reverse the chat history list to display the latest question first
    reversed_chat_history = reversed(st.session_state.chat_history)

    latest_messages = list(reversed_chat_history)[:2]  # Get the last two messages
    # Display the latest question and answer
    if len(latest_messages) >= 2:
        bot_message = latest_messages[0]
        user_message = latest_messages[1]

        st.write(user_template.replace("{{MSG}}", user_message.content), unsafe_allow_html=True)
        st.write(bot_template.replace("{{MSG}}", bot_message.content), unsafe_allow_html=True)
```

5.7 Main Function:

This is the main function where the app's user interface and logic are defined.

It sets up the **Streamlit** page configuration, imports HTML templates for styling, and displays the app's header and user input field.

The user's question is taken as input, and the conversation history and conversation chain are initialized if not present. The sidebar allows users to upload PDF files and process them to generate the vector store and conversation chain.

When a user question is submitted, the **handle_userInput** function is called to generate a response and display it.

```python
def main():
    # To load .env settings
    load_dotenv()

    if 'conversation' not in st.session_state:
        st.session_state.conversation = None
    if 'chat_history' not in st.session_state:
        st.session_state.chat_history = None
    #Setting the page Configuration
    st.set_page_config(page_title="Query Your PDF",page_icon=":books:")
    st.write(css,unsafe_allow_html=True)

    st.header("Query Your PDFs :books:")
    user_question = st.text_input("Enter your Question here for the document:")

    # Initialize chat history and new conversation flag in session state
    if 'chat_history' not in st.session_state:
        st.session_state.chat_history = []
    if 'conversation' not in st.session_state:
        st.session_state.conversation = get_conversation_chain(vector_store)

    with st.sidebar:
        st.subheader("Your Documents")
        pdf_files= st.file_uploader("Upload PDF and click on Process",accept_multiple_files=True)
        if st.button("Process"):
            with st.spinner("Loading..."):
                    # 1. Get the PDF text
                    raw_text = get_text_from_pdf(pdf_files)
                    # 2. Get the Text Chunks from PDFs
                    text_chunks = get_text_chunks(raw_text)
                    # 3. Create Vector Store for these embeddings
                    vector_store = get_vector_embeddings(text_chunks)
                    # 4. Creating a conversation chain
                    st.session_state.conversation = get_conversation_chain(vector_store)

    if user_question:
            handle_userInput(user_question)
```

6. Evaluation Metrics:

These could be appropriate metrics to use when assessing the performance of the AI product.

- **Accuracy:** The proportion of correct responses that the chatbot provides.
- **Response Time:** The typical amount of time the chatbot needs to respond.
- **User satisfaction:** User comments on the chatbot's efficiency and usability.
- **Error Rate:** The proportion of responses that were incorrect or irrelevant.
- **Comparison with Baseline:** Evaluation is measured against a straightforward baseline based on rules to identify improvements.

7. Limitations and Ethical Considerations:

The AI product may have the following limitations.

- **Reliance on Text:** The effectiveness of the chatbot depends on how well and accurately text from PDFs is extracted.
- **Complex Questions:** With complex or ambiguous questions, the chatbot might have trouble responding.
- **Model Biases:** The selected language model may display biases in its results, necessitating careful monitoring and mitigation.

8. Ethics-related factors include:

- **Bias Mitigation:** Measures taken to lessen responses that are biased and to ensure the dissemination of accurate and impartial information.
- **Privacy:** Ensuring that user information (uploaded PDFs) is handled securely and isn't abused.
- Clearly letting users know when they are interacting with an AI chatbot is referred to as transparency.
- **Accountability:** Addressing any potential moral or legal dilemmas brought on using the chatbot.

9. Challenges and Future Enhancements:

- **Rate Limits:** The application faces challenges related to API rate limits when interacting with OpenAI. Implementing strategies like caching, efficient API usage, and error handling can help mitigate these issues.
- **Advanced Conversational Models:** Integrating more advanced conversational models and NLP techniques could lead to even more accurate and informative responses.
- **User Customization:** Future enhancements could include options for users to customize the behavior and responses of the application based on their preferences.

Summary:

The AI product tackles the problem of effective interaction with PDF documents by utilizing the **LangChain** framework and **OpenAI** model's capabilities. It offers a useful solution while considering its ethical and practical limitations.

The presented application bridges the gap between document processing and natural language interactions. By enabling users to engage in meaningful conversations while querying PDF documents, the application showcases the potential of NLP and document processing integration for improved user experiences in various domains.