****Semantic Spotter - Project****

**Fashion search AI - a Generative Search System**

By Mahalakshmi Totad

**Project's goals**

**Problem Statement**

Using Myntra dataset containing information about products to create a Generative Search System using LangChain framework components to store, retrieve and generate search results.

**Project Background**

- Fashion search AI is a Generative Search System capable of searching a plethora of product descriptions to find and recommend appropriate choices against a user query.

- Fashion search AI can combine the power of LangChain framework to use large language models with data from myntra dataset to generate relevant response for user serachs and demonstrate the knowledge of Langchain framework and it's components.

Part 2: System Design

Data sources

We have a dataset `Myntra\_fashion\_products.csv` where each row describes the features of a product and also has a containing information about products ( name, rating, gender, price , description etc). The engine that we build will leverage LLMs to parse this `Description` column and provide search results to user query

Data Analysis

We have a clean dataset so preprocessing is not required. We can move on to implementation with LangChain

Model used

\* ChatOpenAI : gpt-3.5-turbo-0125

\* OpenAIEmbeddings

Key design decisions

**Why LangChain framework** ? It that simplifies the development of LLM applications LangChain offers a suite of tools, components, and interfaces that simplify the construction of LLM-centric applications.

In this project I’m using OpenAI LLM modeks so LangChain enables developers to build applications that can generate creative and contextually relevant content, supports and 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, so it would be ideal to use with Myntra dataset that I’ll be using in this project.

**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)

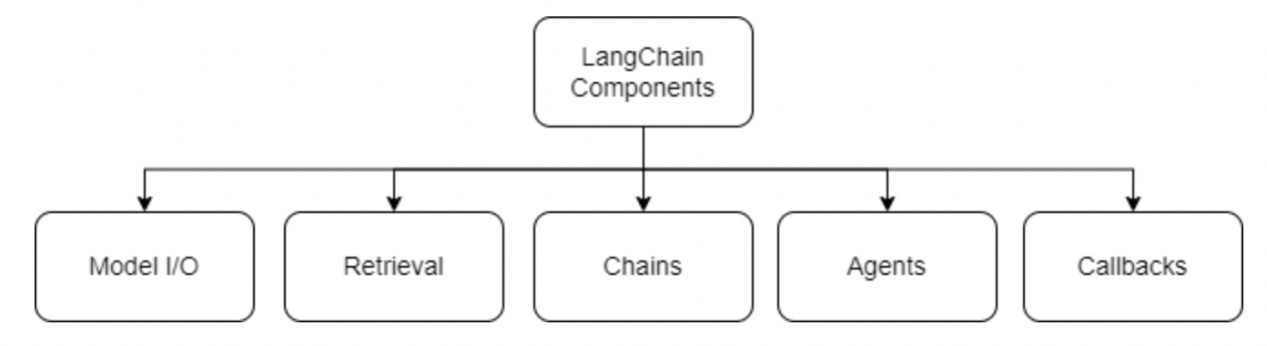
\* Retrieval: Interface with application-specific data (Document loaders, Document transformers, Text embedding models, Vector stores, Retrievers)

\* 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



**LangChain components are being used in following way:**

Model I/O component : provides support to interface with the LLM and generate responses.

The Model I/O consists of:

Prompts: I’m using the the Templatization to capture user query and form LLM prompt

Language Models: Make calls to language models through common interfaces using Prompt consisting of user query and retrieval results

Output Parsers: Extract information from model outputs

Retrieval component : will be used to retrieve the embedding results from Vector Store

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 personalizeable 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 continuosly.

Implementation

Following are the Libraries used for this project. :

- openai

- pandas

- langchain

- langchain-openai

- chromadb

- faiss-cpu

- pypdft

- tiktoken

- os

- langchain.document\_loaders -> CSVLoader

- langchain.embeddings -> OpenAIEmbeddings

- langchain.prompts -> PromptTemplate

- langchain.chains ->LLMChain

- langchain.docstore.in\_memory -> InMemoryDocstore

Data Connections and Retrieval

Many LLM applications require user-specific data that is not part of the model's training set. The primary way of accomplishing this is through Retrieval Augmented Generation (RAG). In this process, external data is retrieved and then passed to the LLM when doing the generation step.

**I'm using the following methods provided by LangChain to process documents efficiently:**

\* ***Document Loaders*** : Using **CSVLoader provided by Langchain document\_loaders**

\* ***Text Embedding*** : The base Embeddings class in LangChain provides two methods: one for embedding documents and one for embedding a query.

Using **OpenAIEmbeddings provided by LangChain embeddings**

\* ***Vector Stores*** : Using **FAISS and Chroma provided by vectorstores**

**Perform Similarity Search:** using Chroma as well as FAISS to verify Similarity search

\* ***Retrievers*** : Retrievers provide Easy way to combine documents with language models. Using **FAISS provided by vectorstores**

**for retrieval**

\* **Chains**

LangChain provides Chains that can be used to combine multiple components together to create a single, coherent application.

**I’m using the following from Langchain libraries:**

**\* LLMChain :** Using The fundamental unit of Chains is a **LLMChain** object which that takes an input, formats it, and passes it to an LLM for processing. The basic components are PromptTemplate, input queries, an LLM, and optional output parsers.

Hence using it to create a chain

**\* PromptTemplate**  provided by **langchain.prompts** that takes user input, formats it , 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.

**Finally using**

***llm\_chain.invoke(prompt\_input)***

**To generate the search results by extracting context from user query and using recommendations retrieved from Myntra data set**

Challenges you encountered/Lessons learnt

* Choosing the appropriate Text embedding model to go with selected Vector stores required some research.
* I encountered lot of version mismatch and syntax compatibility issues.
* The response of text embedding did not match the vector stores required data structure so I had to reformat the responses to match it with vector stores requirement while doing retrieval operations.
* So I can say documenting all the libraries used along with their version is a important learning especially while working in agile environment
* I tried to get the record from cvs file that matched the description of generated search result but was not successful due to time constraints.
* I can explore and learn a bit more in this area