Multi-Class Fish Image Classification

# Abstract:

In recent years, **Retrieval-Augmented Generation (RAG)** has emerged as a powerful architecture for enabling language models to generate intelligent responses grounded in factual data. This project details the development of a RAG-based question-answering (QA) system utilizing the **MongoDB sample\_mflix movie dataset**.

The dataset, initially in NoSQL format, underwent a comprehensive cleaning, flattening, and migration process to a **PostgreSQL relational database**. To enable semantic search, textual content from various collections was transformed into vector representations using **Hugging Face embeddings** and then efficiently stored in the **Pinecone vector database** in the cloud.

The core of the system implements a **LangChain RAG pipeline**, designed to fetch relevant documents based on user queries. These retrieved documents are then passed to the **Groq - Gemma2 model** to generate intelligent, context-aware responses. The project culminates in a user-friendly **Streamlit-based QA interface**, which was **containerized using Docker, with its image built and pushed to Docker Hub, and subsequently deployed on an EC2 instance** for broad accessibility. This robust deployment ensures seamless interaction and delivers accurate responses derived from both structured and unstructured movie data.

# Objectives:

The primary objectives of this project are:

* To extract and transform movie-related data from MongoDB into a clean, SQL-compatible format.
* To design and implement a **semantic search mechanism** using OpenAI embeddings and FAISS.
* To build a **Retrieval-Augmented Generation (RAG)** pipeline that retrieves relevant information and generates context-based answers.
* To develop an interactive **Streamlit UI** for real-time user queries over the movie dataset.
* To apply secure practices like .env credential masking and modular code structuring for project scalability.

# Technologies & Tools Used:

## Languages & Frameworks:

* **Python 3.10+**: Core language for scripting, data processing, and application logic.
* **Streamlit**: For building the interactive, user-friendly frontend Question-Answering (QA) interface.
* **LangChain**: The framework used to orchestrate the RAG pipeline, integrating various components like LLMs, embeddings, and vector stores.

## Databases & Vector Store:

* **MongoDB**: The original NoSQL source database for the movie dataset.
* **PostgreSQL**: The target relational database where the cleaned and flattened movie data was migrated.
* **Pinecone**: A cloud-native vector database used for storing and efficiently retrieving vector embeddings

## Libraries & API:

* **pymongo**: For connecting to and interacting with MongoDB.
* **sqlalchemy**: For interacting with relational databases (used for PostgreSQL).
* **psycopg2**: PostgreSQL adapter for Python.
* **pandas**: For data cleaning, transformation, and manipulation.
* **langchain-groq**: To integrate with Groq's API for the Gemma2 Large Language Model.
* **langchain-huggingface**: To integrate Hugging Face models for generating vector embeddings.
* **langchain-pinecone**: For connecting LangChain with the Pinecone vector database.
* **pinecone**: The official client library for Pinecone.
* **sentence-transformers**: The underlying library for Hugging Face embeddings.
* **dotenv**: To securely manage environment variables and API credentials.

## Security:

* **.env File**: Used to store and protect sensitive credentials (API keys, database URIs).

## Deployment & Orchestration:

* **Docker**: Used for containerizing the Streamlit application, enabling consistent environments and easy deployment.
* **Docker Hub**: A cloud-based registry service for storing and sharing Docker images.
* **AWS EC2**: The cloud computing service where the containerized Streamlit application was deployed.

# Private Credential Handling:

To ensure the safety and privacy of sensitive credentials like:

* **MongoDB URI**
* **PostgreSQL database URL**
* **Groq API Key**
* **Pinecone API Key**

We have utilized Python's python-dotenv package. All sensitive data is stored in a .env file and accessed in code via os.getenv(). This approach ensures:

* Credentials are not exposed directly within the codebase.
* Safer collaboration and public deployment on platforms like GitHub by excluding the .env file from version control.
* Easy replacement and management of credentials without altering the core application logic.

# Methodology:

## ETL Process

### Extract:

We began by connecting to the MongoDB Atlas cloud instance. The following collections were extracted:

* movies
* comments
* users
* theaters
* sessions
* Each document was converted to a Python dictionary for further processing.

### Transform:

The documents were cleaned and transformed:

* Fields like cast, genres, languages were converted from list to comma-separated string.
* Missing values were filled or dropped.
* Dates were formatted for SQL compatibility.
* Foreign key fields (like movie\_id) were type-cast to strings.

### Load:

* We built a relational schema in **PostgreSQL**, where tables mirrored the structure of the original collections. Primary and foreign key relationships were established to ensure data integrity.
* All transformed data was loaded into SQL tables using Python’s SQLAlchemy and raw INSERT queries.

## Embedding Preparation

### Merging Data for Text Input:

* To prepare the movie data for embedding, all relevant information from the joined PostgreSQL tables was systematically combined into a list of comprehensive, human-readable strings. This process was managed by a custom get\_final\_merged\_list() function, designed to synthesize various attributes into a cohesive narrative for each movie.
* **Example output:** "Inception is a mind-bending thriller starring Leonardo DiCaprio. Directed by Christopher Nolan, this sci-fi film delves into the world of dreams and features a complex plot about extracting information from a target's subconscious."

### Text Chunking:

* Leveraging LangChain's RecursiveCharacterTextSplitter, these larger textual descriptions were intelligently segmented into smaller, manageable chunks. This ensures compatibility with the input token limits of Large Language Models, with each chunk configured to have a maximum size of 1000 characters and an overlap of 200 characters to maintain contextual continuity.

### Embeddings:

* Each of these processed text chunks was then transformed into high-dimensional vector representations using **Hugging Face Embeddings**. These sophisticated embeddings capture the semantic meaning of the text. The resulting vector embeddings were subsequently stored in the **Pinecone vector database**, optimizing them for efficient semantic search and retrieval within the RAG pipeline.

## RAG Pipeline Implementation

### Retriever Setup:

* The **Pinecone vector database**, populated with the movie embeddings, functions as the primary retriever. Upon receiving a user query, this vector store efficiently identifies and retrieves the most semantically similar and relevant text chunks from the extensive movie dataset. This ensures that only the most pertinent information is considered for generating the answer.

### LLM Response Generation:

* The retrieved, contextually relevant documents, along with the original user query, are then meticulously passed to **Groq's Gemma2 model**. LangChain plays a crucial role in this phase, orchestrating the entire process of chaining the components, formatting the input for the LLM, and parsing the generated responses to deliver coherent and accurate answers.

### Frontend Interface:

**Streamlit** was utilized to construct an intuitive and interactive user interface for the QA system. Users can engage with the system by inputting natural language questions, such as:

* + "What is the plot of Interstellar?"
  + "List the top actors in Sci-Fi movies."

The interface is designed to fetch and display the generated response in real-time, clearly presenting:

* + The final, contextually informed answer.
  + The supporting documents (context) that were used to formulate the answer, accessible via an expandable section for transparency.

# Deployment:

To ensure seamless accessibility and robust operation of the MovieMax QA application, a containerized deployment strategy was implemented. This approach leverages **Docker** for packaging the application and its dependencies, followed by deployment on a cloud-based **EC2 instance** on **Amazon Web Services (AWS)**.

## Containerization with Docker

The entire Streamlit application, along with its Python dependencies and environment configurations, was encapsulated within a **Docker image**. This process involved creating a Dockerfile that specifies the base image, installs necessary libraries, copies the application code, and defines the entry point for running the Streamlit app.

The benefits of containerization include:

* **Consistency:** The application runs in a uniform environment, regardless of the underlying infrastructure, eliminating "it works on my machine" issues.
* **Isolation:** The app's dependencies are isolated from the host system, preventing conflicts.
* **Portability:** The Docker image can be easily moved and run across various environments (local, cloud, on-premises).

Once the Docker image was successfully built, it was **pushed to Docker Hub**. Docker Hub serves as a centralized cloud registry for Docker images, making it simple to store, share, and pull the image from any environment.

## Cloud Deployment on AWS EC2

For public accessibility and scalability, the containerized Streamlit application was deployed on an **AWS EC2 (Elastic Compute Cloud) instance**.

The deployment steps involved:

1. **Provisioning an EC2 Instance:** A suitable EC2 instance was launched, configured with an appropriate operating system (e.g., Ubuntu or Amazon Linux).
2. **Docker Installation:** Docker was installed on the EC2 instance to enable container runtime.
3. **Image Pull:** The Docker image was pulled from **Docker Hub** onto the EC2 instance.
4. **Container Launch:** The Streamlit application was launched as a Docker container, mapping the container's internal port (8501) to the EC2 instance's public port to make the Streamlit UI accessible via the internet.
5. **Security Group Configuration:** AWS Security Groups were configured to allow inbound traffic on the designated port (e.g., 8501) to ensure users could access the application from their web browsers.

This deployment method provides a scalable and maintainable solution, allowing the MovieMax QA system to be reliably accessed by users from anywhere.

# Results:

|  |  |
| --- | --- |
| **User Query** | **Model Answer** |
| “What is the plot of Inception?” | “Inception is a sci-fi thriller where a thief enters dreams to steal secrets...” |
| “Who acted in The Dark Knight?” | “The cast includes Christian Bale, Heath Ledger, Michael Caine...” |
| “What genres are most popular?” | “Top genres include Action, Drama, Sci-Fi, and Thriller.” |

# Conclusion:

This project successfully developed and deployed a robust **Retrieval-Augmented Generation (RAG)** based question-answering system for movie information. By meticulously transforming the **MongoDB sample\_mflix dataset** into a structured **PostgreSQL database** and leveraging **Hugging Face embeddings** stored in **Pinecone**, the system established a powerful semantic search capability.

The integration of **LangChain** facilitated a streamlined RAG pipeline, enabling efficient retrieval of relevant context which was then intelligently synthesized by **Groq's Gemma2 model** to generate accurate and contextually rich answers. The user-friendly **Streamlit interface** provides an intuitive way for users to interact with the system, making complex movie data accessible through natural language queries.

Furthermore, the entire application was **containerized using Docker**, built into an image, pushed to **Docker Hub**, and finally deployed on an **AWS EC2 instance**. This deployment strategy ensures the application's portability, consistency, and accessibility, demonstrating a complete end-to-end RAG solution from data preparation and model integration to scalable cloud deployment. This system stands as a testament to the power of combining traditional databases with modern LLM and vector database technologies to deliver intelligent, data-grounded responses.