Semantic Spotter – Project Submission

1. Overview

This project demonstrates a **Retrieval-Augmented Generation (RAG)** system tailored for the **insurance domain**, built using the LangChain framework.

---

2. Problem Statement

The objective is to develop a **generative search system** that can accurately and efficiently answer questions using the contents of various policy documents.

---

3. Dataset

The insurance policy documents used in this project are available in the `DocumentsPolicy` directory.

---

4. Methodology

LangChain is a modular framework that simplifies the development of LLM-driven applications by offering reusable components, tools, and integrations.

Key Features:

- **Modular Components** – Prompt templates, chains, agents, memory, callbacks, etc.
- **Composability** – Build pipelines by combining reusable blocks.
- **LLM-Agnostic** – Supports various providers like OpenAI, Cohere, Hugging Face.
- **External Integration** – Easily connects to APIs, databases, and document sources.

---

5. System Components

Document Loading

- Uses `PyPDFDirectoryLoader` to load PDFs from the specified folder.

Text Splitting

- Splits documents into chunks using `RecursiveCharacterTextSplitter`, which intelligently segments text to preserve context.

Embedding Generation

- Converts text to vectors using `OpenAIEmbeddings`, enabling similarity search and semantic comparison.

Vector Storage

- Stores embeddings in **ChromaDB**, backed by `CacheBackedEmbeddings` for performance and caching.

Document Retrieval

- Uses `VectorStoreRetriever` to fetch relevant chunks based on similarity with the user query.

Re-Ranking with Cross Encoder

- Applies `HuggingFaceCrossEncoder` (`BAAI/bge-reranker-base`) to re-rank retrieved chunks based on contextual relevance.

Chains

- Uses `LLMChain` and LangChain Hub's `rlm/rag-prompt` to format queries, combine inputs, and generate final responses from the LLM.

---

6. System Architecture

The system follows a modular, four-stage Retrieval-Augmented Generation (RAG) pipeline:

1. **Load**
    - Source documents are ingested from formats like PDFs, JSON, plain text, images, URLs, and HTML.
    - LangChain's document loaders (e.g., `PyPDFDirectoryLoader`) are used to handle structured and unstructured content.
2. **Split**
    - Large documents are divided into smaller, semantically coherent chunks using tools like `RecursiveCharacterTextSplitter`.
    - This ensures that each chunk fits within token limits and retains meaningful context.
3. **Embed**
    - Each chunk is transformed into a dense vector using an embedding model such as `OpenAIEmbeddings`.
    - The embedding step converts natural language into a numeric form suitable for similarity comparison.
4. **Store**

- Vectors are stored in a vector database like ChromaDB for fast semantic retrieval.
- Optionally, `CacheBackedEmbeddings` can be used to avoid redundant embedding computations.

At query time, the same embedding model converts the user query into a vector. The system performs a similarity search to retrieve the most relevant document chunks, optionally reranks them using a cross-encoder, and finally passes them to the LLM to generate a context-aware response.

---

7. Requirements

- Python 3.7 or higher
- `langchain==0.3.13`
- OpenAI API Key (store it in a file named `OpenAI_API_Key.txt`)

---

8. Getting Started

Clone the Repository

```
git clone https://github.com/jafarijason/semantic-spotter-project.git
cd semantic-spotter-project
```

Open the Notebook

Launch and run all cells in the following notebook: semantic-spotter-langchain-notebook.ipynb

Manual Setup Instructions

```
virtualenv venv
source venv/bin/activate
pip install -r requirements.txt
```

```
jupyter notebook \
    --notebook-dir="." \
    --ip=0.0.0.0 --port=3225
```

Contributors

Jason Jafari