Smart Policy Assistant: Implementing a Retrieval-Augmented LLM for Life Insurance

Background

In the insurance industry, life insurance policy documents are often lengthy and complex involving legal terminology. This makes it difficult for policyholders to quickly find answers to specific questions such as eligibility, exclusions, claim procedures, or grace periods. Traditional search systems struggle to interpret natural language queries.

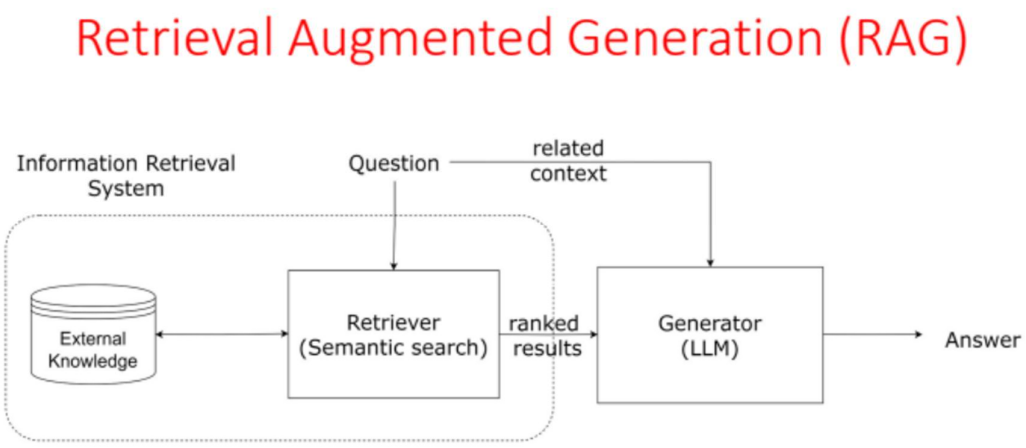
Mr.HelpMate AI project aims to implement such a system for life insurance documents. By leveraging embedding models, vector databases like ChromaDB, cross-encoder reranking, and OpenAI's GPT models, the system delivers accurate, context-aware answers to user queries. Additionally, it implements a caching mechanism to optimize repeated query performance, improving both response speed and cost efficiency.

Objectives

1. **Design a Retrieval-Augmented Generation (RAG) System**
   Build a modular, multi-layer architecture (Embedding → Search → Generation) capable of answering user questions from a life insurance policy PDF.
2. **Implement Effective Embedding**
   Experiment with different embedding models (OpenAI & SentenceTransformers) to generate high-quality vector representations of the policy document.
3. **Build and Query a Vector Search Index Using ChromaDB**
   Store chunk embeddings in ChromaDB and retrieve relevant chunks using semantic similarity for user queries.
4. **Implement a Cache-First Query Resolution Strategy**
   Introduce a dual-layer search approach with a fast cache collection and a fallback full-data search, optimizing repeated query performance.
5. **Integrate a Cross-Encoder Re-ranker**
   Improve search result accuracy by re-ranking top-k chunks using a sentence-level cross-encoder model
6. **Generate Context-Aware Responses via LLMs**
   Construct dynamic prompts incorporating retrieved context chunks and use OpenAI GPT-3.5 to generate answers.
7. **Evaluate System Performance Across Sample Queries**
   Test the pipeline against at least 3 user-designed queries and measure relevance, accuracy, and completeness of generated answers.
8. **Reusability in Code Design**
   Keep future extension and code reusability while writing code.

Design

RAG System working(High Level)



Embedding Layer

- **PDF Extraction**: Text is extracted from the life insurance policy PDF using pdfplumber.
- **Embedding**: OpenAI's 'text-embedding-ada-002' model is used to generate embeddings for the extracted text and stored them in a ChromaDB collection.
- **Storage**: The embeddings are stored in ChromaDB within a primary vector collection (RAG_on_InsurancePolicy). A separate ChromaDB collection was created to serve as a cache.

2. Search Layer

- **Query Handling**: A user question is embedded and used to search relevant chunks.
- **Cache**: The query is first checked against a cache collection.
- **Fallback Search**: If no results are found in the cache, the main collection is queried.
- **Cache Update**: Any new results from the main collection are stored in the cache for future reuse.
- **Re-ranking**: Top retrieved chunks are re-ranked using a cross-encoder model (cross-encoder/ms-marco-MiniLM-L-6-v2).

3. Generation Layer

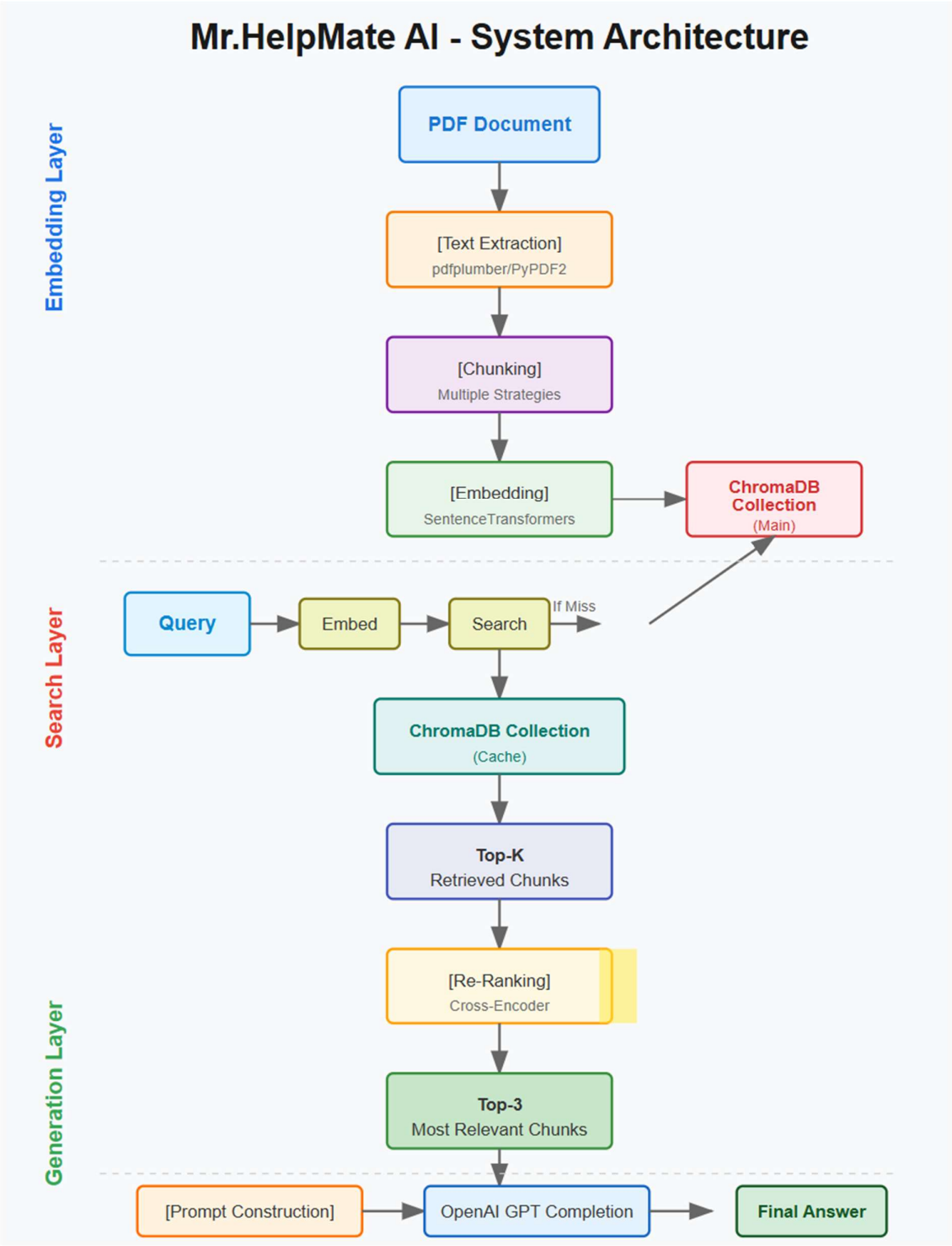
- **Prompt Construction**: The top reranked chunks and query are inserted into a structured prompt.
- **Response Generation**: OpenAI GPT-3.5 is used to generate answer.

4. Evaluation

- Evaluation via executing the three queries end-to-end, capturing and get 3 screenshots of the top-3 chunks retrieved (Search Layer) and 3 screenshots of the final generated answers (Generation Layer)

5. Analyze system performance and document insights

Flow diagram



Implementation

Retrieval-Augmented Generation (RAG) system is designed to answer questions based on a life insurance policy PDF and answer user queries from complex policy documents. Here's a summary:

1. **Data Ingestion and Processing**: The policy PDF is loaded, text and tables are extracted, and the data is organized into a pandas DataFrame with associated metadata (page number and document name).
2. **Embedding**: The processed text from the PDF is converted into numerical representations (embeddings) using OpenAI's text-embedding-ada-002 model.
3. **Vector Store (ChromaDB)**: These embeddings are stored in a ChromaDB collection (insurancedata_collection), which acts as your main knowledge base for semantic search.
4. **Caching (ChromaDB)**: A separate ChromaDB collection (cache_collection) is used to store previous queries and their results. This acts as a cache to speed up responses for repeated or similar queries.
5. **Retrieval with Cache**: When a user enters a query, the system first searches the cache_collection. If a sufficiently similar query is found (based on a distance threshold), the cached results are used.
6. **Retrieval from Main Store**: If the query is not found in the cache, the system queries the main insurancedata_collection to retrieve the most relevant document chunks based on semantic similarity.
7. **Cache Update**: The query and the retrieved results from the main store are then added to the cache_collection for future use.
8. **Re-ranking**: The retrieved document chunks from either the cache or the main store are then re-ranked using a cross-encoder model (cross-encoder/ms-marco-MiniLM-L-6-v2). This step helps to improve the relevance of the top results by considering the query and each retrieved document pair together.
9. **Generation**: The top re-ranked document chunks are provided as context to a language model (like GPT-3.5-turbo) using a carefully crafted prompt. The language model then generates a natural language answer to the user's query based *only* on the provided context.

Summary

The system retrieves relevant information from your document corpus, potentially leveraging a cache for speed, re-ranks the information for better accuracy, and then uses a language model to synthesize a concise and informative answer based on that information.

Challenges Faced

1. Handling Long Documents
2. Choosing the Right Chunking and Embedding Strategy
3. Semantic Search Precision

4. Cross-Encoder Model Overhead
5. Prompt Sensitivity
6. Caching Strategy

Lessons Learned

1. RAG Systems Are Modular but Sensitive - changes in chunking or prompts can significantly affect output quality
2. Search and Reranking needs to be effective.
3. Prompt Engineering Is Crucial
4. Caching Matters for Efficiency - to reduce latency and cost for repeated queries.
5. Structured Design Enables Debuggability and code reusability.

Future improvements

- The above prompt is a very simplistic one. Improvement can be done by building a more robust prompt template that can give you the search results, along with citations (Page No, metadata, etc).
- Modularize the different parts from semantic search section.