Hackathon: Pharma Knowledge Assistant — ESA

Large Language Models Hackathon - UE21CS421AC1

**Team 27**

PES2UG21CS411 Rachana Binumohan

PES2UG21CS415 Rahul H R

PES2UG21CS408 R S Gowri

PES2UG21CS403 Punith Anand

PES2UG21CS402 Priyansh Gupta

**Introduction**

This report documents the design, implementation, and results of the Pharma Knowledge Assistant developed during the LLM Hackathon. The system is built to provide intelligent, context-aware responses to user queries regarding pharmaceutical products, leveraging advanced LLM technologies, Retrieval-Augmented Generation (RAG), and modular agent-based architecture.

**Design**

1. **Dataset Preparation**

Web Scraping:

Extracted prescribing information from a pharmaceutical website using requests and BeautifulSoup.

Data was saved in structured JSON files, each containing product-specific details (e.g., dosage, indications, contraindications).

Preprocessing:

Standardized and segmented documents into manageable chunks using a RecursiveCharacterTextSplitter with overlaps to preserve context.

Embeddings and Vector Database:

Generated vector representations using SentenceTransformer.

Stored embeddings in a FAISS (Facebook AI Similarity Search) vector database for fast, context-aware querying.

2. **System Architecture**

Agent-Based Design:

Implemented distinct agent nodes for each functionality (QA, Summarizer, Recommendation, Alternatives).

A centralized dispatcher (dispatch\_agent) routed queries to the appropriate nodes.

Retrieval-Augmented Generation (RAG):

Integrated RAG with LangChain to retrieve context-specific document chunks and generate responses.

Streamlit-Based GUI:

Developed a user-friendly chatbot interface allowing real-time interactions with the assistant.

3. **Features Implemented**

Question Answering (QA):

Processed user queries and retrieved relevant chunks to generate fact-based answers.

Example: "What are the side effects of Amoxicillin?"

Recommendation System:

Provided warnings and recommendations based on symptoms or medication history.

Example: "Can I take Ibuprofen if I have a history of stomach ulcers?"

Alternatives Generator:

Suggested alternative medications or treatments based on user queries.

Example: "What are alternatives to Paracetamol for pain relief?"

Summarization:

Produced concise summaries for lengthy pharmaceutical product descriptions.

Example: Summarizing all indications and usage for Amoxicillin.

Error Handling:

Implemented robust error resilience to provide user feedback in case of failures.

**Key Results**

1. **Dataset Expansion and Query Efficiency**

Expanded the dataset from 10 to over 100 pharmaceutical products.

Contextual retrieval ensured precise answers, reducing unnecessary LLM token usage.

2. **Feature Evaluation**

Question Answering:

Achieved high accuracy by grounding responses in retrieved chunks.

Recommendations:

Delivered actionable advice tailored to user conditions.

Example Output: "Consider Acetaminophen for pain relief if you have a history of stomach ulcers."

Summarization:

Generated clear and concise summaries for quick user understanding.

Alternatives Generator:

Provided reliable and context-aware alternative suggestions.

Example: "Alternatives to Ibuprofen include Acetaminophen and Naproxen."

3. **GUI and User Interaction**

Developed a Streamlit-based chatbot allowing users to:

Input queries in natural language.

View structured, context-aware responses.

Access dynamic follow-up recommendations.

### **Conclusion**

The Pharma Knowledge Assistant successfully integrates agent-based design with RAG to deliver an intuitive and intelligent solution for pharmaceutical queries. Its robust architecture, efficient query handling, and user-friendly interface make it a scalable and versatile tool for real-world applications. The assistant's design and implementation effectively balance accuracy, efficiency, and user engagement.

**Source Code:**

1.Loading json and Saving vector store:

import os

import json

from langchain.vectorstores import FAISS

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.schema import Document

# Load JSON dataset

def load\_dataset(json\_folder: str):

"""

Load all JSON files from a folder into a list of Document objects.

Extract relevant fields and combine them into a single text document.

"""

if not os.path.exists(json\_folder):

raise FileNotFoundError(f"The dataset folder '{json\_folder}' does not exist.")

documents = []

for filename in os.listdir(json\_folder):

if filename.endswith(".json"):

file\_path = os.path.join(json\_folder, filename)

try:

with open(file\_path, 'r') as f:

content = json.load(f)

# Define relevant fields to extract and combine

relevant\_fields = [

"HIGHLIGHTS OF PRESCRIBING INFORMATION",

"1 INDICATIONS AND USAGE",

"2 DOSAGE AND ADMINISTRATION",

"3 DOSAGE FORMS AND STRENGTHS",

"4 CONTRAINDICATIONS",

"5 WARNINGS AND PRECAUTIONS",

"6 ADVERSE REACTIONS",

"7 DRUG INTERACTIONS",

"8 USE IN SPECIFIC POPULATIONS",

"10 OVERDOSAGE",

"11 DESCRIPTION",

"12 CLINICAL PHARMACOLOGY",

"13 NONCLINICAL TOXICOLOGY",

"14 CLINICAL STUDIES",

"15 REFERENCES",

"16 HOW SUPPLIED/STORAGE AND HANDLING",

"17 PATIENT COUNSELING INFORMATION",

"PACKAGE LABEL.PRINCIPAL DISPLAY PANEL",

"INGREDIENTS AND APPEARANCE"

]

# Combine text from relevant fields

combined\_text = "\n".join(

content.get(field, "") for field in relevant\_fields if field in content

)

# Include metadata

metadata = {

"product\_name": content.get("product\_name", ""),

"source\_file": filename,

"page": content.get("page", ""),

}

# Add to documents if combined text is not empty

if combined\_text.strip():

documents.append(Document(page\_content=combined\_text, metadata=metadata))

else:

print(f"Skipping file '{file\_path}' as it contains no relevant text.")

except json.JSONDecodeError:

print(f"Error decoding JSON in file: {filename}")

except Exception as e:

print(f"Unexpected error with file {filename}: {e}")

return documents

# Create FAISS vector store

def create\_vector\_store(documents, api\_key: str):

"""

Create a FAISS vector store for RAG using OpenAI embeddings.

"""

if not api\_key or api\_key == "your\_openai\_api\_key":

raise ValueError("A valid OpenAI API key is required. Set the key as an environment variable or pass it explicitly.")

embeddings = OpenAIEmbeddings(openai\_api\_key=api\_key)

vectorstore = FAISS.from\_documents(documents, embeddings)

return vectorstore

# Main script execution

if \_\_name\_\_ == "\_\_main\_\_":

# Ensure OpenAI API key is set

API\_KEY = os.getenv("OPENAI\_API\_KEY")

if not API\_KEY:

raise ValueError("OPENAI\_API\_KEY is not set. Please provide a valid OpenAI API key.")

# Set dataset path

json\_folder = "/content/dataset" # Update with your dataset folder path

try:

# Load documents

docs = load\_dataset(json\_folder)

print(f"Loaded {len(docs)} documents from '{json\_folder}'.")

# Create vector store if documents are loaded

if docs:

vector\_store = create\_vector\_store(docs, API\_KEY)

print("Vector store created successfully!")

else:

print("No valid documents found. Vector store creation aborted.")

except Exception as e:

print(f"Error: {e}")

2.Complete langgraph RAG workflow with GUI integration:

%%writefile app.py

import requests

import json

import re

from typing import Dict, Optional, Any, Tuple

from pydantic import BaseModel, Field, ValidationError, model\_validator

from langchain.tools import tool

from langchain.vectorstores import FAISS

from langchain.embeddings import OpenAIEmbeddings

from langchain.callbacks.manager import CallbackManager

from langchain.callbacks.streaming\_stdout import StreamingStdOutCallbackHandler

from langchain\_core.runnables import RunnableSequence

import streamlit as st # Import Streamlit

# Configuration

LLM\_SERVER\_URL = "https://7dfa-122-172-84-113.ngrok-free.app"

ENDPOINT = f"{LLM\_SERVER\_URL}/v1/chat/completions"

def query\_llm(prompt: str) -> str:

"""

Send a query to the hosted LLM.

"""

payload = {

"model": "llama-3.2-1b-instruct",

"messages": [

{"role": "system", "content": "You are a helpful assistant."},

{"role": "user", "content": prompt}

],

"temperature": 0.7

}

headers = {"Content-Type": "application/json"}

try:

response = requests.post(ENDPOINT, json=payload, headers=headers)

response.raise\_for\_status() # Raise an error for HTTP codes 4xx or 5xx

data = response.json()

# Adjust response parsing based on your API format

return data["choices"][0]["message"]["content"]

except requests.exceptions.RequestException as e:

raise Exception(f"Request failed: {e}")

except KeyError:

raise Exception(f"Unexpected response structure: {response.text}")

class QueryAnalysis(BaseModel):

symptoms: str = Field(default="", description="Extracted symptoms from the query")

condition: str = Field(default="", description="Extracted medical condition")

drugs: str = Field(default="", description="Extracted drug names")

context: Optional[str] = Field(default=None, description="Additional context")

@classmethod

def parse\_response(cls, response: str) -> 'QueryAnalysis':

"""

Robustly parse the LLM response into a QueryAnalysis object

"""

# Debugging print to see the raw response

print("Raw response:", response)

# Preprocessing: Remove code blocks and extra whitespace

response = response.strip('`{}').strip()

try:

# First, try JSON parsing

try:

parsed\_dict = json.loads(response)

return cls(\*\*parsed\_dict)

except json.JSONDecodeError:

# If JSON parsing fails, try more flexible parsing

parsed\_dict = {}

# Extract key-value pairs using regex with more flexible matching

keys\_to\_extract = [

('symptoms', r'symptoms?:\s\*"?([^"\n]+)"?'),

('condition', r'conditions?:\s\*"?([^"\n]+)"?'),

('drugs', r'drugs?:\s\*"?([^"\n]+)"?'),

('context', r'contexts?:\s\*"?([^"\n]+)"?')

]

for key, pattern in keys\_to\_extract:

match = re.search(pattern, response, re.IGNORECASE)

if match:

parsed\_dict[key] = match.group(1).strip()

print("KEYS:", keys\_to\_extract)

# Fallback to empty values if no matches

return cls(\*\*parsed\_dict)

except Exception as e:

print(f"Parsing error: {e}")

# Fallback to default if all parsing fails

return cls()

API\_KEY="<ADD YOUR API KEY>"

# Initialize embeddings (you can replace with your preferred embedding method)

embeddings = OpenAIEmbeddings(openai\_api\_key=API\_KEY)

# Load the existing FAISS vector store from disk

vector\_store\_path = "/content/faiss\_vector\_store"

vector\_store = FAISS.load\_local(vector\_store\_path, embeddings, allow\_dangerous\_deserialization=True)

@tool

def analyze\_query\_tool(query: str) -> str:

"""

Analyze the query to extract key entities.

"""

prompt = f"""

Read the following medical query: '{query}' and give final ouput as only a JSON .

First, classify it into one of these types such as a or b or c etc and dont print it:

a. direct queries or general queries

b. recommendations or warnings

c. summarize product details

d. references to related products

e. agent-based interaction design

Then, extract key entities such as symptoms, conditions, drugs, and context.

Return the output as only a VALID JSON with these keys present in {query}:

{{

"type": "one of the above types",

"symptoms": "extracted symptoms",

"condition": "extracted medical condition",

"drugs": "extracted drug names",

"context": "optional additional context"

}}

"""

# Use the query\_llm function directly

response = query\_llm(prompt)

try:

print("AWGWAGWG",response)

# analysis = json.loads(response)

return response

except json.JSONDecodeError:

raise Exception(f"LOUDA JSON response: {response}")

@tool

def retrieve\_relevant\_data\_tool(analysis: str) -> Dict[str, Any]:

"""

Retrieve relevant medical data based on the analyzed query using the vector store.

"""

print("in retrieve relevant data", analysis)

analysis=json.loads(analysis)

query\_type = analysis.get("type", "")

# Construct a standardized search query for similarity scoring

if analysis.get("type", "") == "c":

search\_query = f"Product details related to symptoms: {analysis.get('symptoms', '')}, condition: {analysis.get('condition', '')}, and drugs: {analysis.get('drugs', '')}."

elif analysis.get("type", "") == "d":

search\_query = f"References to related products for symptoms: {analysis.get('symptoms', '')}, condition: {analysis.get('condition', '')}, and drugs: {analysis.get('drugs', '')}."

elif analysis.get("type", "") == "b":

search\_query = f"Medical recommendations or warnings for symptoms: {analysis.get('symptoms', '')}, condition: {analysis.get('condition', '')}, and drugs: {analysis.get('drugs', '')}."

elif analysis.get("type", "") == "a":

search\_query = f"General medical information for symptoms: {analysis.get('symptoms', '')}, condition: {analysis.get('condition', '')}, and drugs: {analysis.get('drugs', '')}."

else:

search\_query = f"Relevant medical information for symptoms: {analysis.get('symptoms', '')}, condition: {analysis.get('condition', '')}, and drugs: {analysis.get('drugs', '')}."

# Perform similarity search and get only the most similar document

docs = vector\_store.similarity\_search(search\_query, k=3)

context = " ".join(doc.page\_content for doc in docs)

# Truncate context to a manageable length (e.g., 3000 characters)

context = context[:3000]

print("CHECKING CONTEXT:", context)

return json.dumps({"context": context, "analysis": analysis})

@tool

def generate\_recommendation\_tool(data: str) -> str:

"""

Generate a answer using the retrieved context and the LLM.

"""

data = json.loads(data)

context = data.get("context", "")

analysis = data.get("analysis", {})

query\_type = analysis.get("type", "")

# Tailor the LLM prompt based on the query type

if query\_type == "b":

prompt = f"""

Based on the following context:

{context}

Generate medical recommendations or warnings relevant to the query.

"""

elif query\_type == "c":

prompt = f"""

Based on the following product details:

{context}

Provide a concise summary of the product.

"""

elif query\_type == "d":

prompt = f"""

Based on the following context:

{context}

Provide references to related products.

"""

elif query\_type == "a":

prompt = f"""

Based on the following context:

{context}

Generate a general medical response relevant to the query.

"""

else:

prompt = f"""

Based on the following context:

{context}

Generate an appropriate response for the query type: {query\_type}.

"""

# Query the LLM with the tailored prompt

recommendation = query\_llm(prompt)

print("Generated Recommendation:", recommendation)

return recommendation

# Correct the chaining of functions

rag\_chain = RunnableSequence(

analyze\_query\_tool |

retrieve\_relevant\_data\_tool |

generate\_recommendation\_tool

)

# Set the title of the Streamlit app

st.title("Medical Query Recommendation System")

# Create a text input for the user to enter their query

query = st.text\_input("Enter your medical query:")

# Create a button to submit the query

if st.button("Get Result"):

if query:

try:

# Invoke the RAG chain with the user's query

recommendation = rag\_chain.invoke(query)

# Display the recommendation

st.success("Result:")

st.write(recommendation)

except Exception as e:

st.error(f"An error occurred: {e}")

else:

st.warning("Please enter a query.")

3. Hosting the streamlit serving using ngrok:

from pyngrok import ngrok

import os

# Set your ngrok authtoken

ngrok\_authtoken = "<YOUR NGROK AUTH" # Replace with your actual ngrok authtoken

os.environ["NGROK\_AUTHTOKEN"] = ngrok\_authtoken

# Kill any existing ngrok tunnels

ngrok.kill()

# Start ngrok tunnel

public\_url = ngrok.connect(8501) # Use integer port number

print(f"Streamlit app is running at: {public\_url}")

# Run the Streamlit app

!streamlit run app.py &>/dev/null&

**Screenshot of GUI:**

