Semantic Spotter
Generative Search System for Policy Documents

Introduction

This report outlines the development of Semantic Spotter, a robust generative search system engineered to deliver accurate and contextually relevant responses to user queries based on policy documents. The system is built using the LangChain framework and follows a Retrieve-and-Generate (RAG) architecture, integrating advanced document processing, semantic embeddings, and intelligent retrieval strategies.

Project Overview

Problem Statement

The primary goal of this project is to design a sophisticated generative search solution capable of extracting precise and meaningful answers from a diverse set of policy documents.

Approach

The system is implemented using LangChain, a modular framework tailored for building applications powered by Large Language Models (LLMs). LangChain offers a rich set of tools and abstractions that simplify the development of LLM-centric applications, supporting seamless integration with providers like OpenAI, Cohere, and Hugging Face.

LangChains architecture emphasizes composability and modularity, enabling rapid development of complex applications. It supports both Python and JavaScript/TypeScript environments and provides standardized interfaces for model I/O, retrieval, memory, agents, and callbacks.

Framework Overview

Core Components

LangChain is structured around two key elements:

1. **Components** Modular abstractions for core LLM operations.
2. **Use-Case Specific Chains** Preconfigured assemblies of components tailored for specific tasks.

Building Blocks

LangChain applications are built using six foundational modules:

- **Model I/O**: Interfaces for LLMs, prompts, and output parsers.
- **Retrieval**: Document loaders, transformers, embedding models, vector stores, and retrievers.
- **Chains**: Sequential LLM workflows.
- **Memory**: Persistent state across interactions.
- **Agents**: Tool selection based on high-level instructions.
- **Callbacks**: Logging and streaming for intermediate steps.

System Architecture

Document Processing Pipeline

- **PDF Ingestion**: Uses PyPDFDirectoryLoader for efficient loading of policy documents.
- **Chunking Strategy**: Implements RecursiveCharacterTextSplitter to preserve semantic coherence during segmentation.
- **Embedding Generation**: Utilizes OpenAIEmbeddings for vector representation of text, supporting similarity search and semantic analysis.
- **Vector Storage**: Embeddings are stored in ChromaDB, enhanced with CacheBackedEmbeddings for performance.
- **Retrieval**: Employs VectorStoreRetriever to fetch relevant documents based on user queries.
- **Re-ranking**: Integrates HuggingFaceCrossEncoder with the BAAI/bge-reranker-base model to improve relevance scoring.
- **Chain Integration**: Uses LangChains PromptTemplate and rlm/rag-prompt from LangChain Hub to structure RAG workflows.

Challenges Encountered

Document Processing

- Insurance PDFs are lengthy and complex.
- Maintaining semantic integrity during chunking was difficult.
- Required advanced chunking logic to preserve meaningful context.

Retrieval Strategy

- Standard retrievers often returned technically similar but contextually irrelevant results.
- Needed to combine ChromaDB with LangChain retrievers and cross-encoder reranking to improve accuracy.

Lessons Learned

Technical Insights

- **Chunking Strategy Matters**: Effective document segmentation is critical for preserving semantic meaning. RecursiveCharacterTextSplitter proved essential for maintaining context in long and complex policy documents.
- **Embedding Quality Impacts Retrieval**: The choice of embedding model significantly affects retrieval accuracy. OpenAIEmbeddings provided reliable semantic representations, but tuning parameters and experimenting with alternatives like BAAI or Cohere could further improve results.
- **Re-ranking Enhances Precision**: Integrating a cross-encoder reranker (BAAI/bge-reranker-base) substantially improved the relevance of retrieved documents, especially in cases where vector similarity alone was insufficient.

Architectural Learnings

- **Modularity Accelerates Development**: LangChains modular design enabled rapid prototyping and integration of components like retrievers, chains, and embeddings.
- **Framework Flexibility**: LangChains support for multiple providers and data sources allowed seamless experimentation and adaptation to evolving project needs.
- **Chain Composition is Powerful**: Combining chains with prompt templates and retrievers enabled the creation of highly customized RAG workflows.