PROJECT REPORT: HELPMATE AI PROJECT

Problem Statement:

We are building a project in the insurance domain aimed at creating a robust generative search system capable of accurately answering questions from various insurance policy documents. LangChain provides a comprehensive framework to integrate, process, and retrieve domain-specific data efficiently using large language models (LLMs). It streamlines the construction of pipelines that connect document retrieval with generative AI tasks.

Why LangChain:

LangChain was chosen for its ability to handle diverse data sources like PDFs and directories while offering seamless integration with tools like Chroma for vector storage and retrieval. Its modularity allows customization of retrieval workflows and prompts, enabling us to use GROQ AI as the generative model. LangChain's flexibility ensures a scalable, LLM-agnostic solution for developing efficient RAG systems.

Project Goals:

- Develop an intelligent generative search system to answer user queries from insurance policy documents.
- Incorporate semantic search to ensure relevant document sections are retrieved efficiently.
- Generate concise, accurate, and context-aware responses using GROQ AI, an open-source generative AI alternative.
- Maintain a cost-effective, scalable solution by leveraging open-source tools and frameworks.

Data Sources:

- **Document Data:** A collection of 217 publicly available insurance policy documents in PDF format.
  - Example files: HDFC Life Sanchay Plus Policy Document, HDFC Life Sampoorna Jeevan Policy Document.
- **Directory Structure:**
  - Documents are stored in /content/drive/MyDrive/Policy-Documents/.
  - Extracted metadata includes document source paths and page numbers.

Tools and Frameworks Used:

- **LangChain:** Simplifies integration with LLMs and enables the construction of end-to-end pipelines.
- **GROQ AI:** Open-source and cost-free alternative to OpenAI for high-performance generative AI tasks.
- **SBERT (Sentence-BERT):** Embedding model for semantic text representation.
- **Chroma:** A vector database for storing and querying text embeddings.
- **Text Splitters:** To preprocess large documents into manageable chunks for efficient retrieval.

Solution Design Overview

1. Indexing Process (Offline):

- Objective: Prepare documents for fast semantic search during runtime.
- Steps:
  1. Load Documents: Use DirectoryLoader to load all PDF files.
  2. Split Documents: Break large documents into smaller, overlapping chunks using RecursiveCharacterTextSplitter.
  3. Generate Embeddings: Use SBERT to convert text into dense vector embeddings that capture semantic meaning.
  4. Store in Vector Database: Save embeddings in a Chroma vector store to enable fast similarity-based search.

2. Retrieval and Generation (Runtime):

- Objective: Provide accurate answers to user queries.
- Steps:
  1. User Query Input: Accept a user query as input.
  2. Retrieve Relevant Documents: Perform similarity search on the vector store to fetch the top-k matching document chunks.
  3. Generate Answer: Use GROQ AI to generate a response by combining retrieved documents with the user query.
  4. Return Answer: Display a concise, context-aware answer to the user.

Implementation Details:

Step 1: Install Dependencies

Install required Python libraries:

```
!pip install --quiet --upgrade langchain langchain-community langchain-chroma langchain-groq pypdf sentence-transformers
```

Step 2: Load and Preprocess Documents

- Load documents using:

```
from langchain_community.document_loaders import PyPDFLoader
from langchain.document_loaders import DirectoryLoader

file_path = "/content/drive/MyDrive/Policy-Documents"
loader = DirectoryLoader(file_path, glob="**/*.pdf", loader_cls=PyPDFLoader)
# Use DirectoryLoader to load all PDF files in the directory

docs = loader.load()

print(f"Loaded {len(docs)} documents")
```

```
Loaded 217 documents
```

- Split documents for embedding generation:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200, add_start_index=True
)
all_splits = text_splitter.split_documents(docs)

len(all_splits)
```

```
760
```

Step 3: Generate Embeddings

Convert document chunks into embeddings:

```
from langchain_chroma import Chroma
from langchain.embeddings import HuggingFaceEmbeddings  # For SBERT

# Use HuggingFaceEmbeddings with a Sentence-BERT model
embedding_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-mpnet-base-v2")

# Initialize Chroma with the updated embedding model
vectorstore = Chroma.from_documents(documents=all_splits, embedding=embedding_model)
```

Step 4: Perform Document Retrieval

- Use similarity search to retrieve top-k documents:

```
retriever = vectorstore.as_retriever(search_type="similarity", search_kwargs={"k": 6})

retrieved_docs = retriever.invoke("What are the available benefit options in the HDFC Life Sanchay Plus plan?")

len(retrieved_docs)
```

```
6
```

Step 5: Generate Responses

Generate answers using GROQ AI:

```
from langchain import hub

prompt = hub.pull("rlm/rag-prompt")

example_messages = prompt.invoke(
    {"context": "filler context", "question": "filler question"}
).to_messages()

example_messages
```

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough


def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)


rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

for chunk in rag_chain.stream("What are the eligibility criteria for members under the group insurance plans?"):
    print(chunk, end="", flush=True)
```

Challenges Faced:

- **API Key Configuration:** Properly setting up GROQ and LangChain API keys to avoid errors during LLM initialization.
- **Document Variability:** Handling inconsistent formats, page layouts, and missing metadata in policy documents.
- **Chunk Optimization:** Fine-tuning chunk_size and chunk_overlap for maximum embedding quality.

Lessons Learned:

- The significance of reliable data extraction processes.
- The critical role of thorough data preprocessing.
- The need to optimize retrieval mechanisms for efficiency.
- Addressing input length limitations effectively.
- Recognizing that evaluation involves multiple dimensions.
- Safeguarding and managing sensitive data appropriately.