

**VIETNAM NATIONAL UNIVERSITY – HCM
INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**



**MACHINE LEARNING PLATFORMS
PROJECT REPORT**

Semester 1, 2024–2025

Instructor: Ho Long Van

Topic: Movies Recommender System

TEAM MEMBERS

No.	Name	Student ID	Contribution
1	Nguyen Huy Bao	ITDSIU21076	33%
2	Nguyen Nhat Khiem	ITDSIU21091	33%
3	Trinh Binh Nguyen	ITDSIU21103	33%

TABLE OF CONTENTS

Contents

1	INTRODUCTION	7
1.1	Overview	7
1.2	Objectives	7
1.3	About the dataset	7
1.4	Specifications	7
1.5	Project workflow	9
2	DATA SELECTION AND PREPROCESSING	9
2.1	Data selection	9
2.2	Preprocessing Steps	10
2.3	Handling Missing Values	10
2.4	Preprocessing Steps	10
2.5	Preprocessing Steps	11
3	VECTOR DATABASE SELECTION AND CREATION	11
3.1	Selection Criteria	11
3.2	Database Implementation	13
3.2.1	Database Choice	13
3.2.2	Index Creation	13
3.2.3	Data Insertion	13
3.2.4	Vector Search Implementation	14
3.2.5	Advantages of HNSW and Cosine Similarity	14
3.2.6	Error Handling and Robustness	14
4	UTILITIES IMPLEMENTATION	16
4.1	Embedding Generation	16
4.2	Smooth Description Generation	16
4.3	Search and Recommendation	17
4.4	Data Display	17
5	USER INTERFACE DESIGN AND AI INTEGRATION	17
5.1	User Interface	17
5.2	AI Integration	18
6	CONCLUSION	19
6.1	Accomplishments	19
6.2	Limitations	21
6.3	Future Works	21
6.4	References	22

TABLE OF FIGURES

List of Figures

1	Project's Pipeline	9
2	Illustration of the dataset structure and key columns.	10
3	Illustration of preprocessing steps applied to the dataset.	10
4	Illustration of handling missing data during preprocessing.	11
5	Illustration of embedding generation and storage during preprocessing.	12
6	Database implementation showcasing MongoDB's vector search and HNSW index configuration.	14
7	Vector Search Pipeline	15
8	Illustration of smooth description generation using the llama 3.2 model.	16
9	Illustration of data display interface with interactive features.	18
10	Illustration of the user interface showing the initial data preview.	18
11	Illustration of the user interface showing movie recommendations with posters and descriptions.	19
12	Illustration of AI integration for embedding comparison and natural language output generation.	19

PROJECT TIMELINE

Stage	Task	Member	Week No.
Planning	Individual's topic research	All	1 - 2
	Set up project timeline, design workflow	Khiem	1 - 2
	Topic confirmation	All	1 - 2
	Identify the objectives and purposes of the analysis	All	1 - 2
	Agree on means of communication, workflow and tools	All	1 - 2
	Decide scopes, aims, goals and requirements of the project	All	1 - 2
	Create GitHub repository	Khiem	1 - 2
Data Selection & Pre-processing	Data searching and confirmation	Khiem	3
	Detect and deal with missing, duplicated values	Nguyen	3
	Detect and deal with outliers	Nguyen	3
	Compare pre-process and post-process	Bao	3

Stage	Task	Member	Week No.
Set up Vector Database	Select the database platform and set up connection	Nguyen	3 - 7
	Create a vector database schema with embeddings	Nguyen	3 - 7
	Establish index search on vector database	Nguyen	3 - 7
	Set up the vector search function for choosing essential info for retrieval	Nguyen	3 - 7
Implement Utilities	Set up embedding creation for both data and user input	Bao	3 - 7
	Set up function to leverage embedding for natural answer retrieval and practical use	Bao	3 - 7
Design User Interface	Display search results with add more function	Khiem	9 - 12
	Set up Streamlit design for UI	Khiem	9 - 12
	Integrate embedding and language model for text generation into UI	Khiem	9 - 12
Report & Presentation	Report form setting up	Khiem	13 - 15
	Data Overview and Preprocessing	Nguyen	13 - 15
	Database selection and implementation	Nguyen	13 - 15
	Utils functions details	Bao	13 - 15
	UI implementation and AI integration	Khiem	13 - 15
	Accomplishments, Restrictions, Future works	Khiem	13 - 15

1 INTRODUCTION

1.1 Overview

The Film Recommender System is designed to suggest movies by comparing their stories and identifying those that are most similar. It stores information about movie plots in a special database that helps find connections between them. By focusing on the content of the stories rather than just basic details like genre or year, the system can provide recommendations that feel more relevant and meaningful. With its simple and easy-to-use interface, the system offers users a fun and engaging way to discover movies tailored to their interests.

1.2 Objectives

The goal of this project is to develop a user-friendly film recommender system that provides personalized movie suggestions by analyzing plot content and themes. It aims to enhance the movie discovery experience using advanced technologies, ensuring accurate and engaging recommendations tailored to users' preferences.

1.3 About the dataset

The chosen dataset contains detailed information about movies in the Western, Action, or Fantasy genres. Each record includes attributes such as title, release year, cast, plot summary, genres, runtime, ratings, directors, writers, production countries, languages, and awards. Additionally, the dataset features a `plot_embedding` field with embeddings generated using OpenAI's `text-embedding-ada-002` model, enabling advanced search and analysis based on plot similarities.

For our project, we replaced these pre-generated embeddings with custom embeddings created using the Ollama Nomic text embedding model to better align with the system's specific requirements, limitations, and improve recommendation accuracy.

The dataset contains detailed information about movies, including:

- **Plot:** A brief description of the movie storyline.
- **Runtime:** Duration of the movie.
- **Genres:** Categories such as Drama, Action, etc.
- **Directors, Writers, Cast:** Key contributors to the movie.
- **Countries, Languages:** Locations and languages associated with the movie.
- **Metacritic:** Critical rating of the movie (dropped during preprocessing).

float

1.4 Specifications

Tool	Purpose
Python 3	For interpreting and running Python scripts
Streamlit	For creating attractive and straightforward user interfaces
MongoDB Atlas Cloud	For embedding vector creation and storage
Ollama	For embedding model and LLM integration to the system

Table 1: Tools and their purposes used in the project.

1.5 Project workflow

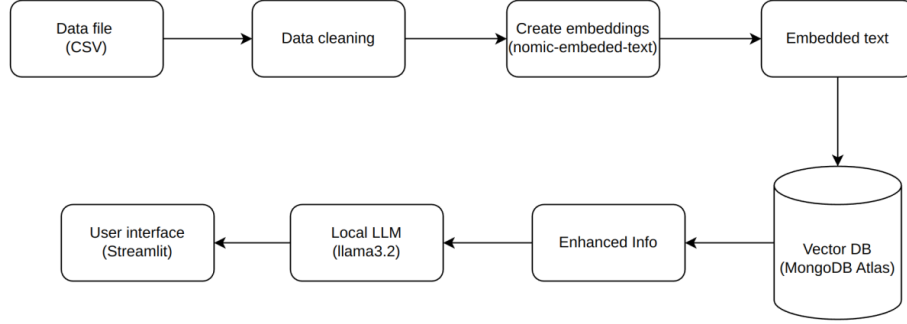


Figure 1: Project's Pipeline

The project's workflow involves the following steps:

1. **Data File (CSV):** The system begins with a dataset containing raw movie information, including plot details and metadata, stored in a CSV file.
2. **Data Cleaning:** The raw data undergoes a cleaning process to ensure consistency, remove noise, and prepare it for embedding generation.
3. **Create Embeddings:** Using the Nomic-embedded-text model, movie plots are converted into vector embeddings that capture their semantic meaning.
4. **Embedded Text:** These embeddings are then stored in a Vector Database (MongoDB Atlas), enabling efficient similarity searches and data retrieval.
5. **Enhanced Information:** The stored embeddings and metadata are accessed to generate improved recommendations by leveraging additional context.
6. **Local LLM (Llama 3.2):** A local language model processes the retrieved data to refine and polish the output, ensuring the recommendations are user-friendly and contextually appropriate.
7. **User Interface (Streamlit):** Finally, the recommendations are displayed in a streamlined and interactive user interface built with Streamlit, allowing users to easily explore suggested movies.

2 DATA SELECTION AND PREPROCESSING

2.1 Data selection

The dataset was selected for its comprehensive coverage of movie metadata, allowing for diverse and meaningful recommendations. Key columns included are illustrated below:

- **Plot descriptions:** Used for semantic embedding.

- **Genre and runtime:** Used for metadata-based filtering.
- **Contributors and filming locations:** Provide added context for recommendations.

	plot	runtime	genres	fullplot	directors	writers	countries	poster	languages	cast	title
0	Young Pauline is left a lot of money when her ...	199.0	['Action']	Young Pauline is left a lot of money when her ...	['Louis J. Gasnier', 'Donald MacKenzie']	['Charles W. Goddard (screenplay)', 'Basil Dic...']	['USA']	amazon.com/images/M/MV5BMzgxDOD...	['English']	['Pearl White', 'Crane Wilbur', 'Paul Panzer', ...]	The Perils of Pauline
1	A penniless young man tries to save an heiress...	22.0	['Comedy', 'Short', 'Action']	As a penniless man worries about how he will m...	['Alfred J. Goulding', 'Hal Roach']	['H.M. Walker (titles)']	['USA']	amazon.com/images/M/MV5BNzE1OW...	['English']	['Harold Lloyd', 'Mildred Davis', 'Snub Poll...	From Hand to Mouth
2	Michael "Beau" Geste leaves England in disgrac...	101.0	['Action', 'Adventure', 'Drama']	Michael "Beau" Geste leaves England in disgrac...	['Herbert Brenon']	['Herbert Brenon (adaptation)', 'John Russell ...']	['USA']	NaN	['English']	['Ronald Colman', 'Neil Hamilton', 'Ralph Forb...	Beau Geste

Figure 2: Illustration of the dataset structure and key columns.

2.2 Preprocessing Steps

1. Column Cleaning:

- Removed unnecessary columns like *metacritic*.
- Converted string representations of lists (e.g., directors, cast) into comma-separated strings.

```

1 import ast
2 column_list = ['directors', 'writers', 'cast', 'countries', 'languages']
3
4 # Drop the 'metacritics' column
5 df.drop(columns='metacritic', inplace=True)
6
7 # Convert string representations of lists to comma-separated strings
8 for col in column_list:
9     df[col] = df[col].apply(lambda x: ', '.join(ast.literal_eval(x)) if isinstance(x, str) else x)

```

Figure 3: Illustration of preprocessing steps applied to the dataset.

2.3 Handling Missing Values

2.4 Preprocessing Steps

1. Column Cleaning:

- Removed unnecessary columns like *metacritic*.
- Converted string representations of lists (e.g., directors, cast) into comma-separated strings.

2. Handling Missing Data:

- Imputed missing values where necessary.
- Dropped rows with insufficient data in critical fields such as *plot*, *poster*, *rated*, and *metacritics*.

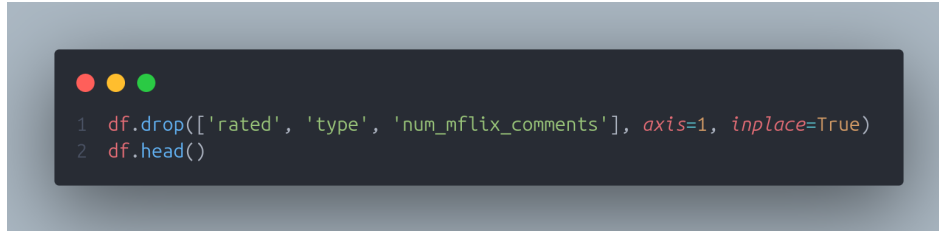


Figure 4: Illustration of handling missing data during preprocessing.

2.5 Preprocessing Steps

1. Column Cleaning:

- Removed unnecessary columns like *metacritic*.
- Converted string representations of lists (e.g., directors, cast) into comma-separated strings.

2. Handling Missing Data:

- Imputed missing values where necessary.
- Dropped rows with insufficient data in critical fields such as *plot*, *poster*, *rated*, and *metacritics*.

3. Embedding Generation:

- Transformed *plot* into vector embeddings using the *nomie-embedded-text* model through the Ollama API.
- Added a new column *plot_embedding* to store the generated vectors (as described below).

3 VECTOR DATABASE SELECTION AND CREATION

3.1 Selection Criteria

The selection of the Vector Database (VectorDB) for this project was guided by the following essential criteria:

1. Scalability:



Figure 5: Illustration of embedding generation and storage during preprocessing.

- The database needed to efficiently handle large datasets, such as those containing thousands of movie embeddings. This ensures that the system remains responsive and effective as the dataset grows in size over time.
- Scalability is crucial for real-world applications, where increasing user interactions or additional data sources can lead to exponential growth in stored information.

2. Similarity Search:

- High-speed nearest neighbor search was a key requirement for identifying embeddings that are most similar to a given query. This is particularly important for use cases like movie recommendations, where quick and accurate retrieval of relevant data directly impacts the user experience.
- The ability to compare vector embeddings with minimal latency was a non-negotiable feature, given the need for real-time query resolution.

3. Integration:

- The chosen VectorDB had to seamlessly integrate with machine learning workflows, allowing for direct embedding storage and retrieval. This compatibility eliminates the need for additional data conversion layers and simplifies the overall architecture.
- Support for Python-based APIs and libraries, such as `pymongo`, was prioritized to align with the existing tech stack.

4. Accessibility:

- The database should be easy to configure and accessible through standard Uniform Resource Identifiers (URIs). This simplifies the setup process and allows for rapid deployment across various environments, including local, cloud, and hybrid setups.
- MongoDB’s flexibility in providing managed services and cross-platform support enhanced its appeal as the database of choice.

3.2 Database Implementation

3.2.1 Database Choice

- MongoDB was selected for its ability to handle unstructured and semi-structured data efficiently. It offers features such as flexible schema design, high availability, and horizontal scaling, making it an ideal choice for embedding-based applications.
- MongoDB’s built-in vector search capabilities further solidified its position as a robust solution for this project. These capabilities allow for the direct handling of high-dimensional vector data without requiring additional plugins or tools.

3.2.2 Index Creation

- To enable efficient similarity search, a Hierarchical Navigable Small World (HNSW) index was created on the *plot_embedding* field. This index supports the cosine similarity metric, which is particularly effective for comparing embeddings derived from machine learning models.
- The HNSW algorithm ensures low-latency retrieval by organizing vectors into a navigable graph structure, allowing for quick nearest neighbor searches.
- The index was configured to accommodate embeddings with 768 dimensions, reflecting the output size of the machine learning model used for feature extraction.
- This ensures that the database is optimized for high-performance queries, even as the dataset grows.

3.2.3 Data Insertion

- The insertion process involved preprocessing movie data, including title, genres, director, and plot embeddings generated by a pre-trained language model. These embeddings, stored as 768-dimensional vectors, represent the semantic meaning of the movie descriptions.
- Each data entry was structured as a document in MongoDB, with fields such as title, director, genres, plot_embedding, and other relevant metadata.



Figure 6: Database implementation showcasing MongoDB’s vector search and HNSW index configuration.

3.2.4 Vector Search Implementation

- MongoDB’s **VectorSearch** stage was used for querying the vector data. The query involved providing a 768-dimensional vector as input and retrieving the top matching documents based on cosine similarity.
- The pipeline for vector search was designed to include stages for similarity scoring, data projection, and optional field removal for better performance (as shown in Figure 7).

3.2.5 Advantages of HNSW and Cosine Similarity

- The HNSW index offers logarithmic complexity for search operations, ensuring fast response times.
- Cosine similarity is particularly well suited for text embeddings as it focuses on the angle between vectors rather than their magnitude, ensuring robust comparisons even when vector scales vary.

3.2.6 Error Handling and Robustness

- Comprehensive error handling mechanisms were implemented to manage potential issues during data insertion, index creation, and query execution.
- This ensures system stability and a better user experience.

```

1 def vector_search(user_query, collection: Collection, field):
2     query_embedding = create_embedding_query(user_query)
3     if not query_embedding:
4         st.error("Invalid query or empty embedding")
5         return 'Invalid query'
6
7     vector_search_stage = {
8         "$vectorSearch": {
9             "index": "vector_search_index",
10            "queryVector": query_embedding,
11            "path": field,
12            "numCandidates": 25,
13            'limit': 20
14        }
15    }
16    unset_stage = {
17        '$unset': field
18    }
19
20    project_stage = {
21        '$project': {
22            '_id': 0,
23            'fullplot': 1,
24            'title': 1,
25            'director': 1,
26            'countries': 1,
27            'genres': 1,
28            'poster': 1,
29            'score': {
30                '$meta': 'vectorSearchScore'
31            }
32        }
33    }
34
35    pipeline = [vector_search_stage, unset_stage, project_stage]
36
37    try:
38        results = collection.aggregate(pipeline)
39        return list(results)
40    except Exception:
41        st.error("Error performing vector search!")
42        return []

```

Figure 7: Vector Search Pipeline

4 UTILITIES IMPLEMENTATION

4.1 Embedding Generation

1. Model Used:

The *nomie-embed-text* model from Ollama was employed for generating 768-dimensional embeddings.

2. Embedding Process:

- Each movie plot is passed through the embedding model to generate high-dimensional vector representations.
- The embeddings are stored in MongoDB's *plot_embedding* field for efficient similarity matching.

3. Query Embedding:

- User input is also embedded into a vector, which is then compared against stored embeddings to retrieve similar movies.

4.2 Smooth Description Generation

1. Model Used:

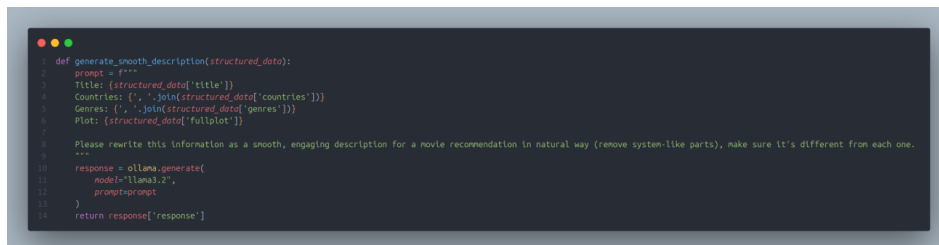
- The *Lamas 3.2* local language model was integrated for the generation of descriptions.

2. Functionality:

- Structured metadata such as title, genre, and plot are processed by the language model.
- The model transforms the information into a conversational and humanized format.

3. Usage:

- This utility is invoked after retrieving search results to refine and polish the movie details presented to users.



```
1 def generate_smooth_description(structured_data):
2     prompt = f"""
3     Title: {structured_data['title']}
4     Countries: {', '.join(structured_data['countries'])}
5     Genres: {', '.join(structured_data['genres'])}
6     Plot: {structured_data['fullplot']}
7
8     Please rewrite this information as a smooth, engaging description for a movie recommendation in natural way (remove system-like parts), make sure it's different from each one.
9 """
10    response = ollama.generate(
11        model="llama3.2",
12        prompt=prompt
13    )
14    return response['response']
```

Figure 8: Illustration of smooth description generation using the llama 3.2 model.

4.3 Search and Recommendation

1. Vector Search:

- MongoDB's `VectorSearch` stage is utilized to perform similarity searches based on cosine similarity.
- User queries are matched with the stored embeddings, retrieving the top-ranked movies.

2. Pipeline:

- Query embedding creation.
- Vector search execution with similarity scoring.
- Projection of relevant fields such as title, plot, genres, and scores.

3. Dynamic Retrieval:

- The system supports iterative queries, allowing users to retrieve additional results incrementally.

4.4 Data Display

1. Interactive Features:

- Display of movie titles, smooth descriptions, and metadata.
- Inclusion of movie posters (if available) for visual appeal.

2. Dynamic Loading:

- Users can load more results with a "Get More Results" button.
- The results are presented in batches, ensuring smooth scrolling and usability.

3. Error Handling:

- Messages are displayed for invalid queries or missing results.

5 USER INTERFACE DESIGN AND AI INTEGRATION

5.1 User Interface

1. Developed using Streamlit for an intuitive and interactive experience.

- First, once the data is loaded into the system, it displays the first 10 rows of the dataset. This provides users with a basic intuition about the data structure and its contents.

2. Interact to initialize the data preparation and vector database.

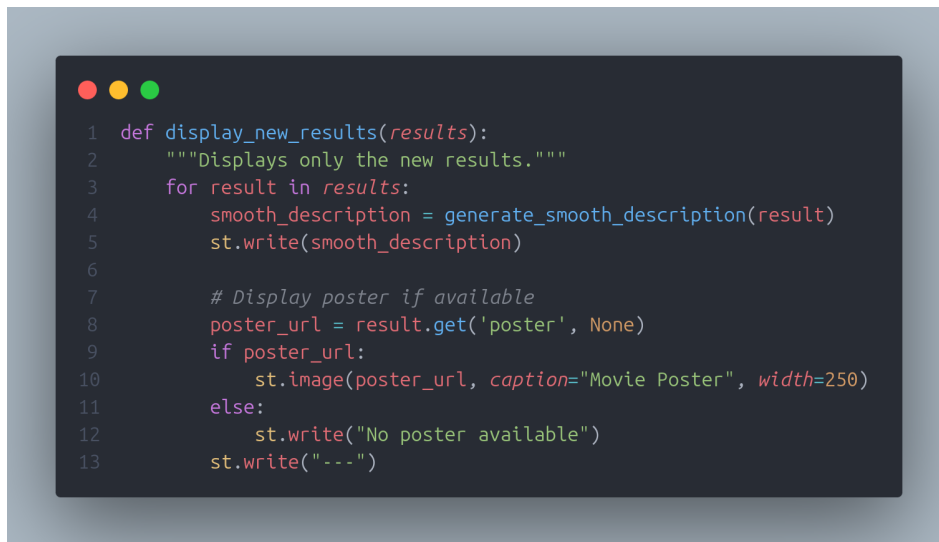


Figure 9: Illustration of data display interface with interactive features.

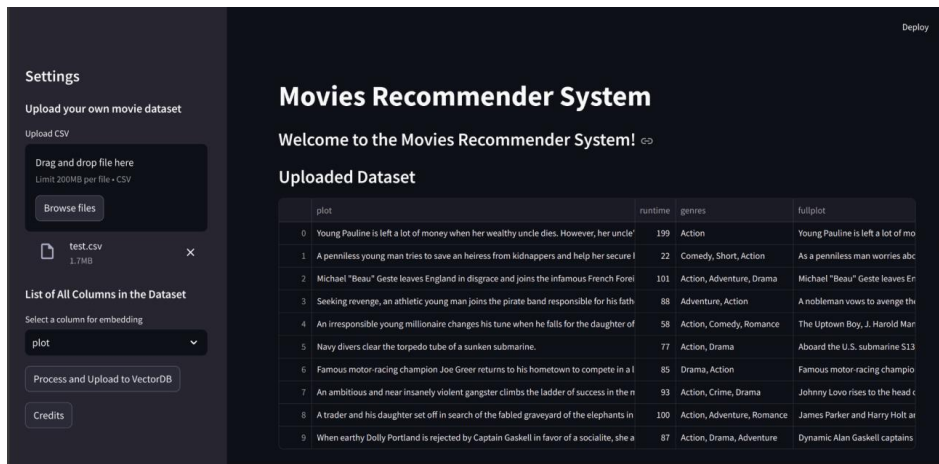


Figure 10: Illustration of the user interface showing the initial data preview.

- After the data is fully loaded, users need to press *Process and Upload to VectorDB* for the system to be set up and ready for use.

3. Recommended movies with posters and descriptions.

- Next, users can enter the plot of the film they want to search into the user chat box, hit *Enter*, and press the *Search* button. The system will then generate the desired films, displaying recommended movies with posters and descriptions.

5.2 AI Integration

Once users have entered the input for the plot, the system leverages AI models to generate recommendations. The integration process involves the following steps:

1. User Input Embedding:

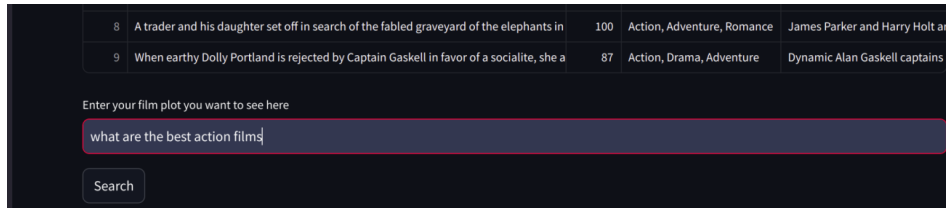


Figure 11: Illustration of the user interface showing movie recommendations with posters and descriptions.

- A dedicated model is employed to create embeddings from the user input.
- These embeddings are then compared to those stored in the VectorDB to identify similar movies.

2. Natural Language Output with LLM:

- Another LLM (*llama 3.2*) processes the embeddings of both the user input and the stored ones.
- This step ensures the recommendations are presented in a conversational and natural human language format for better user engagement.

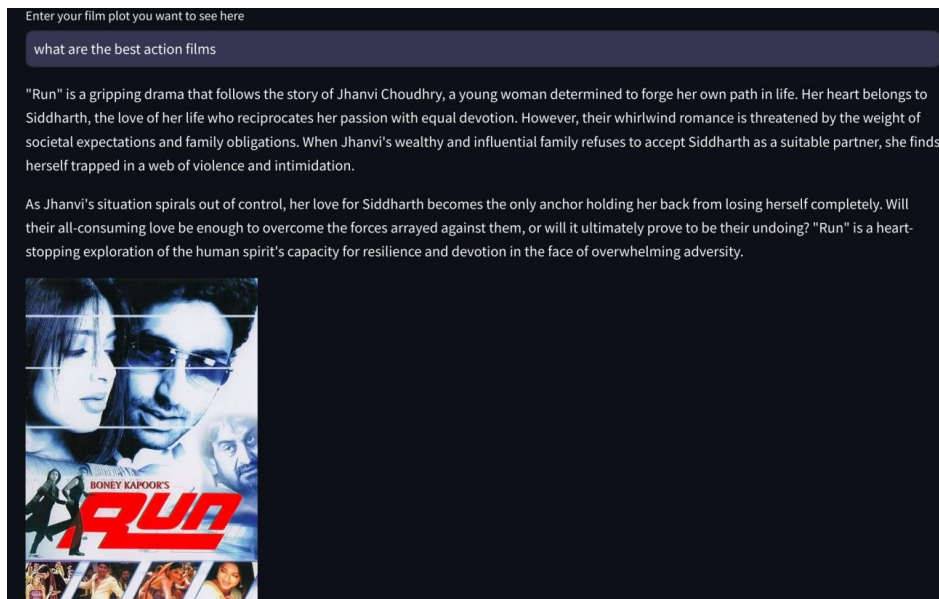


Figure 12: Illustration of AI integration for embedding comparison and natural language output generation.

6 CONCLUSION

6.1 Accomplishments

1. Effective Embedding Implementation:

- The system successfully replaced pre-existing embeddings with custom embeddings generated using the *Nomic-embedded-text* model. These embeddings were specifically tailored to capture the subtle nuances of movie plots, ensuring more accurate and contextually relevant recommendations.
- A key accomplishment of this implementation is its ability to run smoothly on a local machine, requiring minimal computational resources while maintaining high efficiency. This makes the system accessible for smaller-scale deployments without the need for expensive infrastructure.

2. End-to-End System Design:

- A complete workflow was designed and implemented, covering every stage from raw data preprocessing to final user interaction. This includes steps such as cleaning and structuring the input data, generating embeddings, storing them in a Vector Database, and retrieving refined recommendations.
- The design ensures a seamless flow of data and an efficient recommendation process, with a focus on simplicity, robustness, and scalability. The system’s modular architecture allows for easy updates and enhancements, making it both adaptable and future-ready.

3. Refined Recommendation Generation:

- The *Llama 3.2* local language model was leveraged to refine and contextualize movie recommendations, providing users with clear and engaging suggestions.
- This model enhances the system’s ability to deliver personalized outputs by polishing and formatting the retrieved information into user-friendly recommendations.
- Furthermore, it was optimized to run efficiently on a local machine, requiring minimal computational resources. This lightweight design makes the system practical for personal or small-scale use without compromising the quality of the recommendations.

4. Real-World Application:

- The system demonstrates practical value by successfully integrating advanced machine learning techniques with modern database technologies.
- By focusing on semantic plot similarities rather than superficial metadata, the recommender provides meaningful and personalized suggestions.
- Its lightweight and resource-efficient design ensures that it can be deployed on local machines, making it suitable for a variety of real-world scenarios.
- This project showcases how cutting-edge technologies can be utilized to solve practical problems and enhance user experiences, bridging the gap between research and application.

6.2 Limitations

1. Dependence on Dataset Quality:

- The effectiveness of the system heavily relies on the quality and completeness of the dataset. Incomplete or inaccurate data can significantly affect the accuracy of recommendations and limit the system's ability to provide meaningful results.

2. Limited Multi-Lingual Support:

- Currently, the system has limited support for multi-lingual movie descriptions. This restricts its usability for non-English datasets or international users who prefer recommendations in their native language.

3. Restricted Search Indexing:

- The system is limited by the free-tier indexing constraints of MongoDB Atlas, which restricts the number of search indexes that can be created for different datasets. This can pose challenges when scaling or adapting the system to accommodate larger or more diverse datasets.

4. Plot-Based Recommendations Only:

- The system is designed to generate recommendations solely based on the similarity of movie plots. It does not consider other factors like awards, ratings, or suggestions based on directors or actors. This focus on plot content, while useful, limits the range and versatility of the recommendations.

6.3 Future Works

1. Enhanced Embedding and Search Indexing:

- To support a wider range of queries and use cases, future improvements will focus on increasing the availability of embeddings and expanding the number of search indexes.
- This could involve upgrading to a higher tier of MongoDB Atlas or exploring alternative vector databases with fewer limitations.

2. Expanding the Dataset:

- The dataset will be expanded to include more diverse and multi-lingual movies, enabling the system to cater to a broader audience and deliver recommendations in multiple languages.
- This will also enhance the system's global applicability.

3. Improved Embedding Models:

- Future iterations will involve upgrading the embedding models to better capture deeper semantic relationships in movie plots.

- This could include using more advanced models or fine-tuning existing ones to improve accuracy and contextual understanding.

4. Incorporating Additional Recommendation Algorithms:

- To provide a richer recommendation experience, the system will integrate additional algorithms, such as user profile-based filtering or collaborative filtering.
- Comparative evaluations will be conducted to optimize the performance and match recommendations more closely to user preferences.

6.4 References

1. MongoDB. (n.d.). Create an Atlas Search Index. [Website]. Retrieved from <https://www.mongodb.com/docs/atlas/atlas-search/create-index/>
2. Streamlit Docs. (n.d.). Retrieved from <https://docs.streamlit.io/>
3. ProtonX. (2024, May 23). Chi tiet ve RAG - Retrieval Augmented Generation. [Video]. YouTube. Retrieved from <https://www.youtube.com/watch?v=6523788337e7f90012890145>
4. llama3.2. (n.d.). Retrieved from <https://ollama.com/library/llama3.2>
5. Hugging Face Datasets. (n.d.). Retrieved from https://huggingface.co/datasets/MongoDB/embedded_movies
6. nomic-embed-text. (n.d.). Retrieved from <https://ollama.com/library/nomic-embed-text>