NATURAL LANGUAGE PROCESSING
AI 829

IIITB

Disease Diagnosis.

Karnadevsinh Zala(MT2023188)

Somesh Awasthi(MT2023172)

Mandate 4

Brief Description

Update on the project so far:

In the course of this project, we explored various models to enhance our system's capabilities. We experimented with two embedding models, namely:

- BAAI/bge-large-en-v1.5
- sentence-transformers/all-MiniLM-L6-v2

However, the performance of these models did not meet our expectations when compared to the OpenAI embedding model.

Additionally, we incorporated TheBloke/Llama-2-7B-Chat-GGML as a transformer model into our system. Despite its potential, the retrieval process with this model proved to be slow, and the generated responses exhibited a higher error rate. Consequently, we concluded that gpt-3.5-turbo would better suit our requirements.

Moreover, we endeavored to implement a multimodal Retrieval-Augmented Generation (RAG) approach, which involved embedding both textual and tabular data to generate vector embeddings.

Subsequently, we sought to evaluate the efficacy of our RAG implementation. While there exists no standardized methodology for evaluating RAG systems, we conducted an initial assessment by presenting ten disease symptoms sourced from Google to our RAG model. The model successfully identified seven diseases, with two inaccuracies and one response labeled as inconclusive.

Furthermore, we explored the Giskard library, which facilitates RAG evaluation by generating questions from a knowledge base (utilizing OpenAI) and comparing the generated answers with ground truth responses. However, due to the library's restrictive data format requirements, we were unable to employ it for our evaluation, as our intention was to assess our system's performance on the "Disease Symptom Prediction" dataset sourced from Kaggle.

To facilitate a comprehensive evaluation, we developed a custom evaluation model. This model populates a dataset with actual diseases and corresponding symptoms, subsequently predicting diseases based on provided symptoms. The results of this evaluation will be appended to this report at a later stage.

Regarding the user interface design, both team members contributed by creating their respective interfaces. Karna utilized Streamlit to develop the user interface, while I implemented the backend using Flask, with the front-end design based on a template obtained from Google. Although not obligatory, we opted to explore new technologies as part of our learning process.

In conclusion, this project has been an enriching experience. We extend our gratitude to Professor Srinath Srinivasa for his initial guidance, as well as to our teaching assistants, particularly Dhruv and Atul, for their continuous support and assistance throughout this endeavor. Additionally, we appreciate the valuable input provided by our classmates.

Brief Recap of Previous Mandate

Step 1: we have install all the necessary dependencies

```
%%capture
!pip install langchain langchain-openai
!pip install ctransformers sentence-transformers langchain-chroma
!pip install pandas nltk spacy PyPDF
%pip install --upgrade --quiet  sentence-transformers langchain-chroma langchain langchain-openai > /dev/null
```

Step 2: we load the data for embedding

```
from langchain.document_loaders import PyPDFLoader, DirectoryLoader
# Load documents from PDF
loader = DirectoryLoader(path, glob="*.pdf", loader_cls=PyPDFLoader)
documents = loader.load()
```

Step 3: we do all the necessary pre-processing

```
import re
import nltk
import spacy
import string
from nltk.corpus import stopwords

nltk.download('stopwords')

nlp = spacy.load("en_core_web_sm")

def preprocess_text(text):
    # Tokenization and POS tagging using SpaCy
    doc = nlp(text)

    # Filtering out tokens based on POS tags and dependency parsing
    filtered_tokens = [token.text.lower() for token in doc if token.pos_ not in ["SPACE", "X"] and token.dep_ not in ["det", "punct"]]

    # Stopword removal
    filtered_tokens = [token for token in filtered_tokens if token not in stopwords.words('english')]

    # Lemmatization
    lemmatized_tokens = [token.lemma_ for token in nlp(" ".join(filtered_tokens))]

    return " ".join(lemmatized_tokens)
```

Step 4: we split the documents into chunks

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
# Split the preprocessed documents
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100,
    length_function=len,
    add_start_index=True,
)
chunks = text_splitter.split_documents(documents)
print(f"Split {len(documents)} documents into {len(chunks)} chunks.")
```

```
Split 4762 documents into 38997 chunks.
```

Step 5: we save the chunks in vector database

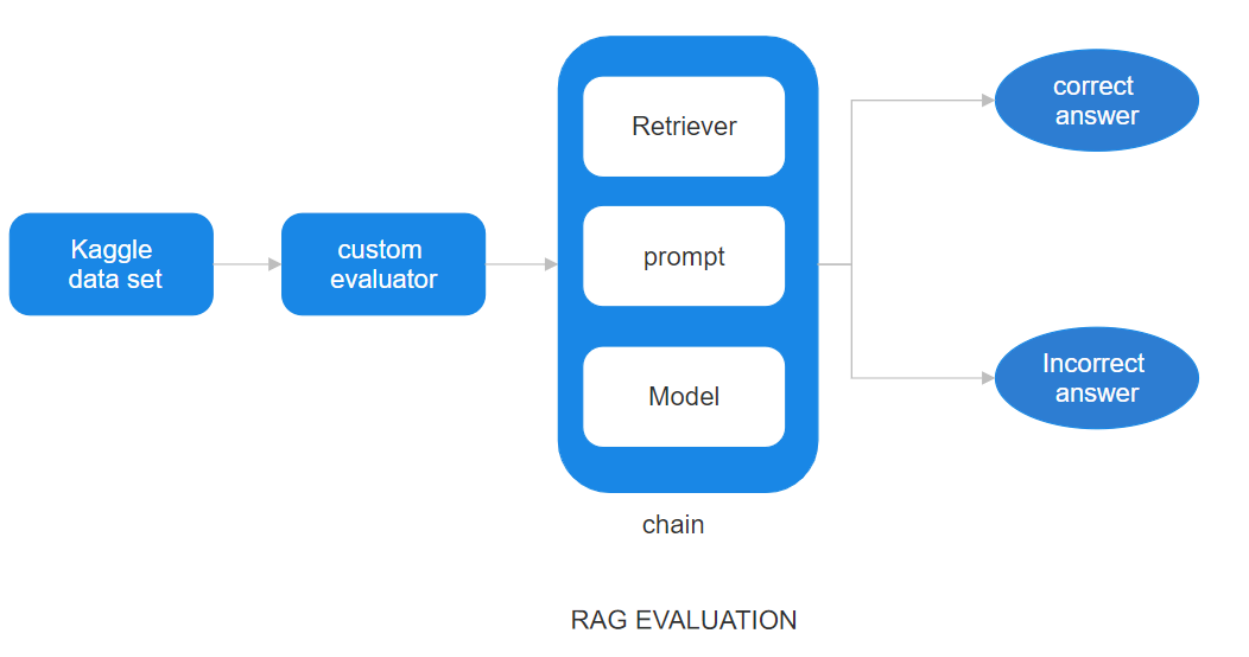
```python
# Save preprocessed chunks to Chroma

import os
import getpass
os.environ['OPENAI_API_KEY'] = getpass.getpass('Enter your OpenAI API key:')

from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma
```

```python
# Load the document, split it into chunks, embed each chunk and load it into the vector store.
# db = Chroma.from_documents(chunks, OpenAIEmbeddings(), persist_directory="./drive/MyDrive/Colab Notebooks/nlp/chroma_db-v(Harrison-mehta)")
db = Chroma(embedding_function=OpenAIEmbeddings(), persist_directory="./drive/MyDrive/Colab Notebooks/nlp/chroma_db-v(Harrison-mehta)")
```

Step 6: Run the retrieval loop

```python
context_text = ""
while True:
    query_text = input("Enter your query (type 'quit' to exit): ")

    if query_text.lower() == 'quit':
        break

    # Search the DB.
    results = db.similarity_search_with_relevance_scores(query_text, k=7)

    # Reorder documents
    reordering = LongContextReorder()
    reorder_docs = reordering.transform_documents(results)

    if len(reorder_docs) == 0 or reorder_docs[0][1] < 0.7:
        print(f"Unable to find matching results.")
        continue

    new_context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in reorder_docs])
    context_text += "\n\n---\n\n" + new_context_text
    prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
    prompt = prompt_template.format(context=context_text, question=query_text)
    # print(prompt)

    model = ChatOpenAI()
    response_text = model.invoke(prompt)

    # Load the model
    sources = [doc.metadata.get("source", None) for doc, _score in reorder_docs]
    formatted_response = f"Response: {response_text}\nSources: {sources}"
    print(formatted_response)
```

Evaluation



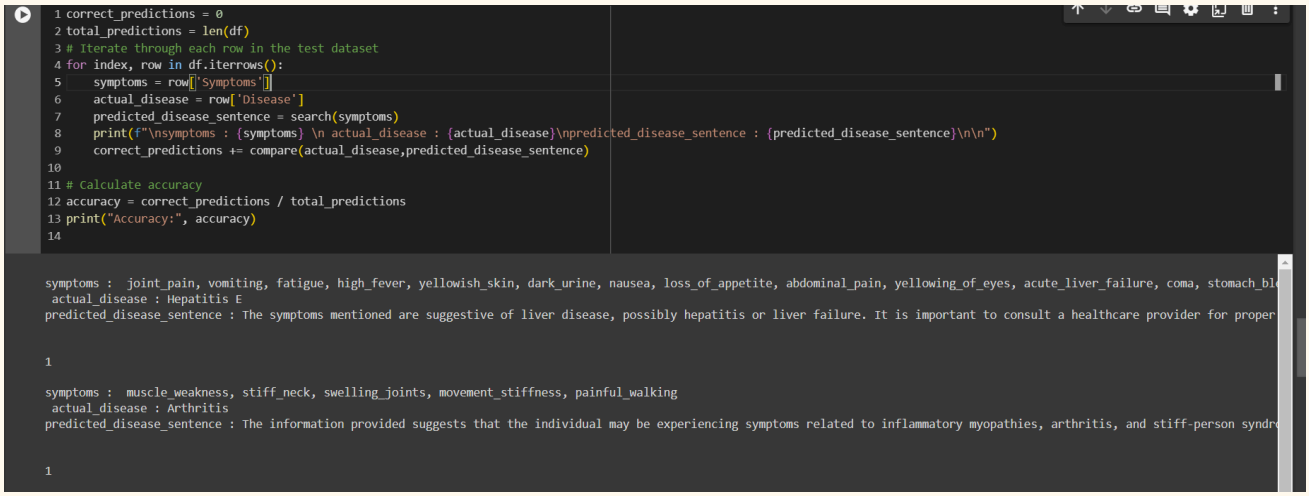
In the context of our evaluation process, our pipeline mandates the inclusion of several key components:

1. A Comprehensive Evaluation Dataset:
   Our evaluation framework relies on a robust dataset containing question-answer pairs (QA pairs). This dataset serves as the foundation for assessing the performance of our system.
2. Evaluator Tool:
   An evaluator tool has been integrated to meticulously gauge the precision of our system's performance, leveraging the extensive QA dataset as a benchmark.

Notably, the incorporation of Large Language Model (LLM) technology plays a pivotal role throughout our evaluation journey. Initially, an LLM is tasked with rows and columns provided by the data set, ensuring its thoroughness and relevance. Subsequently, additional LLMs are deployed to filter questions from this dataset, thereby ensuring their appropriateness for evaluation purposes. Finally, an LLM assumes the role of a judging agent, orchestrating the evaluation process on the synthetic dataset and offering invaluable insights into the efficacy of our system.

Moreover, our project's validation process is underpinned by a comprehensive evaluation dataset comprising QA pairs. We have leveraged a Kaggle dataset for this purpose, where our

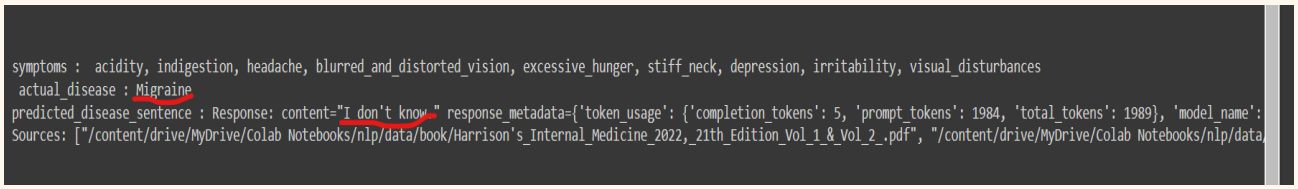
observations indicate a notable correlation between the answers generated by our Retrieval-Augmented Generation (RAG) system and the content of the Kaggle dataset. Our analysis reveals that over 60% of the time, the disease mentioned in the Kaggle dataset corresponds closely to the RAG-generated answer.



```
correct_predictions = 0
total_predictions = len(df)
# Iterate through each row in the test dataset
for index, row in df.iterrows():
    symptoms = row['Symptoms']
    actual_disease = row['Disease']
    predicted_disease_sentence = search(symptoms)
    print(f"\nsymptoms : {symptoms} \n actual_disease : {actual_disease}\npredicted_disease_sentence : {predicted_disease_sentence}\n\n")
    correct_predictions += compare(actual_disease,predicted_disease_sentence)

# Calculate accuracy
accuracy = correct_predictions / total_predictions
print("Accuracy:", accuracy)
```

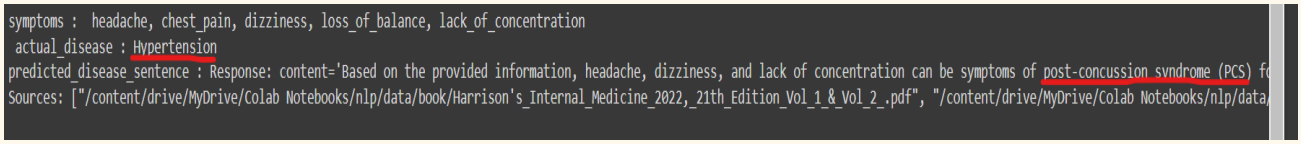
```
symptoms :  joint_pain, vomiting, fatigue, high_fever, yellowish_skin, dark_urine, nausea, loss_of_appetite, abdominal_pain, yellowing_of_eyes, acute_liver_failure, coma, stomach_bl
 actual_disease : Hepatitis E
predicted_disease_sentence : The symptoms mentioned are suggestive of liver disease, possibly hepatitis or liver failure. It is important to consult a healthcare provider for proper


1

symptoms :  muscle_weakness, stiff_neck, swelling_joints, movement_stiffness, painful_walking
 actual_disease : Arthritis
predicted_disease_sentence : The information provided suggests that the individual may be experiencing symptoms related to inflammatory myopathies, arthritis, and stiff-person syndr


1
```

Additionally, instances where the RAG system responds with "I don't know". I think it is a good thing which means machine is trying to give ans only on the basis of facts.



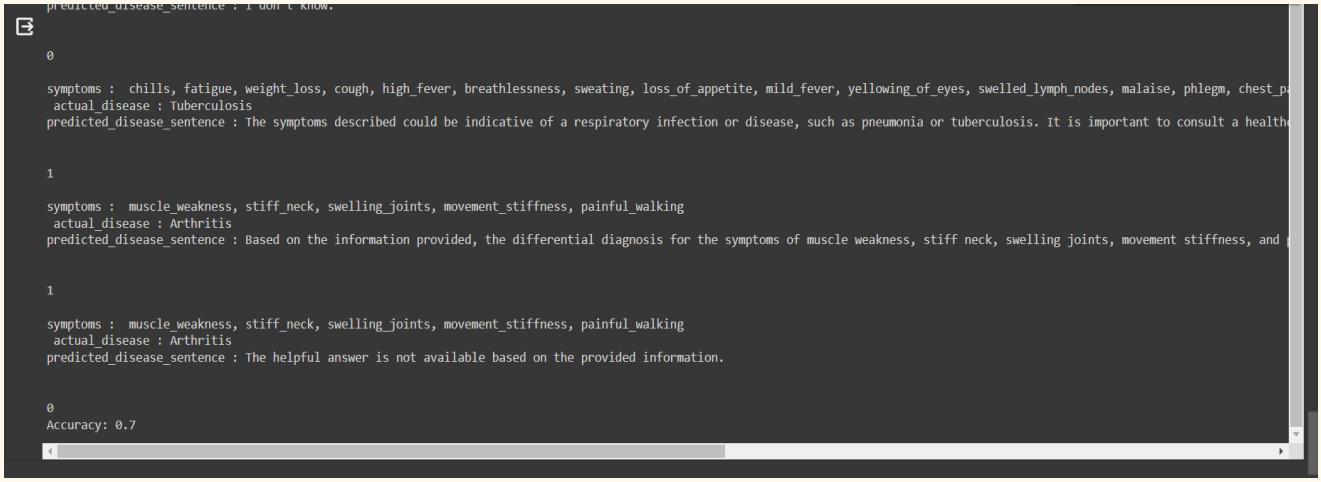
```
symptoms :  acidity, indigestion, headache, blurred_and_distorted_vision, excessive_hunger, stiff_neck, depression, irritability, visual_disturbances
 actual_disease : Migraine
predicted_disease_sentence : Response: content="I don't know." response_metadata={'token_usage': {'completion_tokens': 5, 'prompt_tokens': 1984, 'total_tokens': 1989}, 'model_name':
Sources: ["/content/drive/MyDrive/Colab Notebooks/nlp/data/book/Harrison's_Internal_Medicine_2022,_21th_Edition_Vol_1_&_Vol_2_.pdf", "/content/drive/MyDrive/Colab Notebooks/nlp/data/
```

However, it is imperative to acknowledge that discrepancies exist, with occasional instances of incorrect answers generated by the system.

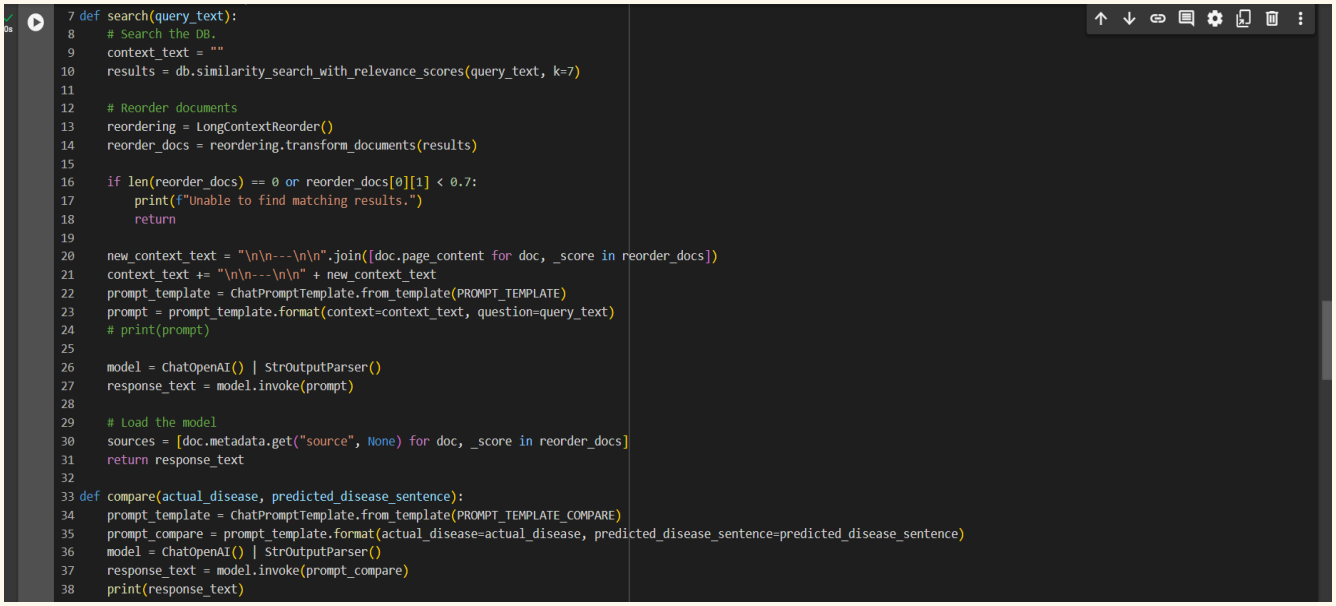


```
symptoms :  headache, chest_pain, dizziness, loss_of_balance, lack_of_concentration
 actual_disease : Hypertension
predicted_disease_sentence : Response: content='Based on the provided information, headache, dizziness, and lack of concentration can be symptoms of post-concussion syndrome (PCS) fo
Sources: ["/content/drive/MyDrive/Colab Notebooks/nlp/data/book/Harrison's_Internal_Medicine_2022,_21th_Edition_Vol_1_&_Vol_2_.pdf", "/content/drive/MyDrive/Colab Notebooks/nlp/data/
```

We've realized that getting wrong answers is a big problem for our project. That's why we believe that in the future, only doctors should use it at first. Our data is limited, and sometimes the model can't find the right information. We think that doctors should be the ones to decide if the answer the system gives is correct. If it's wrong, the model should note down all the symptoms and diseases related to the wrong answer. Then, we can add this information to our file of doctor suggestions.

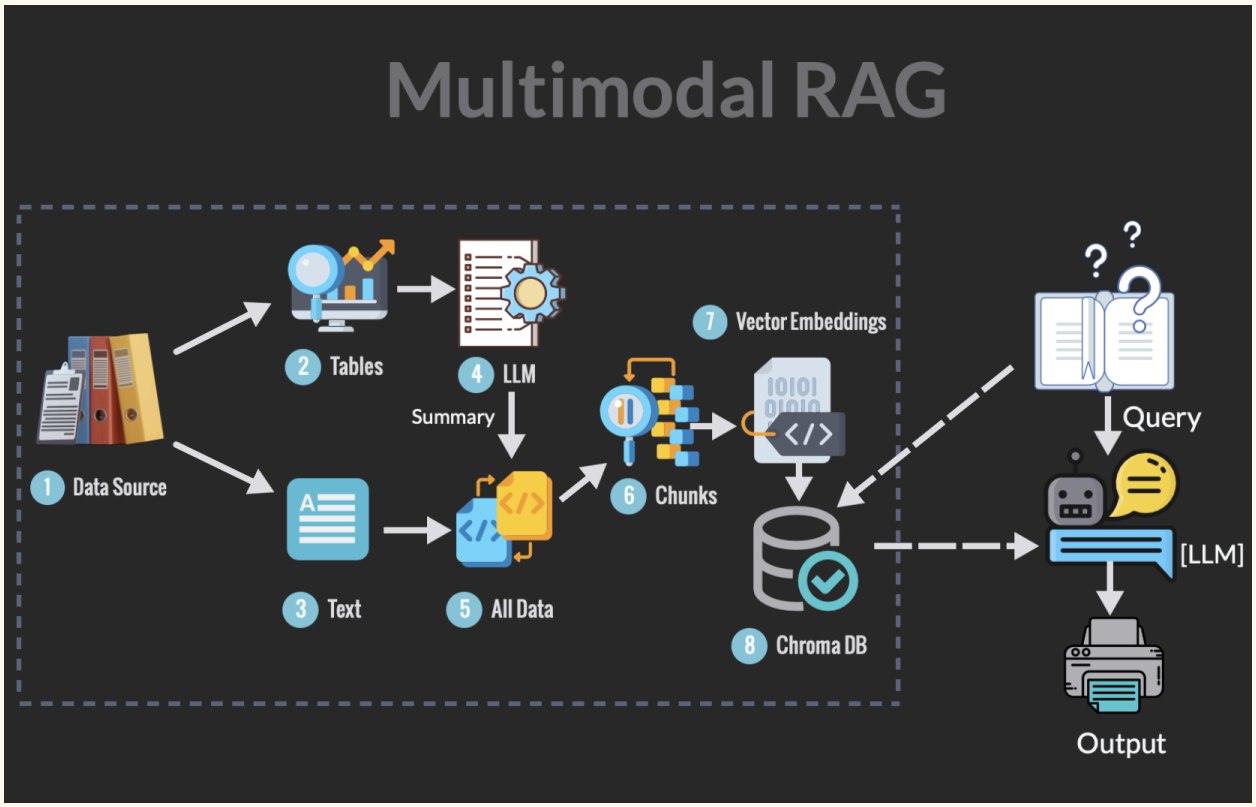


Our model's performance demonstrates variability due to OpenAI retrieval. At times, we achieve a bit lower accuracy, while in other instances, our accuracy reaches 0.7.



```python
def search(query_text):
    # Search the DB.
    context_text = ""
    results = db.similarity_search_with_relevance_scores(query_text, k=7)

    # Reorder documents
    reordering = LongContextReorder()
    reorder_docs = reordering.transform_documents(results)

    if len(reorder_docs) == 0 or reorder_docs[0][1] < 0.7:
        print(f"Unable to find matching results.")
        return

    new_context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in reorder_docs])
    context_text += "\n\n---\n\n" + new_context_text
    prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
    prompt = prompt_template.format(context=context_text, question=query_text)
    # print(prompt)

    model = ChatOpenAI() | StrOutputParser()
    response_text = model.invoke(prompt)

    # Load the model
    sources = [doc.metadata.get("source", None) for doc, _score in reorder_docs]
    return response_text

def compare(actual_disease, predicted_disease_sentence):
    prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE_COMPARE)
    prompt_compare = prompt_template.format(actual_disease=actual_disease, predicted_disease_sentence=predicted_disease_sentence)
    model = ChatOpenAI() | StrOutputParser()
    response_text = model.invoke(prompt_compare)
    print(response_text)
```

Multimodal Embeddings



As the dataset contains data in tabular form as well apart from just text. We have implemented the multimodal approach where tables would be extracted out from data & stored in different variable.

Let's understand step wise:

1. Data Source: Contains all the medical textbooks.
2. Tables: The partition function will parse the document & store the tables separately.
3. Text: Partitioning would also result in yielding the text component.
4. LLM: The table data is given to a LLM model to summarize.
5. All Data: It combines the text component with the summary generated by LLM.
6. Chunks: Data is splitted into multiple small size chunks.

7. Vector embeddings: Chunks are converted into vector embeddings using OpenAi embedding functions.
8. Chroma DB: All the vector embeddings are stored into the chroma database.

Now what will happen is when User fires a query then based on the similarity search performed on vector embeddings, top 7 context chunks would be extracted & would be embedded in Prompt Template.

Prompt would be invoked into the LLM model which would then provide a response text based on the context provided.

Code:

Partitioning PDF

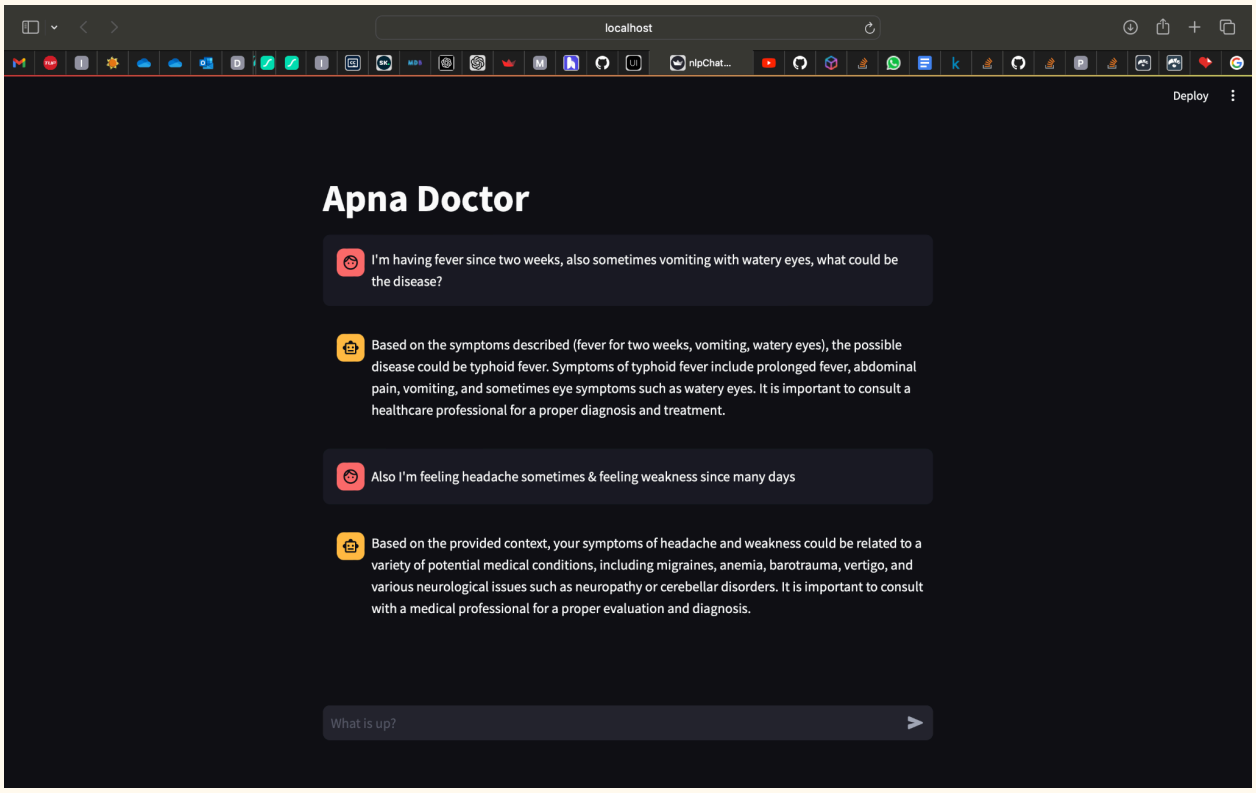
```
def partition_doc():

    raw_pdf_elements = partition_pdf(

        filename= DATA_PATH + "/docmerged.pdf",

        extract_images_in_pdf=False,

        infer_table_structure=True,

        chunking_strategy="by_title",

        max_characters=6000,

        new_after_n_chars=3800,

        combine_text_under_n_chars=2000,

    )



    # Categorize by type

    tables = []

    texts = []
```

```
    for element in raw_pdf_elements:
        if "unstructured.documents.elements.Table" in str(type(element)):
            tables.append(str(element))
                elif "unstructured.documents.elements.CompositeElement" in
str(type(element)):
            texts.append(str(element))

    print(len(tables), " ", len(texts))
    # Prompt
     prompt_text = """You are an assistant tasked with summarizing tables and
text. \
    Give a concise summary of the table or text. Table or text chunk: {element}
"""
    prompt = ChatPromptTemplate.from_template(prompt_text)

    # Summary chain
    model = ChatOpenAI(temperature=0, model="gpt-3.5-turbo")
        summarize_chain = {"element": lambda x: x} | prompt | model |
StrOutputParser()

    table_summaries = summarize_chain.batch(tables, {"max_concurrency": 5})

    return table_summaries, texts
```

Unstructured library provides method to break down the source file into multiple elements through which we can extract tables & just text separately.

Once extracted then we're forwarding the table data to LLM for summarization so we can get meaningful insights from numericals.

UI Design



Code:

```
st.title("Apna Doctor")


CHROMA_PATH = "chromaV1"
```

```
PROMPT_TEMPLATE = """

Answer the question based only on the following context:



{context}



---



Answer the question based on the above context: {question}

"""



embedding_function = OpenAIEmbeddings()

db                    =                    Chroma(persist_directory=CHROMA_PATH,
embedding_function=embedding_function)

context_text = ""



if "openai_model" not in st.session_state:

   st.session_state["openai_model"] = "gpt-3.5-turbo"



if "messages" not in st.session_state:

   st.session_state.messages = []



for message in st.session_state.messages:

   with st.chat_message(message["role"]):

       st.markdown(message["content"])
```

```
if prompt := st.chat_input("What is up?"):
   st.session_state.messages.append({"role": "user", "content": prompt})
   with st.chat_message("user"):
       st.markdown(prompt)


   # Perform similarity search
   results = db.similarity_search_with_relevance_scores(prompt, k=3)
   if len(results) == 0 or results[0][1] < 0.5:
       st.write("Unable to find matching results.")
   else:
        new_context_text = "\n\n---\n\n".join([doc.page_content for doc, _score
in results])
       context_text = "\n\n---\n\n".join([context_text, new_context_text])
       prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
       promptt = prompt_template.format(context=context_text, question=prompt)


       # Generate response using OpenAI
       with st.chat_message("assistant"):
           with st.spinner("Generating response..."):
               messages = [
                   {"role": m["role"], "content": promptt}
                   for m in st.session_state.messages
               ]
               model = ChatOpenAI() | StrOutputParser()
               response_text = model.invoke(promptt)
```

```
                text = f"{response_text}"
                st.write(text)



                # Load the model
                  sources = [doc.metadata.get("source", None) for doc, _score in
results]
                formatted_response = f"{response_text}"
                # st.write(response_text)
                st.session_state.messages.append({"role": "assistant", "content":
formatted_response})
```

We're using ^Streamlit for publishing it to the web. Session state is used to store & retrieve messages, making it a chat interface.

^ Streamlit is an open-source Python library used for creating web applications with minimal effort. It allows developers to build interactive web applications directly from Python scripts, without needing to write HTML, CSS, or JavaScript code.

Reference book for creating vector database

1. PJ Mehta’s Practical Medicine
2. Harrison's Principles of Internal Medicine, 21e

Future Plans for this project

This project has been an enriching experience, during which we've learned a lot of new things. As we conclude Mandate 4, we're submitting this project for assessment. However, there's still ample room for improvement. Primarily, we need to enhance the dataset, and secondly, we should refine our evaluation methods.

Github link: click here
NoteBook link: click here

REFERENCES

1. DAY-13 | End to End Medical Chatbot Project | Part -2
2. LangChain official website
3. Build a Streamlit Chatbot FAST
4. How to evaluate an LLM-powered RAG application automatically.
5. RAG + Langchain Python Project: Easy AI/Chat For Your Docs