# Documentation for Streamlit PDF QA App with Langchain Framework using Google Gemini AI

Debajyoti Maity

November 4, 2024

## 1 Introduction

This document explains each line of the Python code for a Streamlit app that allows users to ask questions from uploaded PDF files, using Google Gemini AI for generating answers.

## 2 Code Explanation

```
import streamlit as st
```

Imports the `Streamlit` library, which is used to build the user interface for the app.

```
from PyPDF2 import PdfReader
```

Imports the `PdfReader` class from the `PyPDF2` library, which is used to extract text from PDF documents.

```
from langchain.text_splitter import
    RecursiveCharacterTextSplitter
```

Imports the `RecursiveCharacterTextSplitter` class from the `Langchain` library, which is used to split the extracted text into manageable chunks for processing.

```
import os
```

Imports the `os` module to interact with the operating system, especially for handling environment variables.

```
from langchain_google_genai import
    GoogleGenerativeAIEmbeddings
```

Imports the `GoogleGenerativeAIEmbeddings` class from the `langchain_google_genai` library to handle text embeddings using Google Gemini.

```
import google.generativeai as genai
```

Imports the `google.generativeai` library to configure and interact with the Google Generative AI (Gemini).

```
from langchain.vectorstores import FAISS
```

Imports the `FAISS` vector store from `Langchain`, used to store and search through vector embeddings of the text chunks.

```
from langchain_google_genai import
    ChatGoogleGenerativeAI
```

Imports the `ChatGoogleGenerativeAI` class from `langchain_google_genai`, which is used to interact with the Google Generative AI (Gemini) in a chat-like manner.

```
from langchain.chains.question_answering import
    load_qa_chain
```

Imports the `load_qa_chain` function from `Langchain` to create a chain that handles question-answering tasks.

```
from langchain.prompts import PromptTemplate
```

Imports the `PromptTemplate` class from `Langchain`, which allows the creation of custom prompts for question answering.

```
from dotenv import load_dotenv
```

Imports the `load_dotenv` function to load environment variables from a `.env` file, which is where the API key is stored.

```
# Load environment variables
load_dotenv()
api_key = os.getenv("GOOGLE_API_KEY")
```

Loads the API key stored in the environment variable `GOOGLE_API_KEY` using the `dotenv` library.

```
# Configure the Google Generative AI
genai.configure(api_key=api_key)
```

Configures the Google Generative AI by passing the API key obtained from the environment variables.

```
# Function to extract text from PDF documents
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
```

Defines the function `get_pdf_text` that takes uploaded PDF files, reads them using `PdfReader`, and extracts the text from all pages.

```
# Function to split text into chunks
def get_text_chunks(text):
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=10000, chunk_overlap=1000)
    chunks = text_splitter.split_text(text)
    return chunks
```

Defines the function `get_text_chunks` which splits large blocks of text into smaller chunks of 10,000 characters with 1,000 characters of overlap between chunks. This is done using the `RecursiveCharacterTextSplitter`.

```
# Function to create and save vector store
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")
```

Defines the function `get_vector_store` that takes the text chunks, converts them to embeddings using Google Generative AI, and stores them in a FAISS vector index. The index is saved locally as `faiss_index`.

```
# Function to get conversational chain
def get_conversational_chain():
    prompt_template = """
        Answer the question in a detailed and
            structured way using bullet points to
            ensure clarity and easy understanding.
        Do not provide the answer in paragraph form.
            Use bullet points to organize the
            information effectively.
        If the answer is not available in the provided
             context, simply state: "The answer is not
            available in the context."

        Context:\n {context}\n
        Question: \n{question}\n

        Answer:
    """
    model = ChatGoogleGenerativeAI(model="gemini-pro",
         temperature=0.1)
    prompt = PromptTemplate(template=prompt_template,
        input_variables=["context", "question"])
    chain = load_qa_chain(model, chain_type="stuff",
        prompt=prompt)
```

```
    return chain
```

Defines the function `get_conversational_chain`, which sets up a custom prompt template for answering questions from the PDF content using bullet points. The `gemini-pro` model from Google Generative AI is used for generating the responses.

```
# Function to handle user input and get response
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"])
```

Defines the function `user_input`, which handles user input (a question), retrieves relevant document chunks from the FAISS vector store, passes them to the conversational chain, and outputs the response to the user.

```
# Main function for Streamlit app
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")
    st.button("Get Answer:")
    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")
```

Defines the main function `main` for the Streamlit app. It allows users to upload PDF files, ask questions, and receive answers. The sidebar contains a file uploader, and the text input field accepts user questions.

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

This is the entry point of the application. When the script is executed, it runs the `main` function to launch the Streamlit app.