Chatbots Machine Learning Project

Streamlit-based customer support chatbot that processes and classifies customer inquiries from uploaded PDF documents using vector search and language model embeddings.

1. Imports and Initialization

- Essential libraries like Streamlit, PyPDF2, LangChain, and dotenv are imported for app interface, text extraction from PDFs, text processing, embedding creation, and environment variable handling.
- The Google API key is loaded from an .env file and configured with genai.configure(api_key=api_key).

2. PDF Text Extraction

- **Function**: get_pdf_text
- **Description**: Reads uploaded PDF files page by page, extracting all text content and returning it as a single string.
- **Purpose**: Collects and prepares raw text data from PDFs for further processing.

3. Text Splitting

- **Function**: get_text_chunks
- **Description**: Splits the extracted text into chunks of manageable size (10,000 characters with a 1,000-character overlap), using RecursiveCharacterTextSplitter.
- **Purpose**: Breaks down large text blocks into smaller, contextually manageable parts for improved semantic search accuracy.

4. Vector Store Creation

- **Function**: get_vector_store
- **Description**: Uses the GoogleGenerativeAIEmbeddings model to convert text chunks into embeddings, then stores these embeddings in a FAISS vector store. The vector store is saved locally as "faiss_index" for easy loading and querying.
- **Purpose**: Enables fast similarity search for retrieving relevant document sections based on user queries.

5. Conversational Chain for Intent Classification

- **Function**: get_conversational_chain
- **Description**: Sets up a classification model to identify intents in customer support queries based on provided contexts. It uses the Gemini Pro model with a low temperature (0.3) to ensure coherent and accurate intent classification.
- **Prompt Template**: Guides the model to categorize queries into specified intents (e.g., change_shipping_address, check_cancellation_fee, cancel_order, or cancel), using contextual examples.
- **Purpose**: Enables the chatbot to classify customer inquiries into predefined categories, enhancing the relevance of responses.

6. User Query Handling

- **Function**: user_input
- **Description**: Loads the FAISS vector store, searches for similar text chunks to the user's query, and uses the conversational chain to classify the query intent.
- **Purpose**: Provides a response by retrieving relevant sections from the PDF and classifying the user's question.

7. Main Function and Streamlit Interface

- **Function**: main
- **Description**: Sets up the Streamlit UI with the header, question input field, and sidebar for PDF uploads. When PDFs are uploaded and the "Submit & Process" button is clicked, it extracts text, splits it into chunks, and creates a vector store.
- **Purpose**: Creates an interactive interface for users to upload PDFs, ask questions, and receive responses.

Overall Workflow

1. Users upload PDFs.
2. Text is extracted, chunked, embedded, and saved in a FAISS vector store.
3. When a question is entered, the vector store retrieves relevant sections.
4. The model classifies the intent and generates a response.

Potential Improvements

- **Error Handling**: Additional checks can be added for file handling, embeddings, and vector store loading.
- **Security**: Add a way to securely manage the API key without exposing it in logs or code.
- **Scalability**: Consider cloud storage for the FAISS vector index in cases of large datasets.

Report Summary

The code is structured effectively for a customer support chatbot that uses text extraction, vector search, and intent classification to answer user queries. It provides a reliable and extensible approach to querying and processing PDF-based documents.

2.CODING

Platform: Visual Studio and python code

2.1).env file GOOGLE_API_KEY = “AIzaSyCqzUPzJJAu5Q4KJQ84SaL3vtD6D9WsgNM”

2.2)requiremet.txt file

- Streamlit
- google-generativeai
- python-dotenv
- langchain
- PyPDF2
- faiss-cpu
- fastapi
- langchain
- _google_genai

2.3)app.py file

1. Imports and Environment Setup

```
python
Copy code
import streamlit as st
from PyPDF2 import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter
import os
import hashlib
from langchain_google_genai import GoogleGenerativeAIEmbeddings
import google.generativeai as genai
from langchain.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
```

- Streamlit is imported to build the web interface for the chatbot.
- PyPDF2 allows reading and extracting text from PDF files.
- LangChain modules handle text splitting, embeddings, vector storage, and prompt templates.
- dotenv is used to load environment variables securely, particularly for API keys.

```
load_dotenv()
api_key = os.getenv("GOOGLE_API_KEY")
if api_key:
    genai.configure(api_key=api_key)
else:
    st.error("Google API key not found. Please check your .env file.")
```

- load_dotenv() loads the .env file where sensitive keys (like GOOGLE_API_KEY) are stored.
- The API key is retrieved and configured for use with Google’s generative AI models.

- If the key isn’t found, an error is shown in Streamlit.

2. Text Extraction from PDF

```
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
```

- **Purpose**: This function reads each PDF file page-by-page and extracts the text.
- It iterates through pdf_docs (a list of PDF files), reads each page, and appends the extracted text to a text string.

3. Text Splitting into Chunks

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

- **Purpose**: Splits long text into smaller chunks for efficient vector storage and retrieval.
- chunk_size and chunk_overlap define the chunk length and overlap between chunks, respectively, which improves retrieval accuracy.

4. Vector Store Creation

```
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")
```

- **Purpose**: Converts text chunks into embeddings (numerical representations) and stores them in a FAISS vector database.
- GoogleGenerativeAIEmbeddings converts text chunks to embeddings, and FAISS.from_texts stores these embeddings in a FAISS vector store.
- The vector store is saved as "faiss_index" for future queries.

5. Conversational Chain for Intent Classification

```
def get_conversational_chain():
    prompt_template = """
    Classify each customer support query into the appropriate intent category based on the context provided.
    Use the specified intent categories: `change_shipping_address`, `check_cancellation_fee`, `cancel_order`, and
`cancel`.
    Follow the examples given and select the correct intent based on the customer's inquiry.
    If no intent matches, respond with "intent not available in the context".

    Context:
```

```
    - change_shipping_address: Queries about modifying the delivery address.
    - check_cancellation_fee: Requests for information on early termination or cancellation fees.
    - cancel_order: Requests related to canceling a specific order.
    - cancel: General questions on exiting or stopping a service.

    Examples:
    - "I have an issue changing the shipping address."
      Intent: `change_shipping_address`
    - "I don't know what I need to do to check the early exit fees."
      Intent: `check_cancellation_fee`
    - "Assistance to check the early termination penalty."
      Intent: `check_cancellation_fee`
    - "I cannot find the early exit penalties; can you help me?"
      Intent: `cancel`
    - "I need help canceling the last order I made."
      Intent: `cancel_order`
    - "How can I cancel my recent order?"
      Intent: `cancel_order`

    Query:
    {context}

    Intent:
    """

    model = ChatGoogleGenerativeAI(model="gemini-pro", temperature=0.3)
    prompt = PromptTemplate(template=prompt_template, input_variables=["context"])

    chain = load_qa_chain(model, chain_type="stuff", prompt=prompt)

    return chain
```

- **Purpose**: Creates a prompt and classification chain that categorizes customer support queries based on pre-defined intents.
- ChatGoogleGenerativeAI is initialized with the gemini-pro model and a temperature of 0.3 for more deterministic responses.
- A PromptTemplate is defined, specifying context as the variable input, guiding the model to classify intents.
- The prompt structure helps the model follow rules to match user queries to intents accurately.

6. User Input Handling and Querying the Vector Store

```
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"])
```

- **Purpose**: Handles user input, retrieves similar documents from the vector store, and uses the classification chain to generate a response.

- FAISS.load_local loads the previously saved vector store to perform similarity searches on the user question.
- It retrieves relevant documents, then uses get_conversational_chain() to classify the intent of the user question based on context.
- The response is displayed in Streamlit.

7. Main Function and Streamlit Interface

```
def main():
    st.set_page_config(page_title="Chat PDF")
    st.header("Customer Support Chatbot")

    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"):
            if pdf_docs:
                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("Processing completed!")
            else:
                st.warning("Please upload at least one PDF file.")
```

- **Purpose**: Sets up the Streamlit interface with options to upload PDFs, enter questions, and display results.
- The sidebar allows users to upload multiple PDFs and displays a button to process them.
- When clicked, it extracts text from PDFs, splits it, and creates the FAISS vector store.
- The main section includes an input field where users can ask questions based on the PDF content.

8. Execution

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

- **Purpose**: Runs the main function, launching the Streamlit application when the script is executed.

Summary

The code integrates PDF text extraction, chunking, vector-based search, and intent classification in a Streamlit app. It retrieves information from uploaded PDFs, allows users to ask questions, and provides categorized responses.