Chat with PDF Documents Using Streamlit and Gemini AI

yash.s.halwai

This Streamlit application allows users to interact with PDF documents by asking questions and receiving detailed responses. The application uses advanced technologies like Google Generative AI for text embeddings and the FAISS library for creating a vector store to enable efficient document searching and question answering. Below is a detailed explanation of the code and its functionalities.

Detailed Explanation:

#Import Necessary Libraries
```python
import streamlit as st
from PyPDF2 import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter
import os
from langchain_google_genai import GoogleGenerativeAIEmbeddings
import google.generativeai as genai
from langchain_community.vectorstores import FAISS
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.chains.question_answering import load_qa_chain
from langchain.prompts import PromptTemplate
from dotenv import load_dotenv
```
These libraries include Streamlit for the web interface, PyPDF2 for PDF reading, and LangChain and Google Generative AI for creating text embeddings and handling AI models.

#Load Environment Variables
```python
load_dotenv()
GOOGLE_API_KEY = os.getenv("GOOGLE_API_KEY")
if GOOGLE_API_KEY:
   genai.configure(api_key=GOOGLE_API_KEY)
else:
   raise ValueError("GOOGLE_API_KEY environment variable is not set")
```
The `load_dotenv` function loads the Google API key from an environment file to authenticate requests to Google's AI services.

#Extract Text from PDFs
```python
def get_pdf_text(pdf_docs):
    text = ""
    for pdf in pdf_docs:
        pdf_reader = PdfReader(pdf)
        for page in pdf_reader.pages:
            text += page.extract_text()
    return text
```
This function extracts text from uploaded PDF documents, looping through each page and concatenating the extracted text.

#Split Text into Chunks
```python
def get_text_chunks(text):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=10000, chunk_overlap=1000)
    chunks = text_splitter.split_text(text)
    return chunks
```
The text is split into smaller chunks using a recursive splitter to ensure efficient processing and retrieval.

#Create and Save Vector Store
```python
def get_vector_store(text_chunks):
    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")
    vector_store = FAISS.from_texts(text_chunks, embedding=embeddings)
    vector_store.save_local("faiss_index")
```
This function creates a vector store using text embeddings for efficient similarity searches. The vector store is then saved locally.

#Create Conversational Chain
```python
def get_conversational_chain():
    prompt_template = """
        Answer the question as detailed as possible from the provided context, make sure to
provide all the details, if the answer is not in
```

```
    provided context just say, "answer is not available in the context", don't provide the wrong answer\n\n
    Context:\n {context}?\n
    Question: \n{question}\n
    Answer:
    """
    model = ChatGoogleGenerativeAI(model="gemini-pro", temperature=0.3)
    prompt = PromptTemplate(template=prompt_template, input_variables=["context", "question"])
    chain = load_qa_chain(model, chain_type="stuff", prompt=prompt)
    return chain
```

This function sets up a Q&A model with a specific prompt template to handle user questions and generate accurate answers.

#Handle User Input

```python
def user_input(user_question):
    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")
    new_db = FAISS.load_local("faiss_index", embeddings, allow_dangerous_deserialization=True)
    docs = new_db.similarity_search(user_question)
    chain = get_conversational_chain()
    response = chain({"input_documents": docs, "question": user_question}, return_only_outputs=True)
    print(response)
    st.write("Reply: ", response["output_text"])
```

This function processes the user's question by searching the vector store for relevant documents and using the conversational chain to generate a response.

#Main Function for Streamlit App

```python
def main():
    st.set_page_config(page_title="Chat PDF")
    st.header("Chat with PDF using Gemini")
    user_question = st.text_input("Ask a Question from the PDF Files")
    if user_question:
        user_input(user_question)
    with st.sidebar:
        st.title("Menu:")
        pdf_docs = st.file_uploader("Upload your PDF Files and Click on the Submit & Process Button", accept_multiple_files=True)
        if st.button("Submit & Process"):
            with st.spinner("Processing..."):
                raw_text = get_pdf_text(pdf_docs)
                text_chunks = get_text_chunks(raw_text)
                get_vector_store(text_chunks)
                st.success("Done")
```

This is the main function that sets up the Streamlit app. It includes a text input for user questions, a sidebar for PDF file uploads, and processes the PDFs when the "Submit & Process" button is clicked.

#Run the Application

```python
if __name__ == "__main__":
    main()
```

This ensures the main function runs when the script is executed.

Conclusion

This application leverages advanced AI technologies to provide an interactive way to query PDF documents. Users can upload PDFs, ask questions, and receive detailed answers based on the document content. The integration of Streamlit makes the application user-friendly and accessible.

Connect With Me:

LinkedIn: https://in.linkedin.com/in/yash-sanjeev-halwai-198007203
GitHub: https://github.com/YashHalwai
YouTube: https://www.youtube.com/@codeyaa
Instagram: https://www.instagram.com/yash.s.halwai/?igsh=MzZrb2k2MjJ5ZnNp