**SOURCE CODE**

%pwd # Displays the current working directory.

import os

os.chdir("../") # Moves up one directory level.

%pwd # Displays the updated working directory.

from langchain.document\_loaders import PyPDFLoader, DirectoryLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

# Extract data from all PDF files in the specified directory.

def load\_pdf\_file(data):

loader = DirectoryLoader(data, glob="\*.pdf", loader\_cls=PyPDFLoader) # Loads only .pdf files from the directory.

documents = loader.load() # Processes and loads PDF content into document objects.

return documents

extracted\_data = load\_pdf\_file(data='Data/') # Load all PDFs from the 'Data/' directory.

# Split the loaded data into smaller text chunks with overlap for context preservation.

def text\_split(extracted\_data):

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=20) # Splits data into 500-character chunks with 20-character overlap.

text\_chunks = text\_splitter.split\_documents(extracted\_data) # Returns the split chunks.

return text\_chunks

text\_chunks = text\_split(extracted\_data) # Split the extracted data into text chunks.

print("Length of Text Chunks", len(text\_chunks)) # Display the number of generated text chunks.

from langchain.embeddings import HuggingFaceEmbeddings

# Download and initialize Hugging Face embeddings for text vectorization.

def download\_hugging\_face\_embeddings():

embeddings = HuggingFaceEmbeddings(model\_name='sentence-transformers/all-MiniLM-L6-v2') # Pre-trained transformer for text embeddings.

return embeddings

embeddings = download\_hugging\_face\_embeddings() # Initialize the embedding model.

query\_result = embeddings.embed\_query("Hello world") # Create an embedding for a test query.

print("Length", len(query\_result)) # Display the size of the generated embedding vector.

from dotenv import load\_dotenv

load\_dotenv() # Load environment variables from a .env file.

# Get Pinecone and OpenAI API keys from environment variables.

PINECONE\_API\_KEY = os.environ.get('PINECONE\_API\_KEY')

OPENAI\_API\_KEY = os.environ.get('OPENAI\_API\_KEY')

from pinecone.grpc import PineconeGRPC as Pinecone

from pinecone import ServerlessSpec

import os

pc = Pinecone(api\_key=PINECONE\_API\_KEY) # Initialize the Pinecone client using the API key.

index\_name = "medicalbot" # Define the name of the Pinecone index.

# Create a new Pinecone index for storing document embeddings with cosine similarity as the metric.

pc.create\_index(

name=index\_name,

dimension=384, # Embedding vector dimension.

metric="cosine", # Use cosine similarity for comparisons.

spec=ServerlessSpec(cloud="aws", region="us-east-1") # Specify serverless deployment details.

)

import os

os.environ["PINECONE\_API\_KEY"] = PINECONE\_API\_KEY # Set Pinecone API key as an environment variable.

os.environ["OPENAI\_API\_KEY"] = OPENAI\_API\_KEY # Set OpenAI API key as an environment variable.

# Embed each document chunk and upload the embeddings into the Pinecone index.

from langchain\_pinecone import PineconeVectorStore

docsearch = PineconeVectorStore.from\_documents(

documents=text\_chunks, # Document chunks to be embedded and stored.

index\_name=index\_name, # Target index in Pinecone.

embedding=embeddings, # Embedding model for vectorizing text.

)

# Load an existing Pinecone index and initialize it for retrieval tasks.

from langchain\_pinecone import PineconeVectorStore

docsearch = PineconeVectorStore.from\_existing\_index(

index\_name=index\_name, # Name of the existing index.

embedding=embeddings # Embedding model used during index creation.

)

retriever = docsearch.as\_retriever(search\_type="similarity", search\_kwargs={"k": 3}) # Retrieve top-3 similar documents for a query.

retrieved\_docs = retriever.invoke("What is Acne?") # Query the index for relevant documents about acne.

retrieved\_docs # Display the retrieved documents.

from langchain\_openai import OpenAI

llm = OpenAI(temperature=0.4, max\_tokens=500) # Initialize the OpenAI model with a specified temperature and max token limit.

from langchain.chains import create\_retrieval\_chain

from langchain.chains.combine\_documents import create\_stuff\_documents\_chain

from langchain\_core.prompts import ChatPromptTemplate

# Define a system prompt for generating concise and accurate answers based on retrieved context.

system\_prompt = (

"You are an assistant for question-answering tasks. "

"Use the following pieces of retrieved context to answer "

"the question. If you don't know the answer, say that you "

"don't know. Use three sentences maximum and keep the "

"answer concise."

"\n\n"

"{context}"

)

prompt = ChatPromptTemplate.from\_messages(

[

("system", system\_prompt), # System-level instructions.

("human", "{input}"), # Human query placeholder.

]

)

# Create a question-answering chain that combines retrieved documents and user input.

question\_answer\_chain = create\_stuff\_documents\_chain(llm, prompt)

rag\_chain = create\_retrieval\_chain(retriever, question\_answer\_chain) # Retrieval-augmented generation chain.

response = rag\_chain.invoke({"input": "what is Acromegaly and gigantism?"}) # Query the chain about specific medical conditions.

print(response["answer"]) # Display the generated answer.

response = rag\_chain.invoke({"input": "What is stats?"}) # Query the chain about statistics.

print(response["answer"]) # Display the generated answer.