Semantic Spotter Project Submission

1. Background

This project demonstrate "Build a RAG System" in insurance domain using LangChain.

2. Problem Statement

The goal of the project is to build a robust generative search system capable of effectively and accurately answering questions from a bunch of policy documents.

3. Document

1. The policy documents can be found here

4. Approach

LangChain is a framework that simplifies the development of LLM applications LangChain offers a suite of tools, components, and interfaces that simplify the construction of LLM-centric applications. LangChain enables developers to build applications that can generate creative and contextually relevant content LangChain provides an LLM class designed for interfacing with various language model providers, such as OpenAI, Cohere, and Hugging Face.

LangChain's versatility and flexibility enable seamless integration with various data sources, making it a comprehensive solution for creating advanced language model-powered applications.

LangChain's open-source framework is available to build applications in Python or JavaScript/TypeScript. Its core design principle is composition and modularity. By combining modules and components, one can quickly build complex LLM-based applications. LangChain is an open-source framework that makes it easier to build powerful and personalizable applications with LLMs relevant to user's interests and needs. It connects to external systems to access information required to solve complex problems. It provides abstractions for most of the functionalities needed for building an LLM application and also has integrations that can readily read and write data, reducing the development speed of the application. LangChains's framework allows for building applications that are agnostic to the underlying language model. With its ever expanding support for various LLMs, LangChain offers a unique value proposition to build applications and iterate continuously.

LangChain framework consists of the following:

- Components: LangChain provides modular abstractions for the components necessary to work with language models. LangChain also has collections of implementations for all these abstractions. The components are designed to be easy to use, regardless of whether you are using the rest of the LangChain framework or not.
- Use-Case Specific Chains: Chains can be thought of as assembling these components in particular ways in order to best accomplish a particular use case. These are intended to be a higher level interface through which people can easily get started with a specific use case. These chains are also designed to be customizable.

The LangChain framework revolves around the following building blocks:

- Model I/O: Interface with language models (LLMs & Chat Models, Prompts, Output Parsers)
- Retrieval: Interface with application-specific data (Document loaders, Document transformers, Text embedding models, Vector stores, Retrievers)
- Chains: Construct sequences/chains of LLM calls
- Memory: Persist application state between runs of a chain
- Agents: Let chains choose which tools to use given high-level directives
- Callbacks: Log and stream intermediate steps of any chain

5. System Layers

- Reading & Processing PDF Files: We will be using LangChain PyPDFDirectoryLoader to read and process the PDF files from specified directory.

- Document Chunking: We will be using LangChain RecursiveCharacterTextSplitter. This text splitter is the recommended one for generic text. It is parameterized by a list of characters. It tries to split on them in order until the chunks are small enough. The default list is ["\n\n", "\n", " ", ""].

This has the effect of trying to keep all paragraphs (and then sentences, and then words) together as long as possible, as those would generically seem to be the strongest semantically related pieces of text..

- Generating Embeddings: We will be using OpenAIEmbeddings from LangChain package. The Embeddings class is a class designed for interfacing with text embedding models. LangChain provides support for most of the embedding model providers (OpenAI, Cohere) including sentence transformers library from Hugging Face. Embeddings create a vector representation of a piece of text and supports all the operations such as similarity search, text comparison, sentiment analysis etc. The base Embeddings class in LangChain provides two methods: one for embedding documents and one for embedding a query.

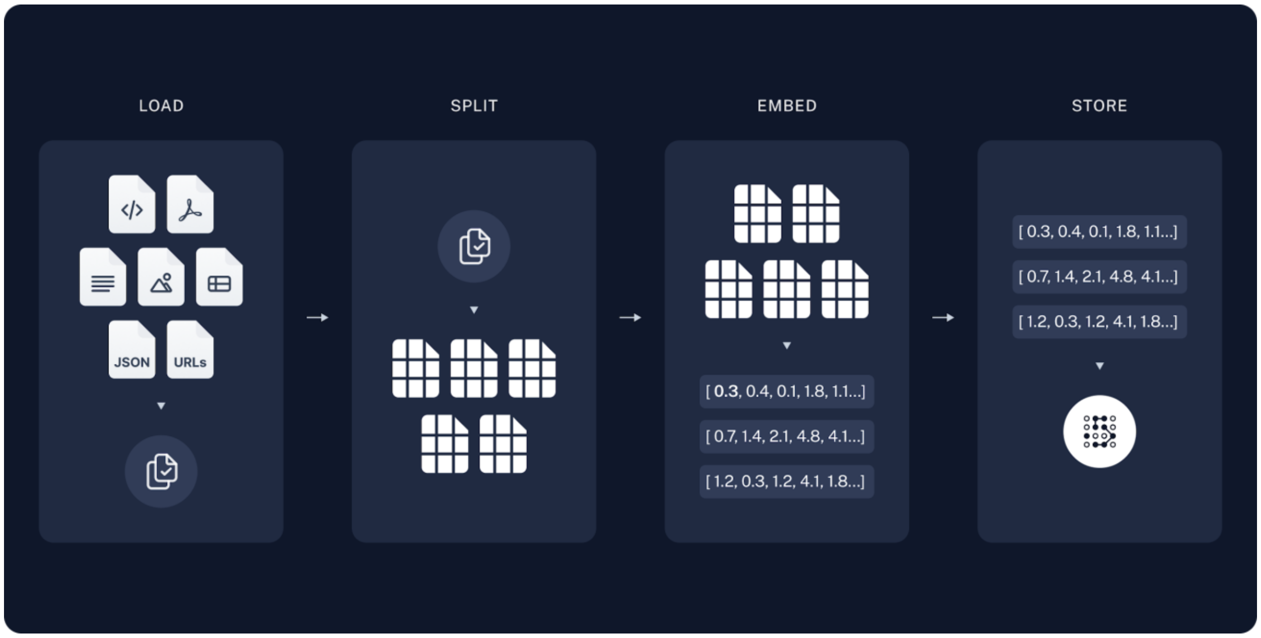
- Store Embeddings In ChromaDB: In this section we will store embedding in ChromaDB. This embedding is backed by LangChain CacheBackedEmbeddings

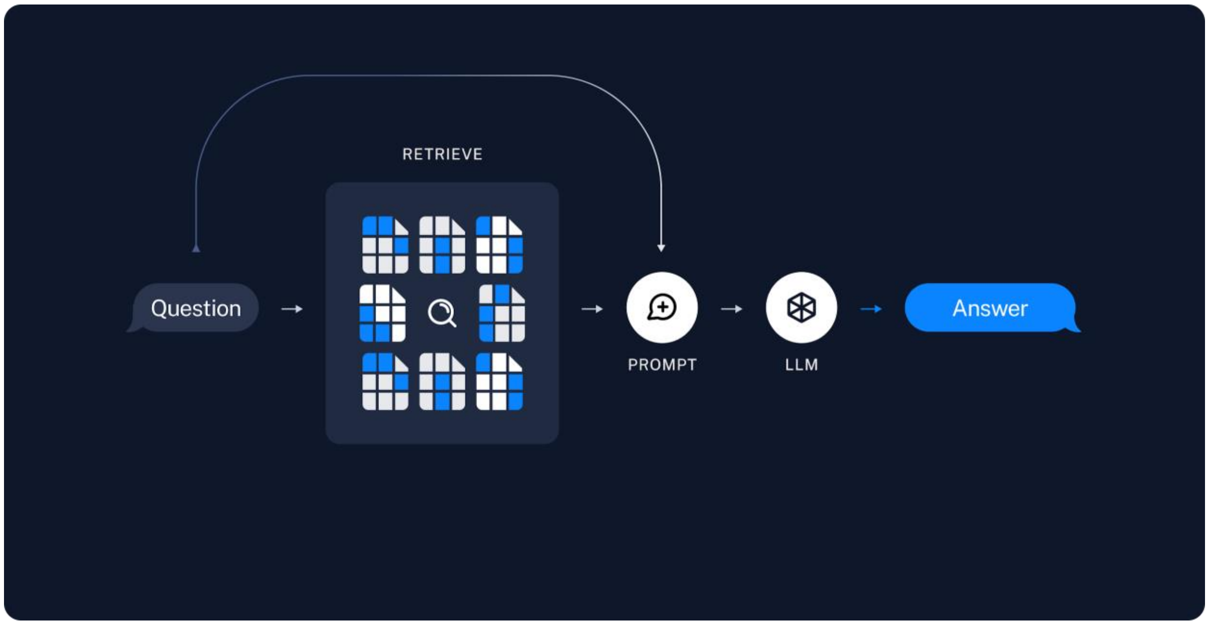
- Retrievers: Retrievers provide Easy way to combine documents with language models.A retriever is an interface that returns documents given an unstructured query. It is more general than a vector store. A retriever does not need to be able to store documents, only to return (or retrieve) them. Retriever stores data for it to be queried by a language model. It provides an interface that will return documents based on an unstructured query. Vector stores can be used as the backbone of a retriever, but there are other types of retrievers as well. There are many different types of retrievers, the most widely supported is the VectoreStoreRetriever.

- Re-Ranking with a Cross Encoder: Re-ranking the results obtained from the semantic search will sometime significantly improve the relevance of the retrieved results. This is often done by passing the query paired with each of the retrieved responses into a cross-encoder to score the relevance of the response w.r.t. the query. The above retriever is associated with HuggingFaceCrossEncoder with model BAAI/bge-reranker-base

- Chains: LangChain provides Chains that can be used to combine multiple components together to create a single, coherent application. For example, we can create a chain that takes user input, formats it with a PromptTemplate, and then passes the formatted response to an LLM. We can build more complex chains by combining multiple chains together, or by combining chains with other components. We are using pulling prompt rlm/rag-promp from langchain hub to use in the RAG chain.

6. System Architecture





7. Challenges Faced

1. Parsing and Chunking Policy PDFs Effectively

- Insurance documents are long and complex.
- Maintaining **semantic coherence** during chunking was difficult.
- Required smart chunking using `RecursiveCharacterTextSplitter` to preserve sentence/paragraph structure for meaningful embeddings.

2. Choosing the Right Retrieval Strategy

- A standard vector store retriever might return text that is **technically similar** but **contextually irrelevant**.
- Had to combine ChromaDB with LangChain retrievers and ranking logic to improve overall accuracy.

8. Lessons Learned

1. RAG System Performance Improves with Re-ranking

- Using a cross encoder significantly boosted the precision of retrieved answers.
- Semantic similarity ≠ usefulness — re-ranking helped prioritize **answer relevance**.

2. LangChain is Powerful but Demands Structure

- Its flexibility comes from well-separated modules (Model I/O, Chains, Retrievers, etc.), but building a **clean, coherent architecture** is key.
- Use-case-specific chains help abstract and manage complexity.

3. Open Source Tools Like ChromaDB Can Scale Fast

- ChromaDB was effectively used for lightweight, local vector search without managing complex infrastructure.

9. Prerequisites

- Python 3.7+
- langchain 0.3.13
- Please ensure that you add your OpenAI API key to the empty text file named "OpenAI_API_Key.txt" in order to access the OpenAI API.

10. Running

- Clone the github repository

-
  ```
  $ git clone https://github.com/aniltiwari-tech/Semantic-Spotter-Project.git
  ```
- Open the notebook in jupyter and run all cells.