Building a Multi PDF RAG Chatbot: Langchain, Streamlit with code : -

---

bit.ly/pdfChatbot

https://blog.gopenai.com/building-a-multi-pdf-rag-chatbot-langchain-streamlit-with-code-d21d0a1cf9e5

Setting the Stage with Necessary Tools :-

- The application begins by importing various powerful libraries:
- **Streamlit:** Used to create the web interface.
- **PyPDF2:** A tool for reading PDF files.
- **Langchain:** A suite of tools for natural language processing and creating conversational AI.
- **FAISS(Facebook):** A library for efficient similarity search of vectors, which is useful for finding information quickly in large datasets.

```
!pip install streamlit
!pip install langchain
!pip install PyPDF2
!pip install dotenv
!pip install streamlit langchain PyPDF2 dotenv
!pip install longchain_community
!pip install langchain_anthropic langchain_openai
!pip install python-dotenv
```

```
import streamlit as st
from PyPDF2 import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_core.prompts import ChatPromptTemplate
from langchain_community.embeddings.spacy_embeddings import SpacyEmbeddings
from langchain_community.vectorstores import FAISS
from langchain.tools.retriever import create_retriever_tool
from dotenv import load_dotenv
from langchain_anthropic import ChatAnthropic
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain.agents import AgentExecutor, create_tool_calling_agent

import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

Once the text is extracted, it is split into manageable chunks:
Text Splitter: Using the Langchain library, the text is divided into chunks of 1000 characters each. This segmentation helps in processing and analyzing the text more efficiently.

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

def get_chunks(text):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
    chunks = text_splitter.split_text(text)
    return chunks
```

Creating a Searchable Text Database and Making Embeddings :-

To make the text searchable, the application converts the text chunks into vector representations:
-**Vector Store:** The application uses the **FAISS library** to turn text chunks into vectors and saves these vectors locally. This transformation is crucial as it allows the system to perform fast and efficient searches within the text.

```
embeddings = SpacyEmbeddings(model_name="en_core_web_sm")

def vector_store(text_chunks):
    vector_store = FAISS.from_texts(text_chunks, embedding=embeddings)
    vector_store.save_local("faiss_db")
```

SpacyEmbeddings is a class used for generating embeddings from text using a specified spaCy model. Here's a detailed explanation of how it works, using the en_core_web_sm model as an example:

Model Initialization:
When you initialize SpacyEmbeddings with model_name="en_core_web_sm", it loads the specified spaCy model. The en_core_web_sm model is a small English language model provided by spaCy, which includes tokenization, part-of-speech tagging, dependency parsing, named entity recognition, and word vectors (embeddings).

Text Processing:
The text you want to generate embeddings for is processed by the spaCy model. This involves several steps:

Tokenization: The text is broken down into individual tokens (words, punctuation, etc.).
Part-of-Speech Tagging: Each token is assigned a part of speech.
Dependency Parsing: The syntactic structure of the sentence is analyzed to understand the relationships between tokens.
Named Entity Recognition: Named entities (such as people, organizations, dates) in the text are identified.
Word Vectors: Each token is mapped to a pre-trained word vector (embedding).

Generating Embeddings:
The embeddings are generated by the spaCy model's word vectors. These embeddings are numerical representations of words that capture semantic meaning. The embeddings for individual words can be combined (e.g., averaged) to create a single embedding for a larger chunk of text (such as a sentence or document).

Using the Embeddings:
The generated embeddings can be used for various downstream tasks such as text classification, clustering, semantic similarity, or as input to other machine learning model.

Setting Up the Conversational AI:-

The core of this application is the conversational AI, which uses OpenAI's powerful models:

- AI Configuration: The app sets up a conversational AI using OpenAI's GPT model. This AI is designed to answer questions based on the PDF content it has processed.

- Conversation Chain: The AI uses a set of prompts to understand the context and provide accurate responses to user queries. If the answer to a question isn't available in the text, the AI is programmed to respond with "answer is not available in the context," ensuring that users do not receive incorrect information.

```
    llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0, api_key="")
    prompt = ChatPromptTemplate.from_messages([...])
    tool=[tools]
    agent = create_tool_calling_agent(llm, tool, prompt)
    agent_executor = AgentExecutor(agent=agent, tools=tool, verbose=True)
    response=agent_executor.invoke({"input": ques})
    print(response)
    st.write("Reply: ", response['output'])

def user_input(user_question):
    new_db = FAISS.load_local("faiss_db", embeddings,allow_dangerous_deserialization=True)
    retriever=new_db.as_retriever()
    retrieval_chain= create_retriever_tool(retriever,"pdf_extractor","This tool is to give answer to
queries from the pdf")
    get_conversational_chain(retrieval_chain,user_question)
```

##
OR for example :-

```
def get_conversational_chain(tools,ques):
    #os.environ["ANTHROPIC_API_KEY"]=os.getenv["ANTHROPIC_API_KEY"]
    #llm = ChatAnthropic(model="claude-3-sonnet-20240229", temperature=0,
api_key=os.getenv("ANTHROPIC_API_KEY"),verbose=True)
    llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0, api_key="")
    prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """You are a helpful assistant. 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""",
        ),
        ("placeholder", "{chat_history}"),
        ("human", "{input}"),
        ("placeholder", "{agent_scratchpad}"),
    ]
)
    tool=[tools]
    agent = create_tool_calling_agent(llm, tool, prompt)

    agent_executor = AgentExecutor(agent=agent, tools=tool, verbose=True)
    response=agent_executor.invoke({"input": ques})
    print(response)
    st.write("Reply: ", response['output'])
```

User Interaction :-

With the backend ready, the application uses Streamlit to create a user-friendly interface:

- User Interface: Users are presented with a simple text input where they can type their questions related to the PDF content. The application then displays the AI's responses directly on the web page.
- File Uploader and Processing: Users can upload new PDF files anytime. The application processes these files on the fly, updating the database with new text for the AI to search.

```
def main():
   st.set_page_config("Chat PDF")
   st.header("RAG based Chat with PDF")
   user_question = st.text_input("Ask a Question from the PDF Files")
   if user_question:
        user_input(user_question)
    with st.sidebar:
        pdf_doc = 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 = pdf_read(pdf_doc)
                text_chunks = get_chunks(raw_text)
                vector_store(text_chunks)
                st.success("Done")
```
