CHAT-PDF with LANG-CHAIN and GEMINI

- **Tools:**
  1. Python
  2. VSCode
  3. Python virtual Environment

- **Pre-requisites:**
  1. Create a python virtual environment
     → Install it through python –m venv <myevnv>
  2. Dependencies for Project
     - streamlit
     - google-generativeai
     - python-dotenv
     - langchain
     - PyPDF2
     - faiss-cpu
     - langchain_google_genai
     - langchain-community

     → Install these using pip install <Dependencies>

- **Packages and its Importance:**
  1. ***streamlit as st:***
     → Streamlit is a framework for building interactive web applications in Python. The alias st is used to call its functions, which are typically used to create the user interface of the app.
  2. ***from PyPDF2 import PdfReader:***
     → PyPDF2 is a library that allows you to read PDF files in Python. The PdfReader class is used to extract text from PDF documents.
  3. ***from langchain.text_splitter import RecursiveCharacterTextSplitter:***
     → LangChain is a framework for building applications that work with large language models (LLMs). The RecursiveCharacterTextSplitter is a utility in LangChain that splits text into manageable chunks based on specified character limits, ensuring that the text chunks are neither too large nor too small for processing.
  4. ***import os:***
     → The os module provides functions to interact with the operating system, such as accessing environment variables, working with file paths, and performing file I/O operations.

5. ***from langchain_google_genai import GoogleGenerativeAIEmbeddings:***
   → This module provides an interface to Google's Generative AI for generating embeddings, which are numerical representations of text that can be used in various natural language processing (NLP) tasks.
6. ***import google.generativeai as genai:***
   → This is the main module for interacting with Google's Generative AI API, enabling the configuration and use of the AI model for generating text, embeddings, or other related tasks.
7. ***from langchain_community.vectorstores import FAISS:***
   → FAISS (Facebook AI Similarity Search) is a library for efficient similarity search and clustering of dense vectors. The FAISS module in LangChain allows you to store and search through these vectors, which is useful for finding relevant pieces of text in large datasets.
8. ***from langchain_google_genai import ChatGoogleGenerativeAI:***
   → This module provides an interface to Google's Generative AI models, specifically tailored for conversational applications. It is used to generate responses in a chat-like format.
9. ***from langchain.chains.question_answering import load_qa_chain:***
   → This module provides pre-built chains for question answering, which orchestrate how the model processes input, retrieves relevant information, and generates answers.
10. ***from langchain.prompts import PromptTemplate:***
    → This module allows you to create custom prompts that can be used to control the input provided to the AI model. The PromptTemplate is used to define how the questions and context are presented to the model.
11. ***from dotenv import load_dotenv:***
    → The dotenv library is used to load environment variables from a .env file. This is often used to securely manage API keys and other sensitive configuration information.

Thoroughly analyzing and interpreting a document is just as important as reviewing it. In today's digital world, we have many tools to assist with this task, and one of the most powerful tools is AI. For our project, we will use an AI model to help with this work. While there are numerous AI tools available across the internet, we are opting to use Google's Gemini AI Services for its top-level security, ease of use, and seamless integration.
The Gemini AI API Key can be obtained from the MakerSuite website, which provides the necessary authorization to access and use Gemini AI Services.

Code Analysis:

1. When a PDF document is uploaded, the code processes and extracts the text from each page of the PDF. The **PdfReader** class from the **PyPDF2** library is responsible for reading the PDF file page by page. As it reads each page, the text is extracted and stored in a variable called text.
2. To manage the extracted text efficiently, it is divided into smaller chunks of 10,000 tokens, with an overlap of 1,000 tokens between chunks. This chunking process is handled by the **RecursiveCharacterTextSplitter** from the **LangChain** library. The purpose of this step is to ensure that the text is split into manageable pieces, which can later be processed more effectively by the AI model.
3. Once the text is divided into chunks, these chunks are converted into vectors using a model from the **GoogleGenerativeAIEmbeddings** library. Specifically, the **embedding-001** model is employed for this task. Converting text into vectors allows the AI to perform various operations, such as similarity searches, by representing the text in a numerical form that the model can understand.
4. The generated vectors are stored in a local database using **FAISS** (Facebook AI Similarity Search). FAISS is a library that efficiently handles similarity searches and clustering of dense vectors, making it an ideal choice for storing and retrieving the vectorized text data locally.
5. At this point, the model has been provided with knowledge in the form of vectorized text chunks. To interact with the model and retrieve information, a conversational chain is established. This is done by defining a prompt that instructs the AI on how to respond to user queries. The **ChatGoogleGenerativeAI** class is used to specify the model being used in this case, the "**Gemini-pro**" model with a moderate temperature setting to control the randomness of the AI's responses.
6. When a user asks a question, the system processes the question by converting it into tokens and then into vectors using the same embedding technique. The **FAISS** system then performs a similarity search with the stored vectors to find the most relevant text chunks. If a match is found, the system retrieves and returns the relevant information. If no match is found, the system responds with "**answer is not available in the context**" ensuring that no incorrect or unrelated information is provided. This process ensures that the responses are strictly based on the content of the uploaded document.
7. This is how the system works for a single document, allowing users to interact with and query the content of a PDF file in a meaningful and accurate way.

Code Review

→

```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
import zipfile
import tempfile
```

- Importing all required package's

→

```python
load_dotenv()
os.getenv("GOOGLE_API_KEY")
genai.configure(api_key=os.getenv("GOOGLE_API_KEY"))
```

- Here, we need to get the Google AI API key, and then store it in the **.env** file which should be in the same directory.

```python
def get_all_pdfs_from_folder(folder_path: str):
    pdf_docs = []
    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.lower().endswith('.pdf'):
                pdf_docs.append(os.path.join(root, file))
    return pdf_docs
```

→

- From the uploaded ZIP file, it traverses through the subfolders, if there are any, and saves all the PDFs in **pdf_docs.**

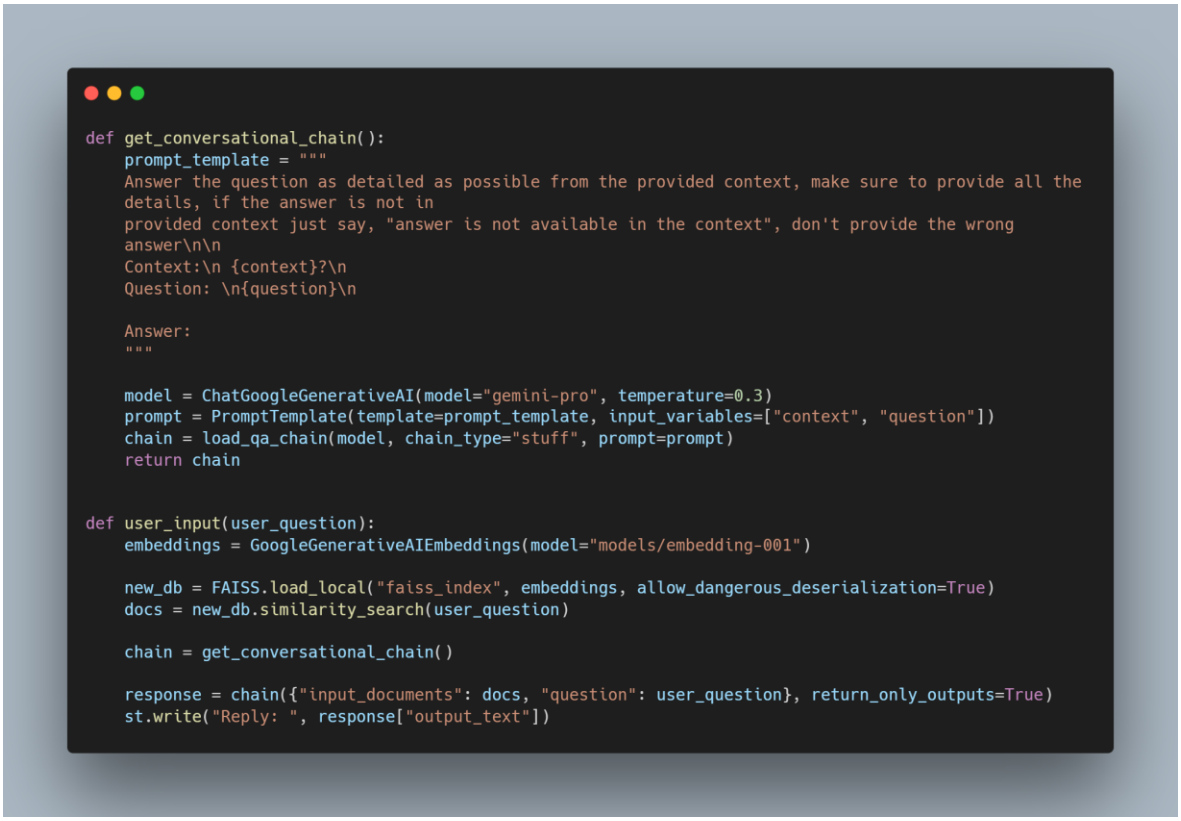
```python
def get_pdf_text(pdf_docs):
    text = ""
    for pdf in pdf_docs:
        pdf_reader = PdfReader(pdf)
        for page in pdf_reader.pages:
            text += page.extract_text()
    return text


def get_text_chunks(text):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=10000,
chunk_overlap=1000)
    chunks = text_splitter.split_text(text)
    return chunks


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")
```

- Refer the **<u>Code Analysis [1-4]</u>**

→



```
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


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)
    st.write("Reply: ", response["output_text"])
```

- Refer **Code Analysis [5-7]**

→ The remaining **def main (),** function is about the front-end part which is readable and understandable.