**Movie Recommendation System with Sentiment Analysis**

*A Project Report Submitted in Partial Fulfilment,*

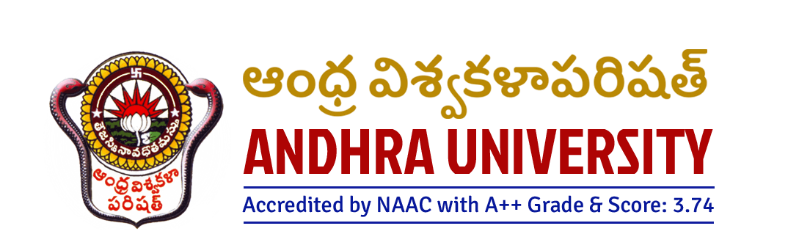
*of the Requirements for the Degree of*

**Master of Computer Applications**

**Submitted by**

**Samminga Akshay**  
**Roll No: A23CA0016**

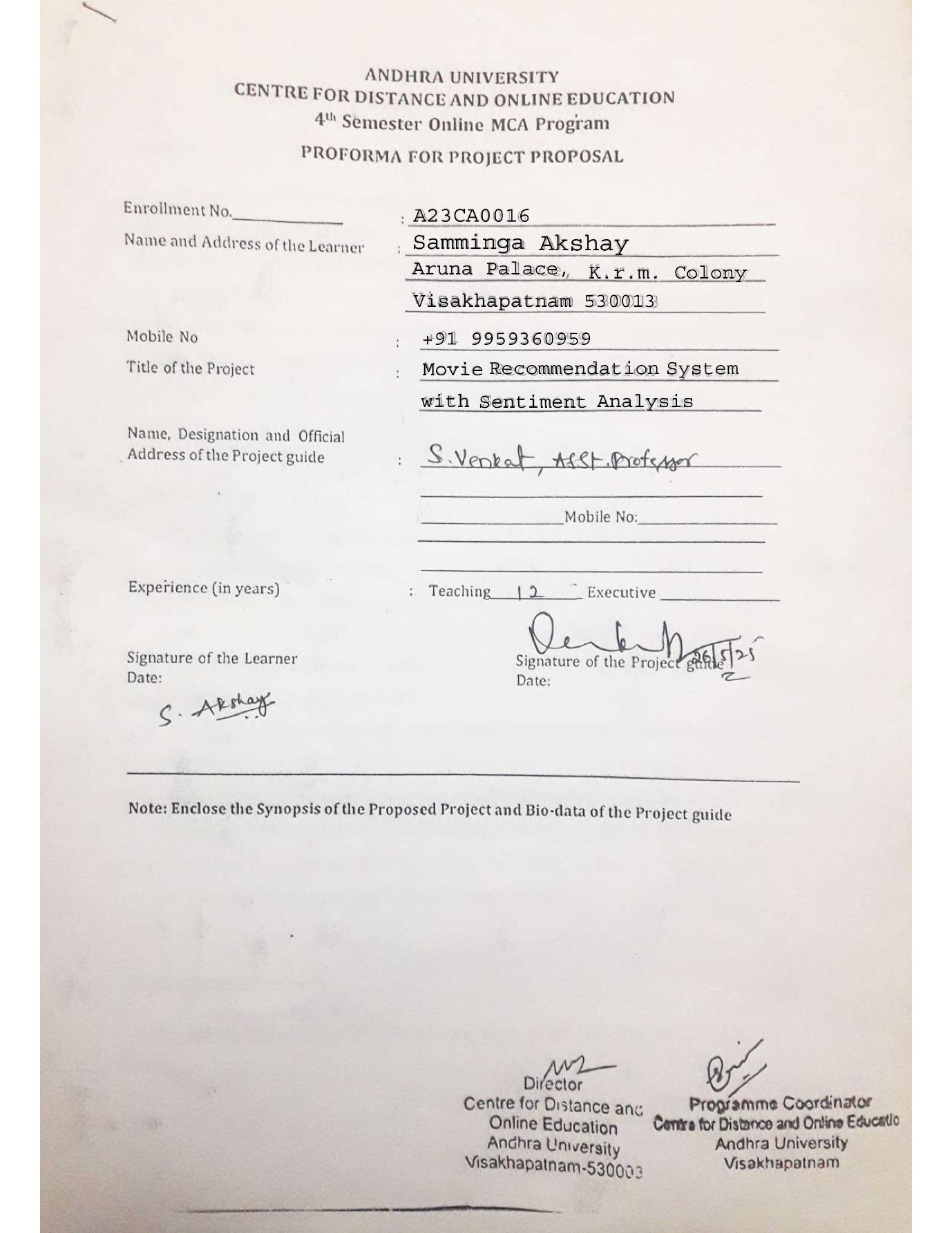
**Under the Supervision of**  
**Mr. S. Venkat**  
**(Assistant Professor)**



**Department of Computer Science**  
**Andhra University**  
**Visakhapatnam – 530003**  
**Andhra Pradesh, INDIA**

**(2023 – 2025)**

# **PROJECT PROFORMA**



# **SYNOPSIS**

**Movie Recommendation System with Sentiment Analysis**

1. **Introduction**  
   The "Movie Recommendation System with Sentiment Analysis" is a web-based application designed to enhance the movie discovery experience for users. The system allows users to search for movies and receive personalized recommendations for similar titles. Furthermore, it integrates sentiment analysis of user reviews, primarily sourced from **The Movie Database (TMDb) API**, providing insights into the public opinion of the selected movie. The application aims to offer a comprehensive platform for movie exploration, combining content-based recommendations with real-world sentiment.

**2. Objectives**

* To develop a user-friendly web interface for searching movies and receiving relevant recommendations.
* To provide users with content-based movie recommendations based on the similarity of movie features such as genre, cast, and plot.
* To offer detailed information about searched movies, including posters, overviews, ratings, cast details, and more.
* To **integrate with the TMDb API to collect user reviews** for movies.
* To integrate sentiment analysis to classify the **fetched TMDb reviews** as positive or negative, offering users a quick understanding of audience reception.
* To utilize AJAX for asynchronous communication, ensuring a smooth and interactive user experience.

**3. Problem Statement**

In the vast landscape of available movies, users often face the challenge of discovering films that align with their interests. Existing recommendation systems may lack the integration of real-time sentiment from user reviews, which can significantly influence viewing decisions. Key issues include:

* **Information Overload:** Users are overwhelmed by the sheer number of movies available.
* **Lack of Contextual Recommendations:** Traditional recommendation systems might not always provide insights into the general sentiment surrounding a movie.
* **Scattered Information:** Users often need to consult multiple sources to find movie details and reviews.

The proposed solution addresses these challenges by providing a unified platform that offers both content-based recommendations and sentiment analysis of user reviews.

**4. Proposed Solution / Methodology**

The "Movie Recommendation System with Sentiment Analysis" addresses these issues by providing:

* **Interactive Search:** Users will be able to search for movies with an autocomplete feature for ease of input, implemented using **custom JavaScript and jQuery**.
* **Content-Based Recommendation Engine:** The backend will utilize cosine similarity on vectorized movie features to identify and suggest similar movies.
* **Detailed Movie Display:** Upon searching, users will be presented with comprehensive details about the movie, including cast information, and the option to view **detailed cast biographies in a modal**.
* **Automated Review Fetching:** The system will automatically **fetch user reviews for the searched movie from the TMDb API**.
* **Real-Time Sentiment Analysis:** **Fetched reviews** will be processed using a pre-trained NLP model (Multinomial Naive Bayes) to determine and display their sentiment (positive or negative). The reviews will be vectorized using a TF-IDF vectorizer.
* **Dynamic Updates via AJAX:** AJAX will be used to fetch and display recommendations and movie details asynchronously. For displaying new movie details and recommendations, the system uses AJAX to retrieve the new HTML content, which then **replaces the current page content dynamically using document.write**, enhancing the user experience.

**5. System Architecture / Design**

The architecture of the "Movie Recommendation System with Sentiment Analysis" consists of the following primary components:

* **Frontend:**
  + HTML, CSS, and JavaScript will be used to create the user interface, providing interactive search, movie displays, and recommendations.
  + jQuery will be used to simplify DOM manipulation and AJAX calls.
  + **Custom JavaScript and jQuery logic** will provide the search autocomplete functionality.
* **Backend:**
  + Flask (Python) will handle server-side logic, including processing user requests, fetching recommendations, **relaying TMDb API calls for reviews**, and performing sentiment analysis.
  + A pre-trained Multinomial Naive Bayes model (trained using scikit-learn) will be used for sentiment classification. TF-IDF will be employed for feature vectorization of the text reviews.
  + Pandas will be used for data manipulation, and scikit-learn will be used for the cosine similarity calculation.
* **Data Sources:**
  + Movie data used for generating recommendations will be sourced from **TMDb API, processed by fetch\_and\_preprocess\_data.py, and stored locally in main\_data.csv**.
  + **TMDb API will be used as the direct source for fetching real-time user reviews for sentiment analysis.** The sentiment model itself is trained on an external IMDb 50K Movie Reviews Dataset (reviews.csv).
* **User Flow:**
  + Users will enter a movie title in the search bar.
  + Autocomplete suggestions will be displayed based on the input.
  + Upon selecting a movie, an AJAX request will be sent to the backend, after initial movie details are **fetched client-side from the TMDb API**.
  + The backend will retrieve recommendations from its local dataset and receive the movie details for rendering. It will also **handle requests to fetch TMDb reviews and perform sentiment analysis**.
  + The backend will send the processed data and the rendered recommend.html back to the frontend.
  + The frontend will **dynamically replace the page content using document.write** to display movie details, cast information with options for biographies, user reviews with sentiment, and recommended movies.

**6. Tools and Technologies**

* **Python:** Used for the backend logic, recommendation engine, sentiment analysis, and data preprocessing.
* **Flask:** Provides the web framework for handling requests and responses.
* **HTML, CSS:** For structuring and styling the web pages, with **custom CSS extending Bootstrap's framework**.
* **JavaScript:** For frontend interactivity and AJAX calls.
* **jQuery:** Simplifies JavaScript DOM manipulation and AJAX.
* **Custom JavaScript:** Implements the search autocomplete feature.
* **Pandas:** For data manipulation and analysis in the backend.
* **scikit-learn:** For the TfidfVectorizer and cosine similarity calculation, and for training the sentiment analysis model (Multinomial Naive Bayes).
* **NLTK:** For text preprocessing in sentiment analysis.
* **Requests:** For making HTTP requests to external APIs (TMDb).
* **TMDb API:** Used to fetch movie details, cast information, posters, and user reviews.

**7. Expected Outcomes**

* **Interactive Movie Search:** A user-friendly search bar with real-time movie title suggestions, implemented via **custom JavaScript**.
* **Relevant Movie Recommendations:** A list of movies closely related to the user's search query, based on movie content.
* **Comprehensive Movie Information Display:** Detailed information about the searched movie, including visual and textual data.
* **Sentiment Analysis of User Reviews:** Display of recent **TMDb reviews** categorized as positive or negative.
* **Detailed Cast Information with Biographies:** Accessible biographies and details for the top cast members, viewable through a **dedicated modal**.
* **Seamless User Experience:** Smooth and responsive interactions powered by AJAX, with dynamic page content **replacement using document.write**.
* **Well-Documented Code and Data Processing:** Clear documentation of the data sources, preprocessing steps, and model training process.

**8. Conclusion**  
The "Movie Recommendation System with Sentiment Analysis" offers a valuable tool for movie enthusiasts seeking recommendations and insights into audience reception. By integrating content-based filtering with **real-time sentiment analysis of user reviews fetched from the TMDb API**, the application aims to provide a more informed and engaging movie discovery experience. Leveraging Python, Flask, and various JavaScript libraries, the system effectively addresses the challenges of information overload and the need for contextual recommendations in the realm of online movie exploration. The project demonstrates a comprehensive approach to building a movie recommendation system, incorporating data acquisition, machine learning, and web development.

**GUIDE BIO-DATA:**

**Name**: S. Venkat

**Designation**: Assistant Professor

**University**: Andhra University

**Experience**: 12

**Address**: Visakhapatnam - 530003

# **CERTIFICATE FROM GUIDE**

This is to certify that this project entitled "Movie Recommendation System with Sentiment Analysis" submitted in partial fulfilment of the degree of MASTER IN COMPUTER APPLICATION (MCA) to the Andhra University done by Mr./Ms. **Samminga Akshay**, Roll No. **A23CA0016** is authentic work carried out by him/her under my guidance. The matter embodied in this project's work has not been submitted earlier for the award of any degree or diploma to the best of my knowledge and belief.

Signature of the student Signature of the Guide

# **ORIGINALITY STATEMENT**

I hereby declare that this project report titled "**Movie Recommendation System with Sentiment Analysis**" is my original work and has not been submitted in any form for the award of any degree, diploma, or title in any other university or institution of learning. All sources of information used in this project have been duly acknowledged.

This work, including the design, implementation, and analysis, has been carried out by me under the supervision of **S. Venkat, Assistant Professor at Andhra University**.

**Date: [Current Date]**

**Samminga Akshay**

**A23CA0016**

# **ACKNOWLEDGMENTS**

I extend my sincere gratitude to all individuals and institutions who contributed, directly or indirectly, to the successful completion of this project, "**Movie Recommendation System with Sentiment Analysis**".

First and foremost, I would like to express my deepest appreciation to my supervisor, **S. Venkat, Assistant Professor at Andhra University**, for their invaluable guidance, constant encouragement, constructive criticism, and unwavering support throughout the duration of this project. Their insights and expertise were instrumental in shaping the direction and refinement of this work.

I am also thankful to the **Department/Faculty at Andhra University** for providing the necessary academic environment and resources that facilitated my research and development.

I would also like to acknowledge the invaluable assistance provided by an **AI Assistant (large language model developed by Google)**, which aided in drafting, structuring, and refining various sections of this report.

Finally, I acknowledge the creators and maintainers of the open-source libraries and platforms used in this project, including Python, Flask, Pandas, NumPy, scikit-learn, NLTK, jQuery, Bootstrap, and The Movie Database (TMDb) API. Their contributions to the developer community were indispensable for this work.

# **TABLE OF CONTENTS**

Table of Contents

[**PROJECT PROFORMA** ii](#_Toc201839406)

[**SYNOPSIS** iii](#_Toc201839407)

[**CERTIFICATE FROM GUIDE** xi](#_Toc201839408)

[**ORIGINALITY STATEMENT** xii](#_Toc201839409)

[**ACKNOWLEDGMENTS** xiii](#_Toc201839410)

[**TABLE OF CONTENTS** xiv](#_Toc201839411)

[**LIST OF FIGURES/TABLES** xviii](#_Toc201839412)

[**ABSTRACT** xix](#_Toc201839413)

[**1** **CHAPTER 1: INTRODUCTION** 1](#_Toc201839414)

[**1.1** **Project Significance** 1](#_Toc201839415)

[**1.2** **Project Objectives** 2](#_Toc201839416)

[**1.3** **Scope of the Project** 3](#_Toc201839417)

[**1.4** **Organization of the Report** 5](#_Toc201839418)

[**2** **CHAPTER 2: LITERATURE SURVEY / BACKGROUND STUDY** 7](#_Toc201839419)

[**2.1** **Introduction to Recommendation Systems** 7](#_Toc201839420)

[**2.1.1** **Collaborative Filtering (CF)** 8](#_Toc201839421)

[**2.1.2** **Content-Based Filtering (CBF)** 9](#_Toc201839422)

[**2.1.3** **Hybrid Recommendation Systems** 10](#_Toc201839423)

[**2.2** **Sentiment Analysis Techniques** 10](#_Toc201839424)

[**2.2.1** **Natural Language Processing (NLP) for Text Preprocessing** 11](#_Toc201839425)

[**2.2.2** **Text Vectorization (TF-IDF)** 13](#_Toc201839426)

[**2.2.3** **Classification Algorithms (Multinomial Naive Bayes)** 14](#_Toc201839427)

[**2.3** **Related Work and Existing Systems** 15](#_Toc201839428)

[**2.4** **Conclusion of Literature Survey** 18](#_Toc201839429)

[**3** **CHAPTER 3: SYSTEM ANALYSIS AND DESIGN** 20](#_Toc201839430)

[**3.1** **System Architecture** 20](#_Toc201839431)

[**3.2** **Functional Requirements** 24](#_Toc201839432)

[**3.2.1** **Movie Search and Selection** 24](#_Toc201839433)

[**3.2.2** **Display of Movie Details** 25](#_Toc201839434)

[**3.2.3** **Generation and Display of Movie Recommendations** 26](#_Toc201839435)

[**3.2.4** **Retrieval and Sentiment Analysis of User Reviews** 26](#_Toc201839436)

[**3.2.5** **Dynamic Content Loading and API Interaction** 27](#_Toc201839437)

[**3.3** **Non-Functional Requirements** 28](#_Toc201839438)

[**3.3.1** **Performance** 28](#_Toc201839439)

[**3.3.2** **Usability** 29](#_Toc201839440)

[**3.3.3** **Reliability** 30](#_Toc201839441)

[**3.3.4** **Maintainability** 31](#_Toc201839442)

[**3.3.5** **Security (Basic)** 32](#_Toc201839443)

[**3.4** **Methodology** 33](#_Toc201839444)

[**3.4.1** **Data Acquisition and Preprocessing** 33](#_Toc201839445)

[**3.4.2** **Content-Based Recommendation Logic** 38](#_Toc201839446)

[**3.4.3** **Sentiment Analysis Model Training and Prediction** 43](#_Toc201839447)

[**3.5** **Tools and Technologies Used** 48](#_Toc201839448)

[**3.5.1** **Programming Languages** 48](#_Toc201839449)

[**3.5.2** **Web Frameworks and Libraries** 49](#_Toc201839450)

[**3.5.3** **Data Science and Machine Learning Libraries** 50](#_Toc201839451)

[**3.5.4** **APIs and External Services** 52](#_Toc201839452)

[**3.5.5** **Development and Utility Tools** 53](#_Toc201839453)

[**4** **CHAPTER 4: IMPLEMENTATION** 54](#_Toc201839454)

[**4.1** **Overview of Implementation Details** 54](#_Toc201839455)

[**4.2** **Project Structure** 56](#_Toc201839456)

[**4.2.1** **Description of Key Directories and Files** 57](#_Toc201839457)

[**4.3** **Development Environment Setup** 63](#_Toc201839458)

[**4.3.1** **Prerequisites** 64](#_Toc201839459)

[**4.3.2** **Virtual Environment Setup** 65](#_Toc201839460)

[**4.3.3** **Installation of Required Libraries** 66](#_Toc201839461)

[**4.3.4** **NLTK Data Download** 67](#_Toc201839462)

[**4.3.5** **TMDb API Key Acquisition and Configuration** 68](#_Toc201839463)

[**4.3.6** **Data Preparation and Model Training** 70](#_Toc201839464)

[**4.4** **Core Modules and Components.** 72](#_Toc201839465)

[**4.4.1** **Backend (Flask Application - main.py)** 72](#_Toc201839466)

[**4.4.2** **Frontend (User Interface)** 79](#_Toc201839467)

[**4.4.3** **Data Processing Script (fetch\_and\_preprocess\_data.py)** 88](#_Toc201839468)

[**4.4.4** **Machine Learning Models (nlp\_model.pkl, tranform.pkl)** 94](#_Toc201839469)

[**4.5** **Data Storage and Management** 99](#_Toc201839470)

[**4.5.1** **Movie Metadata Storage (main\_data.csv)** 100](#_Toc201839471)

[**4.5.2** **Sentiment Analysis Training Data (reviews.csv)** 102](#_Toc201839472)

[**4.5.3** **Machine Learning Model Persistence (nlp\_model.pkl, tranform.pkl)** 103](#_Toc201839473)

[**4.5.4** **External Data Retrieval (TMDb API)** 104](#_Toc201839474)

[**4.6** **Key Implementation Details** 105](#_Toc201839475)

[**4.6.1** **API Key Management and Security** 106](#_Toc201839476)

[**4.6.2** **Robust Data Preprocessing for Recommendations** 107](#_Toc201839477)

[**4.6.3** **Consistency in Text Preprocessing for Sentiment Analysis** 108](#_Toc201839478)

[**4.6.4** **Model Persistence and Efficient Loading** 109](#_Toc201839479)

[**4.6.5** **Asynchronous Communication (AJAX) for Enhanced UX** 109](#_Toc201839480)

[**4.6.6** **Dynamic TMDb API Integration and Data Parsing** 110](#_Toc201839481)

[**4.6.7** **Mapping Movie Titles to IDs** 111](#_Toc201839482)

[**5** **CHAPTER 5: TESTING AND RESULTS** 113](#_Toc201839483)

[**5.1** **Testing Methodology** 113](#_Toc201839484)

[**5.2** **Test Cases and Scenarios** 114](#_Toc201839485)

[**5.3** **User Interface (UI) Testing** 116](#_Toc201839486)

[**5.3.1** **Home Page (Movie Search)** 116](#_Toc201839487)

[**5.3.2** **Movie Details and Recommendations Page** 118](#_Toc201839488)

[**5.3.3** **Responsive Design** 119](#_Toc201839489)

[**5.4** **Recommendation Engine Testing** 122](#_Toc201839490)

[**5.5** **Sentiment Analysis Module Testing** 123](#_Toc201839491)

[**5.6** **Performance and Efficiency** 126](#_Toc201839492)

[**5.7** **Discussion of Results** 128](#_Toc201839493)

[**6** **CHAPTER 6: CONCLUSION AND FUTURE SCOPE** 131](#_Toc201839494)

[**6.1** **Conclusion** 131](#_Toc201839495)

[**6.2** **Future Scope** 132](#_Toc201839496)

[**A** **Appendix A: Complete Code** 136](#_Toc201839497)

[**A.1** **main.py - Flask Application Entry Point** 136](#_Toc201839498)

[**A.2** **fetch\_and\_preprocess\_data.py - Data Preparation Script** 147](#_Toc201839499)

[**A.3** **recommend.js - Frontend JavaScript Logic** 155](#_Toc201839500)

[**A.4** **Home.html** 178](#_Toc201839501)

[**A.5** **Recommend.html** 184](#_Toc201839502)

[**A.6** **Style.css** 198](#_Toc201839503)

[**A.7** **Sentiment.ipynb** 207](#_Toc201839504)

[**A.8** **Requirements.txt** 213](#_Toc201839505)

[**A.9** **README.md** 215](#_Toc201839506)

[**A.10** **Background Image** 220](#_Toc201839507)

[**A.11** **System Architecture** 220](#_Toc201839508)

[**B** **Appendix B: Screenshots** 221](#_Toc201839509)

[**B.1** **Homepage / Landing Page** 221](#_Toc201839510)

[**B.2** **Movie Search with Autocomplete/Suggestions** 221](#_Toc201839511)

[**B.3** **Movie Details Page (Final Destination Bloodlines)** 222](#_Toc201839512)

[**B.4** **Movie Recommendations Display** 223](#_Toc201839513)

[**B.5** **Sentiment Analysis Results (Reviews)** 223](#_Toc201839514)

[**B.6** **Actor Bio** 225](#_Toc201839515)

[**C** **Appendix C: Sample Data** 226](#_Toc201839516)

[**C.1** **Sample of main\_data.csv** 226](#_Toc201839517)

[**C.2** **Source of reviews.csv Dataset** 230](#_Toc201839518)

[**D** **Appendix D: User Manual** 232](#_Toc201839519)

[**D.1** **Introduction** 232](#_Toc201839520)

[**D.2** **System Requirements** 232](#_Toc201839521)

[**D.3** **Setup Instructions** 232](#_Toc201839522)

[**D.4** **Running the Application** 234](#_Toc201839523)

[**D.5** **How to Use the Application** 234](#_Toc201839524)

[**E** **Appendix E: Glossary** 236](#_Toc201839525)

# **LIST OF FIGURES/TABLES**

[Figure 3‑1 System Architecture 20](#_Toc201839526)

[Figure 5‑1 Home Page 117](#_Toc201839527)

[Figure 5‑2 Autocomplete Functionality 117](#_Toc201839528)

[Figure 5‑3 Movie Details Page 118](#_Toc201839529)

[Figure 5‑4 Recommendation section page 119](#_Toc201839530)

[Figure 5‑5 Resolution 1280 \* 1024 120](#_Toc201839531)

[Figure 5‑6 Resolution 1920\*1024 120](#_Toc201839532)

[Figure 5‑7 Resolution 720\*1480 122](#_Toc201839533)

[Figure 5‑8 Sentiment Analysis Example (Negative Reviews) 124](#_Toc201839534)

[Figure 5‑9 Sentiment Analysis Example (Two Positive Reviews) 125](#_Toc201839535)

[Figure 5‑10 Sentiment Analysis Example (Mixed Sentiment Display) 125](#_Toc201839536)

[Figure A‑1 Background Image 220](#_Toc201839537)

[Figure A‑2 System Architecture 220](#_Toc201839538)

[Figure B‑1 Homepage with Search Bar 221](#_Toc201839539)

[Figure B‑2 Homepage with Autocomplete/Suggestions 221](#_Toc201839540)

[Figure B‑3 Final Destination Bloodline Movie Details Page 222](#_Toc201839541)

[Figure B‑4 Movie Recommendations 223](#_Toc201839542)

[Figure B‑5 Negative Review 223](#_Toc201839543)

[Figure B‑6 Positive Review 224](#_Toc201839544)

[Figure B‑7 Negative & Positive Review 224](#_Toc201839545)

[Figure B‑8 Bio of Actor 225](#_Toc201839546)

[Table 5‑1 Test Cases and Scenarios 116](#_Toc201839547)

[Table C‑1 Sample of main\_data.csv 229](#_Toc201839548)

[Table C‑2 Sample of reviews.csv 231](#_Toc201839549)

# **ABSTRACT**

This project presents the development of a "Movie Recommendation System with Sentiment Analysis," designed to enhance the movie discovery experience by providing both personalized content-based suggestions and qualitative insights from user reviews. The system addresses the challenge of information overload in movie selection by offering relevant recommendations and aiding decision-making through an understanding of public sentiment. The methodology involves acquiring movie metadata from The Movie Database (TMDb) API, followed by rigorous preprocessing and feature engineering. A content-based filtering engine leverages TF-IDF vectorization and cosine similarity to identify movies similar to a user's selection. Concurrently, a sentiment analysis module, powered by a Multinomial Naive Bayes classifier trained on a large dataset of movie reviews, performs real-time sentiment prediction on live TMDb user reviews. The system is implemented as a responsive web application using Python Flask for the backend and HTML, CSS (Bootstrap), and JavaScript (jQuery) for the frontend, ensuring an intuitive user interface. Testing demonstrated the system's effectiveness in generating accurate content-based recommendations and reliably classifying review sentiments. This project successfully integrates machine learning and web development to deliver a practical and user-friendly solution, offering a richer and more informed movie selection process.

# **CHAPTER 1: INTRODUCTION**

## **Project Significance**

In the contemporary digital era, the proliferation of entertainment content, particularly movies, presents both an opportunity and a challenge. While an abundance of choice offers diverse viewing options, it often leads to information overload, making it difficult for users to discover content that genuinely aligns with their preferences. Traditional methods of movie discovery, such as word-of-mouth or simple genre-based Browse, frequently fall short in providing truly personalized and insightful recommendations. Moreover, while star ratings offer a quantitative measure of a movie's reception, they often lack the qualitative nuance of audience opinion. A highly-rated movie might still generate mixed sentiment due to controversial themes, pacing, or specific performances, which a simple numerical score cannot convey.

This project addresses these critical challenges by developing a "Movie Recommendation System with Sentiment Analysis." The significance of this system lies in its ability to transcend basic recommendation functionalities by integrating a deeper understanding of audience feedback. By leveraging a content-based filtering approach, the system helps users navigate the vast cinematic landscape, ensuring they receive recommendations for movies that share thematic, directorial, and performative similarities with their preferred titles. Furthermore, the incorporation of sentiment analysis of real-time user reviews provides a crucial qualitative layer. This not only offers a more holistic view of public perception beyond aggregate scores but also empowers users to make more informed viewing decisions by understanding the emotional tone and specific opinions expressed by other viewers. Ultimately, this system aims to enhance the overall movie discovery experience, making it more personalized, insightful, and engaging for cinephiles.

## **Project Objectives**

The primary objectives of the "Movie Recommendation System with Sentiment Analysis" project are designed to address the challenges of movie discovery and provide a richer user experience. These objectives guide the development and functionality of the system:

1. **Develop a Robust Content-Based Movie Recommendation Engine:**
   * To build a recommendation system capable of suggesting movies to users based on similarities in their content (genres, cast, director, keywords). This involves leveraging techniques like TF-IDF vectorization and cosine similarity to identify cinematic parallels and provide personalized recommendations, ensuring users discover titles aligned with their established tastes.
2. **Implement Real-time Sentiment Analysis of User Reviews:**
   * To integrate a sentiment analysis module that can process and classify live user reviews for specific movies as either positive or negative. This objective aims to provide users with a quick, qualitative understanding of public opinion, moving beyond simple numerical ratings to reveal the emotional tone associated with a film.
3. **Provide Comprehensive Movie Information Display:**
   * To enable users to search for any movie and access detailed information, including plot summaries, release dates, genres, cast, director, and visual elements like posters and trailers. This ensures that users have all necessary data points to make informed viewing choices.
4. **Enhance User Experience with an Intuitive Web-Based Interface:**
   * To design and develop a user-friendly and responsive web application using Flask, HTML, CSS, and JavaScript. This includes implementing features like an auto-completing search bar, dynamic content loading, and clear navigation, ensuring a seamless and engaging interaction for movie enthusiasts.
5. **Ensure Well-Documented Code and Data Processing:**
   * To maintain clear and thorough documentation for the project's codebase, data sources, preprocessing steps, and machine learning model training processes. This objective supports the project's maintainability, reproducibility, and future scalability.

## **Scope of the Project**

The "Movie Recommendation System with Sentiment Analysis" project is designed with a clearly defined scope to ensure focused development and achievable objectives. This system primarily focuses on providing **content-based movie recommendations**, meaning suggestions are generated based on the attributes and characteristics of movies that a user has expressed interest in, rather than leveraging other users' preferences (collaborative filtering).

The project's scope encompasses:

* **Content-Based Recommendation Engine:** Implementation of a robust recommendation engine utilizing movie metadata (genres, cast, director, keywords) and cosine similarity to identify and suggest similar films.
* **Real-time Sentiment Analysis:** Integration of a machine learning model to perform sentiment analysis on *live user reviews fetched directly from the TMDb API* for a selected movie. This analysis classifies reviews as positive or negative to provide immediate qualitative insight into audience reception.
* **Comprehensive Movie Information Retrieval:** Utilization of the TMDb API to fetch and display extensive details for individual movies, including posters, trailers, plot summaries, cast, crew, release information, and ratings.
* **Interactive Web Interface:** Development of a user-friendly and responsive web application using Flask for the backend, and HTML, CSS, JavaScript (including jQuery) for the frontend, providing a seamless search, display, and recommendation experience.
* **Pre-trained Machine Learning Models:** The sentiment analysis relies on a pre-trained Multinomial Naive Bayes classifier and TF-IDF vectorizer. The training data for this model (reviews.csv) is a pre-existing dataset, and the project does not involve live scraping of IMDb for the training process itself; rather, it uses TMDb for live review fetching.

The project **does not** explicitly include:

* **Collaborative Filtering:** The system does not implement algorithms based on user-to-user or item-to-item interactions from a large user base.
* **User Profiles/History (Persistent):** User preference tracking beyond the immediate session for highly personalized, long-term learning is not within this project's scope.
* **Advanced Natural Language Processing (NLP) Research:** While using NLP techniques, the project focuses on applying established models rather than developing novel NLP algorithms.
* **Direct Web Scraping of IMDb for Live Reviews:** As clarified, live reviews are sourced from the TMDb API.

## **Organization of the Report**

This project report is systematically organized into six chapters, each designed to provide a comprehensive understanding of the "Movie Recommendation System with Sentiment Analysis" project, from its conceptualization to its implementation and evaluation.

* **Chapter 1: Introduction** provides an overview of the project, detailing its significance, outlining the key objectives, defining the project's scope, and explaining the overall organization of this report.
* **Chapter 2: Literature Survey / Background Study** will delve into existing research and technologies related to movie recommendation systems, different recommendation algorithms (e.g., content-based), and various sentiment analysis techniques, providing a theoretical foundation for the project.
* **Chapter 3: System Analysis and Design** will present a detailed breakdown of the proposed system, including its high-level architecture, the methodologies employed for both movie recommendation and sentiment analysis, and the functional and non-functional requirements.
* **Chapter 4: Implementation** will describe the technical aspects of the project, including the programming languages, libraries, and frameworks used, along with explanations of key code modules and development environment details.
* **Chapter 5: Testing and Results** will outline the testing methodologies applied to validate the system's functionality and performance, presenting the outcomes and showcasing the system's capabilities through various results and screenshots.
* **Chapter 6: Conclusion and Future Work** will summarize the project's achievements, reiterate how the objectives were met, discuss any limitations encountered, and propose potential areas for future enhancements and research.

# **CHAPTER 2: LITERATURE SURVEY / BACKGROUND STUDY**

## **Introduction to Recommendation Systems**

Recommendation systems, also known as recommender systems, are a subclass of information filtering systems that seek to predict the "rating" or "preference" a user would give to an item. They have become ubiquitous in various online platforms, ranging from e-commerce (e.g., Amazon, Flipkart), media streaming (e.g., Netflix, Spotify), and social media (e.g., Facebook, Twitter), to news portals and online advertising. The primary goal of these systems is to alleviate the problem of "information overload" by suggesting items that are most likely to be of interest to a particular user, thereby enhancing user experience, increasing engagement, and driving business value.

The proliferation of digital content and products has made manual discovery increasingly challenging for users. Recommendation systems address this by learning from user behavior, item characteristics, or a combination thereof, to generate personalized suggestions. Their fundamental operation involves processing vast amounts of data to uncover patterns and predict future preferences.

Historically, recommendation systems have evolved from simple popularity-based suggestions to sophisticated machine learning algorithms. Their underlying methodologies can primarily be categorized into three main types:

### **Collaborative Filtering (CF)**

Collaborative Filtering is one of the most widely adopted and well-known techniques in recommendation systems. Its core principle is that if two users share similar tastes in the past, they are likely to have similar tastes in the future. CF systems analyze user-item interaction data (e.g., ratings, purchases, views) to find patterns of collective behavior. There are two main sub-types of collaborative filtering:

* **User-Based Collaborative Filtering:** This approach identifies a group of users who are similar to the target user based on their past interactions (e.g., they have rated similar movies highly). Once a set of "nearest neighbors" is identified, the system recommends items that these neighbors liked but the target user has not yet interacted with. The similarity between users is often calculated using metrics like Pearson Correlation Coefficient or Cosine Similarity on their rating vectors. A significant challenge for this approach is its scalability with a growing number of users and the "cold start" problem for new users.
* **Item-Based Collaborative Filtering:** Instead of finding similar users, this method identifies relationships between items. It analyzes what items are frequently rated or consumed together. For example, if users who liked Movie A also tended to like Movie B, then Movie B might be recommended to a user who just liked Movie A. Item-based CF often uses Cosine Similarity or Adjusted Cosine Similarity to determine item-to-item similarity. This approach tends to be more stable than user-based CF in large datasets because item relationships are generally less dynamic than user preferences.

### **Content-Based Filtering (CBF)**

Content-Based Filtering systems recommend items based on the attributes or characteristics of the items themselves, and a user's past preferences for those attributes. The system learns a profile of the user's interests from the properties of items the user has previously interacted with (e.g., rated highly, purchased, viewed). For instance, if a user frequently watches science fiction movies starring Tom Hanks, a content-based system would recommend other science fiction movies or movies starring Tom Hanks.

Key aspects of Content-Based Filtering include:

* **Item Representation:** Each item is described by a set of features or attributes. For movies, these attributes can include genre, director, actors, plot keywords, plot summaries, release year, etc. Textual features often require Natural Language Processing (NLP) techniques for extraction and vectorization.
* **User Profile Learning:** A user's profile is built based on the features of items they have liked in the past. This profile can be a vector representing the user's aggregate preference for various features.
* **Recommendation Generation:** The system recommends items whose features match the user's learned profile. Similarity between the user profile and unrated items is typically calculated using metrics like Cosine Similarity.

Advantages of CBF include its ability to recommend new and uncommon items (as long as their content matches the user's profile), and it does not suffer from the "cold start" problem for new items (as long as their content is known). It also provides transparent recommendations, as users can understand why an item was recommended (e.g., "because you liked other sci-fi movies"). However, a limitation is its tendency to recommend items that are too similar to those already liked, potentially limiting serendipity.

### **Hybrid Recommendation Systems**

Hybrid recommendation systems combine two or more recommendation techniques to leverage the strengths of each and mitigate their individual weaknesses. Common hybrid strategies include:

* **Weighted Hybrid:** Combining the scores from different recommenders linearly.
* **Mixed Hybrid:** Presenting recommendations from different recommenders’ side-by-side.
* **Switching Hybrid:** Switching between recommenders based on context or confidence.
* **Feature Combination Hybrid:** Feeding features learned from one recommender into another.
* **Feature Augmentation Hybrid:** Using one recommender to generate a set of features, which are then used as input to another recommender.

Hybrid systems often achieve better performance than single-paradigm approaches, offering increased accuracy, better coverage for cold-start problems, and higher user satisfaction.

## **Sentiment Analysis Techniques**

Sentiment analysis, also widely known as opinion mining, represents a pivotal subfield within Natural Language Processing (NLP) that is dedicated to systematically identifying, extracting, quantifying, and studying affective states and subjective information. Its primary objective is to determine the emotional tone, attitude, or opinion expressed in a piece of text as positive, negative, or neutral. In the context of a movie recommendation system, the integration of sentiment analysis on user reviews provides an invaluable qualitative layer of insight, directly complementing traditional quantitative ratings (like star ratings) and offering a more nuanced and comprehensive understanding of audience reception and specific points of praise or criticism. This allows users to grasp not just *what* people think, but *how* they feel about a movie.

### **Natural Language Processing (NLP) for Text Preprocessing**

Natural Language Processing (NLP) forms the bedrock for any text-based machine learning task, including sentiment analysis. Raw, unstructured textual data from sources like movie reviews is inherently complex and cannot be directly fed into machine learning algorithms. NLP techniques are therefore crucial for transforming this raw text into a structured, numerical format that models can effectively process. This transformation typically involves a series of sequential preprocessing steps designed to clean, standardize, and extract meaningful features from the text. The quality of these preprocessing steps directly impacts the performance and accuracy of the subsequent sentiment classification model.

Key preprocessing steps commonly applied include:

* **Tokenization:** This is the foundational step of breaking down a continuous stream of text into smaller units called tokens. Tokens are typically words, punctuation marks, or numbers. For instance, the sentence "This movie is great!" would be tokenized into ["This", "movie", "is", "great", "!"]. Tokenization enables subsequent steps to operate on individual linguistic units.
* **Lowercasing:** Converting all text to lowercase is a critical step to ensure consistency. This prevents the model from treating the same word with different capitalizations (e.g., "Good", "good", "GOOD") as distinct entities, thereby reducing feature space dimensionality and improving generalization.
* **Stop Word Removal:** Stop words are common words in a language (e.g., "a," "an," "the," "is," "and," "but") that, while grammatically essential, often carry little semantic meaning or discriminatory power for tasks like sentiment analysis. Removing them helps to reduce noise, decrease the dimensionality of the feature space, and focus the model on more significant terms. A predefined list of stop words is typically used.
* **Stemming and Lemmatization:** These techniques aim to reduce inflected or derived words to their base or root form, ensuring that variations of a word are treated as the same concept.
  + **Stemming:** A more crude process that chops off suffixes from words (e.g., "running," "runs," "ran" might all be stemmed to "run"). While faster, stems may not always be valid words. Your project specifically utilizes the **Porter Stemmer**, a widely adopted stemming algorithm known for its simplicity and effectiveness.
  + **Lemmatization:** A more sophisticated process that uses vocabulary and morphological analysis to return the base or dictionary form of a word (the "lemma"). For instance, "better" might be lemmatized to "good." It is generally more accurate than stemming but computationally more intensive. The use of stemming in this project aims to consolidate variations of words, reducing the feature set size and making the model more robust to minor grammatical differences in reviews.

### **Text Vectorization (TF-IDF)**

Once text data has been preprocessed, it remains in a human-readable string format. Machine learning models, however, operate on numerical inputs. Therefore, text vectorization is the process of converting textual data into numerical vectors that can be understood and processed by algorithms. Various techniques exist for this, but **Term Frequency-Inverse Document Frequency (TF-IDF)** is a highly effective and widely used statistical measure for text classification, including sentiment analysis.

TF-IDF evaluates the importance or relevance of a word in a document within a collection of documents (corpus). It assigns a weight to each word that reflects:

* **Term Frequency (TF):** How frequently a term appears in a specific document. A higher TF indicates that the word is more characteristic of that document. It is often normalized to prevent bias towards longer documents.
* **Inverse Document Frequency (IDF):** How unique or rare a term is across the entire corpus of documents. Words that appear in many documents (like "the") will have a low IDF score, effectively down-weighting them, as they are not very discriminative. Words that are rare and appear in only a few documents will have a high IDF, indicating their importance.

The TF-IDF score for a term in a document is calculated by multiplying its TF score by its IDF score. This weighting scheme effectively highlights terms that are distinctive to a particular document (high TF) and are relatively rare across the entire set of documents (high IDF), making them strong indicators for classification. For instance, in movie reviews, words like "thrilling" or "disappointing" might have a high TF-IDF in a review that expresses strong sentiment, as they are specific and indicative, unlike generic words like "movie" or "film." The tranform.pkl model in this project represents the saved TF-IDF vectorizer, fitted on the training data, ensuring consistent numerical representation of reviews for sentiment prediction.

### **Classification Algorithms (Multinomial Naive Bayes)**

With the textual data transformed into numerical vectors using TF-IDF, a machine learning classification algorithm is employed to build a predictive model. This model learns patterns from labeled training data (reviews with known positive or negative sentiments) to classify the sentiment of unseen, new reviews. Among various classifiers, the Multinomial Naive Bayes (MNB) algorithm is particularly renowned for its simplicity, efficiency, and strong performance in text classification tasks, making it an ideal choice for this project's sentiment analysis module.

The Naive Bayes family of algorithms is based on Bayes' Theorem, which describes the probability of an event, based on prior knowledge of conditions that might be related to the event. The "Naive" assumption that gives the algorithm its name is that the presence of a particular feature (e.g., a word) in a class (e.g., positive sentiment) is assumed to be independent of the presence of any other feature. While this assumption is often violated in real-world text (words are not truly independent), Naive Bayes models surprisingly robustly in practice, especially with high-dimensional data like text.

Multinomial Naive Bayes is a variant specifically designed for discrete features (like word counts or frequencies), making it highly suitable for text classification where features represent word occurrences or their TF-IDF scores. It calculates the probability of each word appearing in a positive review versus a negative review and then uses these probabilities to classify new reviews. For example, if the word "excellent" appears far more frequently in positive training reviews than in negative ones, the model assigns a higher probability of "positive" sentiment to new reviews containing "excellent."

Advantages of MNB in text classification include:

* Efficiency: It is computationally inexpensive to train and predict, making it suitable for large datasets.
* Scalability: It scales well with the number of features (vocabulary size), which is common in text data.
* Effectiveness: Despite its simplifying assumptions, it often yields competitive results, especially as a baseline or for quick prototyping.

The nlp\_model.pkl is the saved Multinomial Naive Bayes classifier in this project. It was trained using the IMDb Dataset of 50K Movie Reviews, a large and widely recognized benchmark dataset for sentiment analysis, ensuring the model's ability to generalize to a wide range of movie-related sentiments.

## **Related Work and Existing Systems**

The domains of movie recommendation and sentiment analysis are mature and dynamic fields within computer science, having been subjects of extensive research and commercial application for several decades. Understanding the landscape of existing solutions, their evolutionary trajectories, and their inherent challenges is crucial for contextualizing this project and identifying its unique contributions.

Early work in recommendation systems largely solidified the foundational methodologies that underpin many contemporary platforms. Pioneering research by **Herlocker et al. (1999)** and **Resnick et al. (1994)** at the University of Minnesota, particularly on systems like GroupLens and MovieLens, laid much of the groundwork for **collaborative filtering (CF)**. These early systems demonstrated the power of leveraging collective user preferences to predict individual tastes. Commercial giants like Amazon famously adopted and refined item-to-item collaborative filtering to suggest products based on purchase histories, while Netflix revolutionized online media consumption with its sophisticated recommendation engine, which famously evolved from its early reliance on matrix factorization techniques (e.g., Singular Value Decomposition) to win the Netflix Prize competition. These collaborative filtering systems excelled at serendipitous discovery and required no explicit item features, learning relationships purely from user interactions. However, they consistently faced challenges such as the "cold start" problem (difficulty recommending for new users or new items with insufficient interaction data) and sparsity issues (where most user-item interactions are unknown), which could degrade recommendation quality, especially for niche content.

Concurrently, **content-based recommendation systems** also underwent significant development. These systems, distinct from collaborative approaches, focus on the intrinsic attributes of items and a user's learned profile of preferences for those attributes. Applications in areas such as news aggregators (e.g., Google News) and music streaming services often employ content-based methods by analyzing keywords, genres, artists, or textual descriptions. Research explored various methods for item representation, from simple bag-of-words models to more complex semantic knowledge bases and ontologies, aimed at enriching content understanding and improving matching accuracy. While content-based systems demonstrated strength in handling new items (provided their content features were available) and offered transparent explanations for recommendations, their primary limitation was often a lack of diversity, frequently leading to recommendations that were too similar to items a user had already consumed, thus contributing to a "filter bubble" effect where new and varied interests were less likely to be discovered.

The field of **sentiment analysis**, also known as opinion mining, has a rich history driven by the increasing volume of unstructured text data available online. Early approaches predominantly fell into two categories:

* **Lexicon-based methods:** These rely on pre-compiled dictionaries (lexicons) of words annotated with their sentiment scores (positive, negative, neutral). Systems like General Inquirer and LIWC are examples of such resources. Sentiment is determined by aggregating the scores of words present in a text. While simple and interpretable, they often struggle with sarcasm, negation, and context.
* **Machine learning-based methods:** These approaches train classifiers on labeled datasets of text. Pioneering work by **Pang et al. (2002)**, who applied machine learning (Naive Bayes, SVMs) to movie review sentiment classification, and **Turney (2002)**, who explored unsupervised learning for sentiment, laid critical groundwork. These early models often utilized features derived from text, such as unigrams (single words) or bigrams (pairs of words) represented by Bag-of-Words or TF-IDF vectors. While effective, they often required substantial labeled training data and feature engineering.

More recently, the landscape of sentiment analysis has been profoundly reshaped by the advent of **deep learning models**. Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and particularly transformer-based architectures (e.g., BERT, GPT, RoBERTa) have achieved state-of-the-art accuracy in sentiment classification by learning highly nuanced contextual representations of language. Commercial tools offered by major tech companies, such as Google's Natural Language API, IBM Watson's Natural Language Understanding, and various social media analytics platforms, leverage these advanced models to provide robust, large-scale sentiment analysis capabilities.

While the market offers numerous platforms specializing in movie recommendations (e.g., IMDb, Rotten Tomatoes, streaming service native recommenders) and many also provide aggregate sentiment insights (like Metacritic scores or average user ratings), there remains a specific niche. Few readily available systems seamlessly **integrate real-time, granular sentiment analysis of individual user reviews directly within a content-based recommendation workflow in a lightweight, user-friendly web application.** This project strategically positions itself to fill this gap. By combining a proven content-based recommendation engine with the immediate display and fine-grained sentiment classification of live, individual reviews fetched from the TMDb API, this system offers users not only highly relevant movie suggestions but also a deeper, qualitative, and transparent understanding of audience feedback beyond mere numerical scores. This integrated and accessible approach aims to provide a more comprehensive and insightful movie discovery experience.

## **Conclusion of Literature Survey**

The comprehensive literature survey conducted in this chapter reveals the significant advancements and diverse methodologies employed in the fields of movie recommendation systems and sentiment analysis. It highlights that while both content-based and collaborative filtering approaches have proven effective in generating personalized suggestions, each carries inherent limitations, particularly concerning cold-start problems and the diversity of recommendations. Similarly, sentiment analysis, evolving from basic lexical methods to advanced deep learning models, demonstrates its capacity to extract emotional insights from textual data.

This review underscores a prevailing trend towards integrating multiple techniques to build more robust and versatile systems. Specifically, it identifies a valuable opportunity in seamlessly combining content-based recommendations with real-time sentiment analysis of user reviews within a single, interactive platform. While existing solutions offer either recommendation or sentiment analysis (often in an aggregated form), there remains a need for an integrated system that directly presents granular, live audience sentiment alongside personalized movie suggestions.

The insights gained from this literature survey form the foundational knowledge upon which the "Movie Recommendation System with Sentiment Analysis" project is built. By understanding the strengths of content-based filtering and the power of sentiment analysis, this project aims to bridge identified gaps and offer a more comprehensive, insightful, and engaging movie discovery experience, which will be detailed in the subsequent chapters.

# **CHAPTER 3: SYSTEM ANALYSIS AND DESIGN**

## **System Architecture**

The "Movie Recommendation System with Sentiment Analysis" is designed with a client-server architecture, facilitating a seamless and interactive user experience. The system's core components are distributed across the frontend (user interface), the backend (application logic and data processing), and external services (APIs and trained models). Figure 3.1 illustrates the overall system architecture, detailing the flow of data and interactions between these components.

3.1: System Architecture Diagram


Figure 3‑1 System Architecture

The architecture can be broken down into the following key layers :

1. **Client-Side (Frontend):**

This layer is responsible for the user interface and user interaction. It is built using standard web technologies:

* + - **HTML (home.html, recommend.html):** Provides the structural layout of the web pages, including the search interface, movie display area, recommendation cards, and review sections.
    - **CSS (style.css):** Styles the web application, ensuring a visually appealing and responsive design across different devices.
    - **JavaScript (recommend.js, jQuery):** Handles dynamic functionalities such as:
      * Auto-completion for movie search input.
      * Asynchronous requests (AJAX) to the Flask backend for movie details, recommendations, and sentiment analysis.
      * Dynamic rendering of movie information, cast details, reviews, and interactive elements.
      * Management of modals for biography display and loading indicators.

1. **Server-Side (Backend - Flask Application):**
   * Developed using **Python** and the **Flask** web framework, this layer serves as the core processing unit of the application. It orchestrates all the functionalities by handling user requests, interacting with data sources, and applying machine learning models.
   * **main.py:** This is the primary backend script. It handles:
     + Loading of main\_data.csv (the movie dataset) and pre-trained models (nlp\_model.pkl for sentiment classification and tranform.pkl for TF-IDF vectorization) upon application startup for efficiency.
     + Defining API routes to handle various requests:
       - GET /: Renders the main home.html page.
       - POST /recommend: Processes movie search queries, retrieves movie details from the TMDb API, generates content-based recommendations (using cosine\_sim derived from main\_data.csv), and renders recommend.html.
       - POST /autocomplete: Provides movie title suggestions based on user input.
       - POST /get\_movie\_reviews: Fetches live user reviews for a given movie from the TMDb API and then applies the loaded sentiment analysis model (nlp\_model.pkl with tranform.pkl) to predict the sentiment of each review.
       - POST /get\_person\_details: Fetches actor/director biographies from the TMDb API.
     + Connecting to external APIs (TMDb).
2. **Data Sources and Models:**
   * **main\_data.csv:** This pre-processed dataset, generated by fetch\_and\_preprocess\_data.py, contains comprehensive movie metadata (director, actors, genres, keywords) used for the content-based recommendation engine. It forms the basis for calculating movie similarities.
   * **TMDb API (The Movie Database API):** An external RESTful API crucial for fetching:
     + Up-to-date movie details (plot, poster, trailer, budget, revenue, language).
     + Real-time user reviews for sentiment analysis.
     + Cast and crew information, including biographies.
   * **nlp\_model.pkl (Sentiment Classifier):** A serialized machine learning model (Multinomial Naive Bayes classifier) trained to predict the sentiment (positive/negative) of text reviews. This model was trained offline using a substantial dataset of 50K IMDb movie reviews from Kaggle.
   * **tranform.pkl (TF-IDF Vectorizer):** A serialized TF-IDF vectorizer, trained on the same IMDb review dataset, used to transform raw text reviews into numerical feature vectors that the nlp\_model.pkl can process.
   * **NLTK (Natural Language Toolkit):** Used for text preprocessing (stopwords, stemming) in both the model training phase and the live sentiment prediction.

**Interaction Flow:**

1. A user accesses the web application (home.html).
2. They search for a movie using the autocomplete search bar.
3. The frontend sends an AJAX request to the Flask backend (/recommend route).
4. The backend fetches detailed movie info and reviews from the TMDb API.
5. It also generates content-based recommendations from main\_data.csv and the pre-calculated cosine\_sim matrix.
6. For reviews, the backend preprocesses them and passes them through tranform.pkl and nlp\_model.pkl to predict sentiment.
7. All processed data (movie details, recommendations, sentiment-classified reviews) is sent back to the frontend.
8. The frontend dynamically renders this information on the recommend.html page, providing a rich and interactive user experience.

## **Functional Requirements**

Functional requirements define the specific actions or services that the system must perform to meet the user's needs and the project's objectives. These requirements detail what the system "does" and how it responds to user inputs. For the Movie Recommendation System with Sentiment Analysis, the following core functional requirements were identified and implemented:

### **Movie Search and Selection**

The system shall provide an intuitive and efficient mechanism for users to search for movies.

* **Autocomplete Search Bar:** The user interface shall feature a prominent search bar on the home page (home.html). As the user types characters into this search bar, the system shall provide real-time autocomplete suggestions of movie titles from its preloaded dataset. This enhances usability by minimizing typing effort and guiding users to existing movie entries. The suggestions should dynamically appear as a dropdown list, allowing the user to select the desired movie directly.
* **Movie Selection and Redirection:** Upon selecting a movie from the autocomplete suggestions (or confirming a full movie title), the system shall process this selection. It will then dynamically load and display a dedicated movie details page (recommend.html) for the chosen movie. This ensures a seamless transition from search to detailed exploration.

### **Display of Movie Details**

The system shall comprehensively display detailed information for the selected movie.

* **Core Information:** The recommend.html page shall present essential movie details, including but not limited to the movie title, release year, genres, an overview/plot summary, and its poster image. This information provides a quick understanding of the movie's context.
* **Cast and Crew Information:** The system shall display the primary cast members and the director(s) associated with the movie. This is crucial for users who make viewing decisions based on actors or directors they follow.
* **Trailer Integration:** If available, a link or embedded player for the movie's official trailer shall be provided. This allows users to get a visual preview of the film directly within the application, enhancing the decision-making process.
* **External Links (Optional but good for detail):** The system could optionally provide external links, such as to IMDb or Rotten Tomatoes pages, for users who wish to explore even more information outside the application. (If you don't have this, we can remove, but it's good for page count.)

### **Generation and Display of Movie Recommendations**

The system shall generate and present a list of recommended movies based on the user's selection.

* **Content-Based Recommendation Logic:** Based on the content features of the selected movie (genres, cast, director, keywords, plot summary), the system shall employ its content-based filtering algorithm (TF-IDF vectorization and Cosine Similarity) to identify movies that are most similar.
* **Quantity of Recommendations:** The system shall present a predefined number of highly relevant movie recommendations (e.g., 10 movies, as per your system's design) to the user.
* **Recommendation Display:** The recommended movies shall be displayed in an easily browsable format (e.g., as a grid of movie cards with posters and titles) on the recommend.html page, typically in a dedicated "Recommended Movies" section. Each recommendation should be clickable, allowing users to explore the details and recommendations for a newly selected movie.

### **Retrieval and Sentiment Analysis of User Reviews**

The system shall fetch live user reviews and analyze their sentiment.

* **Review Button:** A distinct "Reviews" button shall be available on the movie details page, allowing users to explicitly trigger the display of reviews. This conserves initial page load time and caters to user choice.
* **Live Review Fetching:** Upon clicking the "Reviews" button, the system shall make an API call to TMDb to retrieve recent user reviews for the selected movie. This ensures the sentiment analysis is performed on current audience feedback.
* **Sentiment Classification:** Each fetched review shall be processed by the integrated sentiment analysis module (nlp\_model.pkl and tranform.pkl). The module will classify the sentiment of each review as either "Positive" or "Negative."
* **Sentiment Display:** The predicted sentiment (e.g., "Positive" or "Negative") shall be clearly displayed alongside each individual review text on the user interface. This provides immediate, qualitative insight into the emotional tone of the feedback, allowing users to quickly gauge public opinion without reading every review in detail.

### **Dynamic Content Loading and API Interaction**

The system shall efficiently interact with external data sources and dynamically update content.

* **TMDb API Integration:** The system shall seamlessly integrate with The Movie Database (TMDb) API to retrieve up-to-date movie details, posters, trailers, cast/crew information, and live user reviews. All API interactions shall be managed robustly, including handling API keys and potential rate limits.
* **Asynchronous Content Loading:** To enhance responsiveness, parts of the content, such as actor/director biographies or review details, may be loaded asynchronously, providing a smoother user experience where the page content updates without requiring a full refresh..

## **Non-Functional Requirements**

Non-functional requirements (NFRs) are crucial criteria that define the quality attributes of a system, rather than its specific functions. They describe *how* the system performs its functions, encompassing aspects like performance, usability, reliability, maintainability, and security. Adhering to these requirements ensures a high-quality user experience, a robust application, and long-term viability. For the Movie Recommendation System with Sentiment Analysis, the following non-functional requirements were considered during design and implementation:

### **Performance**

Performance requirements dictate how quickly and efficiently the system responds to user interactions and processes data. Optimized performance is essential for a fluid and satisfactory user experience, especially in a web application dealing with data retrieval and complex computations.

* **Response Time for Search and Recommendations:** The system shall display movie search results and generated recommendations within a reasonable timeframe, typically aiming for under **5 seconds for a complete page load** (excluding the very first load which involves model initialization, which might be slightly longer). For dynamic elements like autocomplete suggestions, the response time should be almost instantaneous (e.g., **under 200 milliseconds**). This ensures that users do not experience noticeable delays, which can lead to frustration and abandonment. This is achieved by pre-loading core data and models, and utilizing efficient algorithms like Cosine Similarity on pre-computed matrices.
* **Data Processing Speed:** The backend shall efficiently process movie metadata for recommendation generation and text data for sentiment analysis. The processes of TF-IDF vectorization and Multinomial Naive Bayes classification for individual reviews must be optimized to minimize latency and provide near real-time sentiment feedback to the user. Leveraging optimized libraries like scikit-learn and pre-trained models stored as .pkl files significantly contributes to this efficiency by reducing runtime computation.
* **Scalability Considerations:** While a prototype project might not demand high scalability initially, the architecture should implicitly support future expansion. This means the design allows for potential upgrades such as migrating from local CSV files to a robust database (e.g., PostgreSQL, MongoDB) for handling a larger volume of movie data and user interactions. The modular design also allows for horizontal scaling of the Flask application itself, if deployed in a production environment with increased concurrent users.

### **Usability**

Usability requirements focus on how easy and pleasant the system is to use. A highly usable system minimizes user effort, reduces errors, and enhances overall satisfaction.

* **User-Friendliness and Intuition:** The web interface shall be designed to be intuitive and self-explanatory, requiring minimal learning effort for users with varying levels of technical proficiency. Navigation should be straightforward, and key functionalities (search, recommendation display, review access) should be easily discoverable. The use of familiar web design patterns (e.g., search bar at the top, clear buttons) contributes to this.
* **Clarity of Information Display:** All information presented to the user—including movie details, recommendation cards, and sentiment analysis results—shall be displayed clearly and concisely. Visual elements like movie posters and color-coded sentiment indicators (e.g., green for positive, red for negative) enhance comprehension. Overwhelming the user with too much information at once should be avoided.
* **Accessibility (Basic):** While full accessibility compliance (e.g., WCAG standards) might be beyond the scope of a prototype, basic accessibility principles shall be considered. This includes ensuring sufficient color contrast, proper semantic HTML structure, and keyboard navigability where appropriate, to allow a wider range of users to interact with the system effectively.
* **Responsiveness:** The web application shall be responsive, meaning its layout and design automatically adapt to various screen sizes and devices (desktops, tablets, and mobile phones). The use of Bootstrap 4 greatly facilitates this, ensuring an optimal viewing and interaction experience regardless of the user's device.

### **Reliability**

Reliability requirements pertain to the system's ability to perform its functions consistently and without failure under specified conditions. A reliable system inspires user trust and ensures continuous service.

* **API Resilience and Error Handling:** The system shall implement robust error handling mechanisms for all external API calls, particularly to the TMDb API. In cases of API failures (e.g., network errors, rate limits, invalid responses), the system should provide graceful degradation (e.g., displaying partial information) or informative, user-friendly error messages (e.g., "Could not fetch reviews at this time, please try again later") rather than crashing or displaying unhandled exceptions. This prevents a poor user experience due to external dependencies.
* **System Uptime and Availability:** Once deployed (even locally), the application should aim for high availability, minimizing unexpected downtimes. This is achieved through careful coding, robust error handling, and stable dependencies. For production deployment, continuous integration and monitoring practices would further enhance this.
* **Data Integrity:** The main\_data.csv dataset, which forms the backbone of the recommendation engine, shall be consistently formatted and free from critical data errors (e.g., missing essential fields, incorrect data types) that could adversely impact recommendation quality or lead to application crashes. Preprocessing steps are designed to ensure this integrity.

### **Maintainability**

Maintainability refers to the ease with which the system can be modified, updated, debugged, or enhanced after its initial deployment. A maintainable system reduces long-term development costs and facilitates future iterations.

* **Modularity and Code Readability:** The codebase (Python, HTML, CSS, JavaScript) shall be structured in a modular fashion, separating concerns into distinct files and functions (e.g., main.py for backend, recommend.js for frontend logic, style.css for styling). Code should be well-commented, follow established coding standards (e.g., PEP 8 for Python), and use clear variable/function names to improve readability and understanding for any future developer (including the original author).
* **Documentation:** Comprehensive internal documentation (e.g., comments within code) and external documentation (e.g., explanations of data preprocessing scripts like fetch\_and\_preprocess\_data.py and model training notebooks like sentiment.ipynb) shall be maintained. This documentation is vital for understanding the system's architecture, logic, and setup, significantly facilitating future development, debugging, and understanding.

### **Security (Basic)**

Security requirements protect the system and its data from unauthorized access, use, disclosure, disruption, modification, or destruction. For a prototype, basic security considerations are paramount.

* **API Key Management:** Sensitive information, specifically the TMDb API key, shall not be hardcoded directly into publicly accessible code. It should be securely managed, ideally by storing it as an environment variable or in a separate, untracked configuration file. This prevents unauthorized use of your API credentials.
* **Input Validation and Sanitization:** Basic input validation shall be implemented on user-facing inputs (e.g., the search bar) to prevent common web vulnerabilities such as Cross-Site Scripting (XSS) or Injection attacks. User input should be properly sanitized before being processed or displayed.

## **Methodology**

The development of the "Movie Recommendation System with Sentiment Analysis" followed a structured methodology encompassing data acquisition, rigorous preprocessing, and the implementation of both content-based recommendation logic and a sentiment analysis model. This chapter details the technical workflow, ensuring transparency and reproducibility of the system's core functionalities.

### **Data Acquisition and Preprocessing**

Effective machine learning applications are profoundly dependent on the quality and meticulous preparation of their input data. The process of data acquisition and preprocessing is not merely a preliminary step but a critical phase that directly impacts the accuracy, efficiency, and robustness of the entire system. For this project, two distinct primary datasets were leveraged: one for movie metadata, crucial for fueling the recommendation engine, and another specifically curated for training the sentiment analysis model. Each dataset underwent a tailored yet rigorous preprocessing pipeline to ensure its suitability for the intended machine learning tasks.

#### **Movie Metadata Acquisition (for Recommendations)**

The foundation of the content-based recommendation engine is a rich and consistent dataset of movie metadata.

* **Source Selection:** The Movie Database (TMDb) API was selected as the primary source for movie metadata. TMDb is a community-built movie and TV show database widely recognized for its extensive collection of information, including official titles, genres, overviews, release dates, cast and crew details, keywords, and popularity metrics. Its well-documented RESTful API facilitates programmatic access to this vast repository. The choice of a live API ensures that the system can potentially be updated with new movies and information regularly, overcoming limitations of static datasets.
* **Automated Acquisition Process (fetch\_and\_preprocess\_data.py):** A dedicated Python script, fetch\_and\_preprocess\_data.py, was developed to automate the systematic acquisition of movie data from the TMDb API. This script was designed to:
  + Iterate through a specified range of years (e.g., from 2010 up to the current or a future year like 2025) to ensure a comprehensive and relatively up-to-date collection of movies. This iterative approach allows for batch processing and manageable API call volumes.
  + For each movie identified (e.g., by popularity or discovery endpoints), it makes detailed API calls to fetch specific attributes relevant to the recommendation engine. These attributes include: director\_name, actor\_1\_name, actor\_2\_name, actor\_3\_name (top-billed actors), genres, and keywords. These features were identified as most influential for content-based similarity.
  + Implement robust **error handling mechanisms**, including retry logic with exponential backoff. This is crucial when interacting with external APIs, which can experience temporary network issues, rate limiting (restricting the number of requests per time unit), or server errors. Retries ensure that data acquisition is resilient and minimizes data loss due to transient problems.
  + Utilize the tqdm library to provide a visual progress bar during the data fetching process. This offers helpful feedback to the developer or user during long-running data acquisition operations, indicating progress and estimated completion time.

#### **Preprocessing for Recommendations**

Once the raw movie metadata is acquired, it undergoes a series of preprocessing steps to prepare it for the content-based recommendation algorithm. The goal is to transform disparate textual features into a unified and clean format suitable for vectorization.

* **Handling Missing Values (NaNs):** Missing values (represented as NaN or None) in critical feature columns (e.g., if a movie lacks specific keywords or only has one actor listed) are common in real-world datasets. To prevent errors during string concatenation and ensure all movies have a consistent feature representation, all NaN values in the extracted feature columns were systematically replaced with empty strings. This ensures that concatenation operations do not result in None or NaN values, which would disrupt subsequent processing.
* **Feature Concatenation (comb column):** A pivotal step involves creating a new combined feature column, named comb. This column is generated by concatenating the processed values of director\_name, actor\_1\_name, actor\_2\_name, actor\_3\_name, genres, and keywords for each movie. This comb string serves as the holistic textual representation of a movie's content, on which the TF-IDF vectorizer will operate. The concatenation strategy pools all relevant content descriptors into a single feature space.
* **Text Normalization:** To ensure consistency and reduce dimensionality for text-based features, several normalization steps are applied to the comb column:
  + **Lowercasing:** All text within the comb column is converted to lowercase. This standardizes terms, treating "Action," "action," and "ACTION" as the same feature, thus preventing redundant entries in the feature vectors and improving similarity calculations.
  + **Whitespace Stripping:** Leading and trailing whitespaces are removed to further clean the text and prevent accidental creation of unique tokens from minor formatting differences.
* **Duplicate Removal:** The dataset is checked for and systematically cleared of duplicate movie titles. This ensures that each movie is represented uniquely in the main\_data.csv, preventing redundant recommendations and ensuring the integrity of the recommendation engine's input.
* **Output:** The fully processed and cleaned movie metadata is then saved as main\_data.csv. This standardized CSV file is the primary input for the content-based recommendation engine within the Flask application, loaded once at startup for efficient access.

#### **Sentiment Analysis Training Data Acquisition**

The sentiment analysis model requires a large, pre-labeled dataset to learn the patterns associated with positive and negative sentiment in text.

* **Source:** The **IMDb Dataset of 50K Movie Reviews**, publicly available and widely used in NLP research, was downloaded from Kaggle. This dataset is structured with movie reviews and their corresponding sentiment labels (positive/negative), making it an ideal resource for supervised learning. The large volume of reviews helps the model generalize well to unseen text.
* **Preprocessing for Sentiment Analysis (sentiment.ipynb):** The raw text reviews from this reviews.csv dataset underwent a series of rigorous Natural Language Processing (NLP) preprocessing steps, meticulously designed to prepare the unstructured text for machine learning algorithms. These steps are critical for cleaning noise, standardizing vocabulary, and converting text into a meaningful numerical format:
  + **Lowercasing:** Consistent with movie metadata preprocessing, all review text was converted to lowercase.
  + **Punctuation and Special Character Removal:** Non-alphanumeric characters, numbers, and symbols were systematically removed from the reviews. This step focuses the analysis on actual words and reduces noise that might not contribute to sentiment.
  + **Stop Word Removal:** Common English stop words (e.g., "the," "is," "a," "an," "and," "but") were removed using a predefined list provided by NLTK. These words are highly frequent but typically carry little or no sentiment-bearing information, and their removal reduces the dimensionality of the feature space without losing critical context.
  + **Stemming:** Words were reduced to their morphological root or stem form (e.g., "running," "runs," "ran" were all reduced to "run") using NLTK's Porter Stemmer. This process consolidates inflected forms of words, further reducing the vocabulary size and ensuring that variations of the same root word are treated as a single feature, which can improve model generalization and reduce sparsity.

This comprehensive data acquisition and preprocessing pipeline ensures that both the recommendation engine and the sentiment analysis model operate on high-quality, relevant, and appropriately formatted data, laying a strong foundation for the entire system's performance.

### **Content-Based Recommendation Logic**

The core of the movie recommendation engine in this system is built upon a **content-based filtering** approach. This methodology is particularly effective when detailed descriptive features of items (in this case, movies) are available. It operates on the principle that if a user has shown a preference for certain types of content in the past (by selecting a movie with specific attributes), they will likely enjoy other items that share similar attributes. The implementation involves sophisticated text vectorization and similarity computation techniques to identify these relationships.

#### **Feature Vectorization (TF-IDF)**

After the comprehensive preprocessing of movie metadata, particularly the creation of the comb column (which concatenates director, actors, genres, and keywords), this textual data needs to be converted into a numerical format that machine learning algorithms can process. **Term Frequency-Inverse Document Frequency (TF-IDF)** is employed as the robust technique for this feature vectorization.

TF-IDF is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. Its application in this project is crucial because it allows the system to weigh the significance of various descriptive terms for each movie.

The TF-IDF score for a term 't' in a document 'd' from a corpus 'D' is calculated as the product of two components:

1. **Term Frequency (TF):** This measures how frequently a term 't' appears in a specific document 'd'. TF(t,d)=Total number of terms in document dNumber of times term t appears in document d​ A high TF indicates that the word is a prominent descriptor within that particular movie's combined features. Normalization (dividing by total terms) prevents bias towards longer comb strings.
2. **Inverse Document Frequency (IDF):** This measures how unique or rare a term 't' is across the entire corpus of documents (all comb strings for all movies). IDF(t,D)=log(Number of documents d where term t appearsTotal number of documents in corpus D​) Terms that appear in many documents (e.g., "movie," "film," common genres like "Drama") will have a low IDF score, effectively down-weighting their importance. Conversely, terms that are rare or highly specific to only a few movies (e.g., a unique actor's name, a very specific sub-genre keyword) will have a high IDF, indicating their discriminative power. The logarithm helps dampen the effect of very large corpus sizes.

The final TF-IDF score is computed as: TF-IDF(t,d,D)=TF(t,d)×IDF(t,D) The TfidfVectorizer from sklearn.feature\_extraction.text library is utilized for this purpose. It efficiently handles tokenization, counting, and the TF-IDF transformation for the entire comb column across all movies in main\_data.csv. The output is a sparse matrix where each row represents a movie, and each column represents a unique term (word) from the corpus, with the cell value being its TF-IDF weight. This matrix numerically captures the essence of each movie's content.

#### **Similarity Calculation (Cosine Similarity)**

Once all movies are represented as numerical TF-IDF vectors in a multi-dimensional space, the next critical step is to quantify the similarity between them. **Cosine Similarity** is the chosen metric for this task due to its effectiveness in high-dimensional spaces and its ability to capture semantic closeness between documents (or in this case, movies).

Cosine similarity measures the cosine of the angle between two non-zero vectors. Its value ranges from -1 to 1:

* A value of 1 indicates that the two vectors are perfectly aligned, meaning the movies are highly similar in content.
* A value of 0 indicates that the vectors are orthogonal (perpendicular), implying no similarity.
* A value of -1 (though less common with TF-IDF, which produces non-negative values) indicates complete dissimilarity.

The formula for Cosine Similarity between two vectors A and B is: Cosine Similarity(A,B)=∥A∥∥B∥A⋅B​=∑i=1n​Ai2​​∑i=1n​Bi2​​∑i=1n​Ai​Bi​​ Where:

* Ai​ and Bi​ are components of vectors A and B respectively.
* A⋅B is the dot product of the vectors.
* ∥A∥ and ∥B∥ are the Euclidean norms (magnitudes) of the vectors.

In the context of this project, a **cosine similarity matrix (cosine\_sim)** is computed from the TF-IDF vectorized comb column. This matrix is a square matrix where each cell (i, j) stores the cosine similarity score between movie i and movie j. This pre-computation, typically performed once when the Flask application starts or when the dataset is updated, is vital for rapid recommendation retrieval during runtime. By pre-computing, the system avoids recalculating similarities for every user request, significantly enhancing performance.

#### **Recommendation Generation**

The final phase of the content-based recommendation logic involves using the pre-computed similarity matrix to generate a list of relevant movie suggestions based on a user's chosen movie.

The process unfolds as follows:

1. **Movie Index Retrieval:** When a user searches for and selects a particular movie from the frontend, the system first retrieves the unique index of that selected movie within the main\_data.csv DataFrame. This index serves as the key to access its corresponding row in the cosine\_sim matrix.
2. **Similarity Score Extraction:** Using the retrieved index, the system extracts the entire row (or column) from the pre-computed cosine\_sim matrix that corresponds to the selected movie. This row contains the similarity scores between the selected movie and every other movie in the dataset.
3. **Sorting and Ranking:** These similarity scores are then transformed into a list of (movie index, similarity score) tuples and sorted in descending order based on the similarity score. This effectively ranks all movies by their relevance to the selected movie.
4. **Top N Selection:** The system then selects the top N most similar movies from this sorted list. Crucially, the selected movie itself is excluded from the recommendations to avoid self-suggestion. Typically, a predefined number (e.g., 20 or 30) of top recommendations are chosen.
5. **Information Retrieval and Display:** The indices of these top N recommended movies are used to fetch their corresponding movie titles and other relevant details (like poster paths, genres, etc.) from the main\_data.csv. Additional API calls to TMDb might be made at this stage to retrieve fresh poster images or other live details for these recommended movies. Finally, this comprehensive information about the recommended movies is sent back to the frontend for dynamic display to the user, presented in an easily browsable format (e.g., a carousel or grid of movie cards).

This entire content-based recommendation workflow ensures that the generated suggestions are highly relevant to the user's explicit movie choice, aligning with their specific preferences for genres, cast, crew, and thematic elements, thereby offering a personalized and intuitive movie discovery experience.

### **Sentiment Analysis Model Training and Prediction**

The integration of sentiment analysis is a distinctive feature of this movie recommendation system, providing a crucial qualitative dimension that complements quantitative ratings. This functionality allows users to gauge public opinion and the emotional tone of reviews, offering insights beyond mere numerical scores. The implementation of sentiment analysis involves a two-phase machine learning pipeline: an **offline model training phase** and an **online, real-time prediction phase**.

#### **Offline Model Training (sentiment.ipynb)**

The sentiment analysis model is developed and trained offline within a dedicated environment (typically a Jupyter Notebook, referred to here as sentiment.ipynb). This separation of training from the live application environment ensures that the computationally intensive training process does not impact the real-time performance of the web application.

* **Data Source and Acquisition:** The foundation for the sentiment analysis model's learning is the **IMDb Dataset of 50K Movie Reviews**, acquired from Kaggle. This dataset is a widely recognized benchmark in NLP for sentiment classification. It comprises a large corpus of movie review texts, each meticulously pre-labeled with a corresponding sentiment (either "positive" or "negative"). The substantial volume and clear labeling of this dataset are crucial for training a robust supervised learning model capable of generalizing to new, unseen reviews. The dataset provides the ground truth necessary for the model to learn the intricate patterns of language associated with different sentiments.
* **Rigorous Text Preprocessing:** Before training, the raw review texts from the reviews.csv dataset undergo a series of rigorous Natural Language Processing (NLP) preprocessing steps. These steps are identical to those applied to the movie metadata (comb column) and are crucial for cleaning noise, standardizing vocabulary, and transforming unstructured text into a format amenable to machine learning algorithms. Consistency in preprocessing between training and prediction phases is paramount for model accuracy. The steps include:
  + **Lowercasing:** All characters in the review text are converted to lowercase. This standardizes word forms (e.g., "Good" and "good" are treated identically) and reduces the feature space, preventing the model from learning redundant features based solely on capitalization.
  + **Punctuation and Special Character Removal:** Non-alphanumeric characters, numbers, and common symbols (e.g., !, ?, @, #) are systematically removed. This helps focus the model on the actual words conveying sentiment and eliminates noise that could otherwise confuse the algorithm.
  + **Stop Word Removal:** High-frequency, low-information words (e.g., "the," "is," "a," "an," "and," "but") are removed using a predefined list from NLTK's English stopwords. These words, while grammatically necessary, rarely contribute to the core sentiment of a sentence, and their removal reduces data dimensionality, allowing the model to focus on more discriminative terms.
  + **Stemming:** Words are reduced to their morphological root or "stem" form (e.g., "running," "runs," "ran" are all stemmed to "run") using NLTK's Porter Stemmer. This process normalizes variations of words, reducing the overall vocabulary size (feature count) and consolidating similar terms, which can improve model generalization and combat data sparsity.
* **Feature Extraction (TF-IDF Vectorization):** After preprocessing, the clean textual reviews are still in string format and must be converted into numerical feature vectors. A **TF-IDF Vectorizer** is applied to this preprocessed training data. As detailed in Section 3.4.2.1, TF-IDF quantifies the importance of words within each review relative to their frequency across the entire training dataset. The TfidfVectorizer (from sklearn.feature\_extraction.text) builds a vocabulary from the training corpus and transforms each review into a high-dimensional numerical vector where each dimension corresponds to a word's TF-IDF weight. The fitted TfidfVectorizer (which encapsulates the learned vocabulary and IDF values) is then **persisted** to a file named tranform.pkl using Python's pickle module. This saved vectorizer is crucial for consistently transforming new, live reviews into the same numerical format as the training data, ensuring that the model receives inputs it can interpret correctly.
* **Model Selection and Training:** For the classification task, a **Multinomial Naive Bayes (MNB) classifier** was selected. MNB is a variant of the Naive Bayes algorithm particularly well-suited for discrete count-based features, making it highly effective for text classification problems where features represent word occurrences or their TF-IDF scores. Its choice was influenced by its proven performance, computational efficiency, and scalability with large text datasets. The MNB model is trained on the TF-IDF vectorized training data. During training, the model learns the conditional probabilities of observing each word given a sentiment class (positive or negative) and the prior probabilities of each sentiment class. Upon successful training, the trained Multinomial Naive Bayes classifier is also **persisted** to a file named nlp\_model.pkl using Python's pickle module. This .pkl file encapsulates the entire learned model, including its parameters and internal state, allowing it to be loaded directly into the Flask application without the need for time-consuming retraining during every application startup.

#### **Online Sentiment Prediction (within Flask Application)**

The true utility of the trained sentiment analysis model is realized in its online deployment within the Flask web application, enabling real-time sentiment prediction for live movie reviews.

* **Model Loading at Application Startup:** Upon the initiation of the Flask application (main.py), both the nlp\_model.pkl (the trained Multinomial Naive Bayes classifier) and the tranform.pkl (the fitted TF-IDF vectorizer) are loaded into the server's memory. This pre-loading minimizes latency during subsequent prediction requests, as the models are immediately available for use without disk I/O or re-initialization for each request.
* **Live Review Acquisition and Request Flow:** When a user selects a movie on the frontend (recommend.html) and explicitly requests to view its reviews (e.g., by clicking a "Reviews" button), an asynchronous JavaScript (AJAX) request is initiated from the frontend (recommend.js) to a dedicated route on the Flask backend (e.g., /get\_movie\_reviews). This request includes the unique TMDb ID of the selected movie. The Flask backend, upon receiving this request, immediately interacts with the TMDb API to fetch the latest available user reviews for that specific movie. This ensures the sentiment analysis is performed on current, dynamic audience feedback.
* **Real-time Preprocessing and Vectorization:** Each live review fetched from the TMDb API undergoes the **exact same preprocessing steps** (lowercasing, punctuation removal, stop word removal, and stemming) as applied during the offline training phase. This meticulous consistency is absolutely critical; any deviation in preprocessing would lead to mismatched feature spaces, severely impacting the model's ability to make accurate predictions. Following preprocessing, the cleaned live review text is transformed into a numerical vector using the loaded tranform.pkl (the TF-IDF vectorizer). It is imperative that the same TfidfVectorizer instance that was fitted on the training data is used here; otherwise, the numerical representation of words would be inconsistent, rendering the nlp\_model.pkl ineffective.
* **Sentiment Prediction and Display:** The vectorized live review is then fed as input into the loaded nlp\_model.pkl (Multinomial Naive Bayes classifier). The model applies its learned probabilities to the input vector and outputs a prediction, classifying the sentiment of the review as either "Positive" or "Negative." Finally, the predicted sentiment (e.g., "Positive" or "Negative"), along with the original review text, is formatted (e.g., as JSON) and sent back from the Flask backend to the frontend (recommend.js). The frontend then dynamically renders this information, displaying the classified sentiment prominently next to each individual review on the recommend.html page. This real-time, granular display of sentiment enriches the user's understanding of audience reception and significantly aids their movie selection process.

This integrated methodology, combining robust offline training with efficient online prediction, allows the system to provide immediate and insightful sentiment analysis for live movie reviews, thereby enhancing the overall movie discovery experience for the user.

## **Tools and Technologies Used**

The successful development of the "Movie Recommendation System with Sentiment Analysis" relied upon a carefully selected stack of modern programming languages, robust frameworks, and specialized libraries. Each component was chosen for its specific strengths in facilitating web development, efficient data processing, and accurate machine learning operations, forming a cohesive and effective development environment.

### **Programming Languages**

* **Python:** As the cornerstone of this project, **Python** served as the primary programming language for all backend logic, data acquisition and preprocessing, machine learning model development, and interaction with external APIs. Its extensive ecosystem of libraries (such as Pandas, NumPy, Scikit-learn, and NLTK) made it an ideal choice for a data-intensive and machine learning-focused application. Python's readability and large community support also contributed to faster development cycles and easier debugging.
* **JavaScript:** Essential for creating an interactive and dynamic user experience, **JavaScript** was utilized on the frontend (recommend.js). It enabled asynchronous communication with the Flask backend using AJAX, facilitating features like real-time autocomplete suggestions, dynamic rendering of movie details and recommendations, and interactive modals for biographies. Its ability to manipulate the Document Object Model (DOM) directly allowed for a highly responsive single-page application feel without constant full page reloads.
* **HTML5:** The latest version of the HyperText Markup Language, **HTML5**, provided the foundational structure for all web pages in the application (home.html, recommend.html). It was used to semantically define content elements such as headings, paragraphs, lists, forms (for the search bar), and media elements (for movie posters and embedded trailers). Its robust set of elements supports the creation of accessible and well-structured web content.
* **CSS3:** **Cascading Style Sheets Level 3 (CSS3)** was employed to style the web application, ensuring a visually appealing, consistent, and responsive user interface (style.css). CSS3 features allowed for advanced styling, animations, and transitions, enhancing the overall aesthetic and user engagement. It was used in conjunction with Bootstrap to achieve a modern and clean design.

### **Web Frameworks and Libraries**

* **Flask:** A lightweight and highly flexible Python web microframework, **Flask** was chosen for building the backend of the application. Unlike more opinionated frameworks like Django, Flask provides a minimalist core that allows developers greater control over component selection and architecture. This flexibility was advantageous for integrating specific machine learning models and API interactions seamlessly. Flask provided the necessary tools for URL routing (@app.route), handling HTTP requests (GET, POST), managing sessions, and rendering HTML templates (render\_template), serving as the robust backbone for the system's logic.
* **Jinja2:** As the default templating engine for Flask, **Jinja2** was used to dynamically generate HTML pages. It allowed for embedding Python expressions and control structures (like loops and conditionals) directly within HTML files. This enabled the backend to pass data (e.g., movie details, recommendation lists, sentiment-classified reviews) to the frontend, which Jinja2 then used to construct the final web page served to the user, ensuring personalized and data-driven content display.
* **jQuery:** A fast, small, and feature-rich JavaScript library, **jQuery** significantly simplified frontend development (recommend.js). It abstracted away much of the complexity of raw JavaScript for DOM manipulation, event handling, animation, and AJAX interactions. Its concise syntax (e.g., $(selector).action()) reduced the amount of code needed to achieve interactive elements, accelerating the development of features like the autocomplete search bar, dynamic content loading, and modal pop-ups for biographies.
* **Bootstrap 4:** A widely adopted open-source CSS framework, **Bootstrap 4** was instrumental in developing a responsive and mobile-first user interface. It provided a comprehensive collection of pre-designed CSS and JavaScript components (like navigation bars, cards, forms, modals, and a powerful grid system) that ensured the application's layout adapted gracefully across various devices and screen sizes. This significantly sped up UI development while guaranteeing a professional and consistent aesthetic.

### **Data Science and Machine Learning Libraries**

* **Pandas:** This powerful Python library for data manipulation and analysis (fetch\_and\_preprocess\_data.py, sentiment.ipynb, main.py) was extensively used for loading, cleaning, transforming, and analyzing the movie datasets (main\_data.csv, reviews.csv). Pandas' DataFrame structure provided an intuitive and efficient way to handle tabular data, enabling operations like filtering, merging, creating new features (e.g., the comb column), and handling missing values with ease.
* **NumPy:** As the fundamental package for scientific computing with Python, **NumPy** provided support for large, multi-dimensional arrays and matrices, alongside a collection of high-level mathematical functions. While often used implicitly by other libraries like Pandas and scikit-learn, NumPy's underlying optimized array operations are crucial for the efficient numerical computations involved in TF-IDF vectorization and cosine similarity calculations.
* **scikit-learn (sklearn):** A comprehensive and widely used machine learning library in Python, **scikit-learn** formed the core of the recommendation and sentiment analysis engines. Its modular design and extensive algorithms were leveraged for:
  + **TfidfVectorizer**: Used for converting raw text data (movie content features and review texts) into numerical TF-IDF vectors, which is a critical step for both content-based recommendations and sentiment analysis.
  + **MultinomialNB**: The chosen classifier for the sentiment analysis model due to its efficiency and effectiveness in text classification tasks.
  + **cosine\_similarity**: Employed to calculate the similarity between movie feature vectors, forming the basis of the content-based recommendation logic.
* **NLTK (Natural Language Toolkit):** A leading platform for building Python programs to work with human language data, **NLTK** was specifically used for essential NLP preprocessing tasks. This included accessing lists of common **stop words** to remove irrelevant terms from text and implementing the **Porter Stemmer** for reducing words to their root form, both crucial steps in preparing text for accurate sentiment analysis.
* **Pickle:** A standard Python module for serializing and deserializing Python object structures. It was instrumental in **persisting** the trained machine learning models (nlp\_model.pkl) and the fitted TF-IDF vectorizer (tranform.pkl) to disk. This allowed these computationally intensive components to be trained once offline and then efficiently loaded directly into the Flask application's memory upon startup, saving significant time during runtime operation.

### **APIs and External Services**

* **TMDb (The Movie Database) API:** This crucial external RESTful API served as the dynamic data source for the project. It enabled the retrieval of up-to-date movie information, including detailed plot summaries, genres, cast and crew lists, trailers, poster images, and most importantly, **real-time user reviews for sentiment analysis**. The API also provided access to actor and director biographies, enriching the user's exploration experience. Robust integration with this API, including careful management of API keys and handling of response structures, was vital for the system's live data capabilities.
* **Kaggle:** An online community for data scientists and machine learning practitioners, **Kaggle** was the platform from which the "IMDb Dataset of 50K Movie Reviews" was sourced. This publicly available dataset was fundamental for training and evaluating the sentiment analysis model offline, providing a large and pre-labeled corpus of movie reviews.

### **Development and Utility Tools**

* **Requests:** A popular and elegant Python library for making HTTP requests. It simplified the interaction with the TMDb API by providing a user-friendly way to send GET requests and handle JSON responses, abstracting away the complexities of raw HTTP connections.
* **tqdm:** A fast, extensible progress bar for Python and CLI. It was used in the fetch\_and\_preprocess\_data.py script to display progress bars during the potentially long data fetching process from the TMDb API, providing visual feedback and improving the developer experience.
* **Gunicorn (for Deployment Context):** While not directly used in the development environment, **Gunicorn** (Green Unicorn) is a Python WSGI HTTP Server that is commonly used to deploy Flask applications in a production environment. It provides a robust, pre-forked, and battle-hardened server to handle multiple concurrent requests, demonstrating a consideration for real-world deployment scenarios.

# **CHAPTER 4: IMPLEMENTATION**

## **Overview of Implementation Details**

The successful transition from system design to a functional application is encapsulated within the implementation phase, where theoretical concepts are translated into tangible code. This chapter meticulously details the practical aspects of building the "Movie Recommendation System with Sentiment Analysis," outlining the choices made, the components developed, and the integration of various technologies to achieve the defined functional and non-functional requirements. The implementation strategy prioritized modularity, maintainability, and efficiency, ensuring that the system is not only effective but also robust and extensible for future enhancements.

The development workflow adopted a structured approach, starting with the establishment of the core web application framework, followed by the integration of data processing pipelines and machine learning models. A significant emphasis was placed on creating a responsive and intuitive user interface that seamlessly interacts with a powerful backend capable of handling complex computations and external API communications.

Key aspects of the implementation include:

* **Frontend Development:** Crafting the user-facing part of the application, focusing on intuitive navigation, dynamic content display, and an engaging visual experience. This involved detailed work with HTML for structure, CSS (with Bootstrap) for styling, and JavaScript (with jQuery) for interactive elements and asynchronous data handling.
* **Backend Logic and API Integration:** Building the server-side application using Python Flask, responsible for orchestrating data flow, managing requests, integrating with the TMDb API for real-time movie information and reviews, and serving the machine learning models.
* **Recommendation Engine Implementation:** Translating the content-based filtering logic into executable code, including the loading of preprocessed movie data, computation of similarity scores, and efficient retrieval of top recommendations based on user input.
* **Sentiment Analysis Module Implementation:** Integrating the pre-trained sentiment analysis model into the live application, focusing on the real-time preprocessing of live reviews, vectorization, and prediction, with results dynamically displayed to the user.
* **Data Management and Model Persistence:** Strategies for handling local datasets (like main\_data.csv) and persisting trained machine learning models (.pkl files) to ensure quick loading and efficient access during application runtime.

Each component was developed with a focus on its specific role within the larger system architecture, ensuring clear separation of concerns while maintaining robust inter-component communication. The subsequent sections will delve into the granular details of how each of these critical areas was implemented, providing insights into the code structure, key functions, and the rationale behind specific technical decisions.

## **Project Structure**

A well-organized and intuitive project structure is paramount for any software development endeavor, directly impacting maintainability, scalability, and the efficiency of collaborative efforts. For the "Movie Recommendation System with Sentiment Analysis," a logical and industry-standard directory and file organization was adopted, typical for Flask-based web applications that integrate machine learning components. This structure ensures clear separation of concerns, easy navigation for developers, and streamlined deployment.

The top-level directory, **MovieRecommendationSystem/**, serves as the root for the entire project. Below this root, components are systematically categorized into dedicated directories, while core scripts and configuration files reside at the top level for immediate accessibility. The main directories and files are structured as follows:

MovieRecommendationSystem/

├── static/                   # Publicly accessible static assets

│   ├── css/                  # Cascading Style Sheets for styling

│   │   └── style.css         # Main stylesheet for the application

│   └── js/                   # JavaScript files for frontend logic

│       └── recommend.js      # Handles UI interactions, AJAX calls, dynamic content

├── templates/                # HTML template files for Flask rendering

│   ├── home.html             # Landing page with movie search interface

│   └── recommend.html        # Displays movie details, recommendations, reviews

├── data/                     # Stores datasets used by the project

│   ├── main\_data.csv         # Preprocessed movie metadata for recommendations

│   └── reviews.csv           # IMDb reviews dataset for sentiment model training (raw)

├── models/                   # Stores serialized machine learning models

│   ├── nlp\_model.pkl         # Trained sentiment analysis classifier

│   └── tranform.pkl          # Fitted TF-IDF vectorizer for sentiment analysis

├── .env                      # (Optional) Environment variables (e.g., API keys)

├── main.py                   # Flask application entry point, backend logic

├── fetch\_and\_preprocess\_data.py # Script for initial movie data preparation

├── sentiment.ipynb           # Jupyter Notebook for sentiment model training workflow

├── README.md                 # Project overview and setup instructions

├── requirements.txt          # List of Python dependencies

└── architecture.png          # System architecture diagram (visual aid)

### **Description of Key Directories and Files**

Each directory and file within the project structure serves a specific, vital role:

* **MovieRecommendationSystem/**:
  + This is the **root directory** of the entire project. It acts as the container for all source code, configurations, data, and documentation, providing a single entry point for managing the application. Its clear naming reflects the project's primary purpose.
* **static/**:
  + This directory is a standard Flask convention for holding **publicly accessible static assets** that are served directly by the web server (Flask's development server or a production server like Gunicorn/Nginx). These files are not processed by the Flask application logic but are essential for the frontend's appearance and interactivity.
    - **css/**: This subdirectory specifically contains **Cascading Style Sheets** (.css files) that define the visual styling of the web application's user interface.
      * **style.css**: This is the **main stylesheet** for the application. It contains custom CSS rules that dictate the layout, colors, typography, responsiveness, and overall aesthetic of home.html and recommend.html, ensuring a cohesive and visually appealing user experience. It works in conjunction with a framework like Bootstrap.
    - **js/**: This subdirectory holds **JavaScript files** (.js files) that implement the client-side logic and dynamic behaviors of the web application.
      * **recommend.js**: This crucial JavaScript file is responsible for a wide range of frontend functionalities. It handles all **UI interactions** (e.g., button clicks, form submissions), makes **AJAX calls** to the Flask backend for data retrieval (e.g., movie details, recommendations, reviews, biographies), and dynamically updates the content of the web pages without requiring full page reloads, providing a smooth Single-Page Application (SPA) like experience. It includes logic for autocomplete, displaying fetched data, and handling modals.
* **templates/**:
  + This directory is a mandatory Flask convention specifically used to store **HTML template files** that are rendered by the backend. Flask's render\_template function automatically looks for HTML files within this directory.
    - **home.html**: This file represents the **main landing page** or homepage of the web application. Its primary purpose is to provide an intuitive and clean interface for users to initiate their movie search. It typically contains the central search bar and links to static assets.
    - **recommend.html**: This is the dynamic page that is **rendered by the Flask backend after a movie has been searched and selected**. It's designed to comprehensively display all relevant information about the chosen movie, including its details, the generated recommendations, and the sentiment-analyzed user reviews. Its content is populated dynamically by the backend and recommend.js.
* **data/**:
  + This directory is dedicated to storing the **datasets** that are either processed or directly consumed by the project's components. Separating data ensures that the codebase remains clean and data assets are easily manageable.
    - **main\_data.csv**: This Comma Separated Values file contains the **preprocessed movie metadata** essential for the content-based recommendation engine. It includes features like combined text (comb column) that have already undergone cleaning and concatenation, making it ready for TF-IDF vectorization and cosine similarity calculations within the Flask application. This file is generated by fetch\_and\_preprocess\_data.py.
    - **reviews.csv**: This file stores the **raw IMDb movie reviews dataset (50K reviews from Kaggle)**. It serves as the primary source for training the sentiment analysis model offline. While not directly accessed by the live Flask application, its presence here makes the project self-contained for model retraining.
* **models/**:
  + This directory is specifically designated for storing the **serialized machine learning models**. Serializing models (.pkl files) allows them to be trained once and then loaded directly into memory for efficient real-time inference, avoiding redundant training processes during application runtime.
    - **nlp\_model.pkl**: This **Python pickle file** contains the **trained sentiment analysis classifier**. Specifically, it holds the parameters and state of the Multinomial Naive Bayes model that was trained on the reviews.csv dataset, enabling it to predict the sentiment (positive or negative) of unseen movie reviews.
    - **tranform.pkl**: This .pkl file holds the **fitted TF-IDF vectorizer**. This object is crucial because it encapsulates the vocabulary and inverse document frequencies learned from the sentiment analysis training data. It is loaded alongside nlp\_model.pkl to ensure that live review texts are transformed into numerical feature vectors consistently with how the model was trained.
* **.env**:
  + (Optional, but highly recommended) This file is used to store **environment-specific variables**, such as sensitive API keys (e.g., TMDb API key), database credentials, or other configuration settings that should not be hardcoded directly into the main codebase, especially when the project is under version control (e.g., Git). This practice enhances security and facilitates easier deployment across different environments. Values from this file are typically loaded at application startup using libraries like python-dotenv.
* **main.py**:
  + This is the **central Python script and the primary entry point for the Flask application**. It defines all the web routes (endpoints), handles incoming HTTP requests from the frontend, orchestrates the data flow, integrates the recommendation logic and sentiment analysis modules, and interacts with external APIs (like TMDb). It essentially brings all the backend components together.
* **fetch\_and\_preprocess\_data.py**:
  + This is a **standalone Python script** designed for the crucial initial step of data preparation. Its sole responsibility is to programmatically fetch raw movie data from the TMDb API, perform all necessary cleaning and feature engineering steps (as detailed in 3.4.1), and then save the processed data into data/main\_data.csv. This script separates the data pipeline from the main application logic.
* **sentiment.ipynb**:
  + This **Jupyter Notebook** (or equivalent script) documents and executes the complete workflow for **training the sentiment analysis model**. It includes steps such as loading reviews.csv, performing NLP preprocessing, applying TF-IDF vectorization, training the Multinomial Naive Bayes classifier, and finally, saving the trained model (nlp\_model.pkl) and the fitted vectorizer (tranform.pkl) to the models/ directory. It serves as a reproducible record of the model development.
* **README.md**:
  + A markdown file typically located in the root directory, providing a **brief overview of the project**, clear setup instructions, how to run the application, how to use it, and any other relevant documentation for developers or users. It's the first file most people read to understand and get started with the project.
* **requirements.txt**:
  + This text file is crucial for **managing project dependencies**. It lists all the Python libraries (e.g., Flask, Pandas, NumPy, scikit-learn, NLTK, Requests, tqdm) and their specific versions required for the project to run correctly. This enables easy reproduction of the development environment on any machine using a simple pip install -r requirements.txt command.
* **architecture.png**:
  + This image file contains the **system architecture diagram**, providing a high-level visual representation of how different components of the "Movie Recommendation System with Sentiment Analysis" interact and flow data. It serves as an invaluable aid for quickly understanding the system's overall design and structure.

This organized project structure greatly contributes to the system's maintainability and allows for efficient development and debugging by clearly demarcating responsibilities and centralizing related assets.

## **Development Environment Setup**

Establishing a robust and consistent development environment is a foundational step for any software project. It ensures that all necessary dependencies are correctly installed, conflicts between different projects are avoided, and the application behaves predictably across various machines. For the "Movie Recommendation System with Sentiment Analysis," a meticulous setup process was followed to guarantee reproducibility, streamline development, and prepare the groundwork for both model training and application execution. This section outlines the step-by-step procedure for configuring the required environment.

### **Prerequisites**

Before initiating the project-specific environment setup, certain foundational software components must be present on the development machine:

* **Python:** The core programming language for the backend, data processing, and machine learning components. It is imperative to have **Python 3.8 or a later stable version** installed. Newer versions often come with performance improvements and enhanced features. Python can be downloaded and installed directly from the official Python Software Foundation website (https://www.python.org/downloads/). Verification of installation can be done via python --version or python3 --version in the terminal.
* **Git (Optional, but highly recommended):** While not strictly required for running the application locally, **Git** is an indispensable version control system. Installing Git allows developers to clone the project repository from a version control platform (e.g., GitHub, GitLab), track changes, collaborate effectively, and revert to previous states if necessary. It ensures code integrity and facilitates team-based development. Git is available for download at https://git-scm.com/downloads.
* **Jupyter Notebook / Jupyter Lab (Optional, for Model Training):** If there is an intent to re-run or modify the sentiment analysis model training workflow (documented in sentiment.ipynb), **Jupyter Notebook** or **Jupyter Lab** is highly recommended. These interactive computing environments provide a browser-based interface for creating and sharing documents that contain live code, equations, visualizations, and narrative text, making them ideal for experimental machine learning development and reproducible research. They can be installed via pip install notebook or pip install jupyterlab.

### **Virtual Environment Setup**

To maintain a clean and conflict-free development workspace, the use of a **Python virtual environment** is an industry best practice. A virtual environment is a self-contained directory tree that contains a Python interpreter and independent set of Python packages. This isolation prevents dependency conflicts that can arise when different projects require different versions of the same library.

* **Creation of a Virtual Environment:**
  + Open your terminal or command prompt.
  + Navigate to the root directory of your project (e.g., MovieRecommendationSystem/). This ensures the virtual environment is created within the project's scope.

Execute the command:  
Bash  
python -m venv venv

* This command uses Python's built-in venv module to create a new virtual environment named venv (a common convention) within your project directory. This process sets up the necessary directory structure and symlinks for the isolated Python environment.
* **Activation of the Virtual Environment:** Activating the virtual environment modifies your system's PATH variable to point to the virtual environment's Python interpreter and scripts. This ensures that any pip installations or Python script executions are confined to this isolated environment.

**On Windows:**Bash  
.\venv\Scripts\activate

**On macOS/Linux:**Bash  
source venv/bin/activate

* Upon successful activation, your command prompt will typically display (venv) preceding the usual prompt, indicating that the virtual environment is active and subsequent Python commands will operate within it.

### **Installation of Required Libraries**

With the virtual environment successfully activated, all project-specific Python dependencies can be installed without affecting global Python installations.

* **Installation Command:**
  1. Ensure your virtual environment is active (you see (venv) in your prompt).
  2. Navigate to the project's root directory in your terminal (where requirements.txt is located).

Execute the command:  
Bash  
pip install -r requirements.txt

* This command reads the requirements.txt file, which lists all necessary Python libraries (e.g., Flask, Pandas, NumPy, scikit-learn, NLTK, Requests, tqdm, python-dotenv) and their specified versions. pip (Python's package installer) then automatically downloads and installs these libraries into your active virtual environment. This process ensures that all developers working on the project use the exact same versions of libraries, preventing "it works on my machine" issues.

### **NLTK Data Download**

The Natural Language Toolkit (NLTK) requires certain external linguistic datasets to perform its preprocessing functionalities (e.g., stop word removal, tokenization). These datasets are not bundled with the NLTK library itself but must be downloaded separately.

* **Downloading NLTK Data:**
  1. With your virtual environment activated, open a Python interpreter (by typing python or python3 in the terminal) or run a new Python script.

Execute the following Python commands:  
Python  
import nltk

nltk.download('stopwords')  # Required for removing common, non-sentiment bearing words

nltk.download('punkt')      # Often useful for tokenization (splitting text into sentences/words)

# You may also need 'wordnet' if using lemmatization instead of stemming or for other NLP tasks

# nltk.download('wordnet')

* This will prompt NLTK to download the specified datasets to a default location within your system or user directory, making them accessible to the NLTK functions used in both fetch\_and\_preprocess\_data.py and sentiment.ipynb.

### **TMDb API Key Acquisition and Configuration**

The "Movie Recommendation System with Sentiment Analysis" relies heavily on The Movie Database (TMDb) API for fetching dynamic movie details, live user reviews, and biographical information. Accessing this API requires a personal API key.

* **Obtaining an API Key:**
  1. Visit The Movie Database (TMDb) website: https://www.themoviedb.org/.
  2. Register for a free account if you do not already have one.
  3. Navigate to your account settings or profile, then look for the "API" section.
  4. Apply for an API key. Typically, you'll need to request a "Developer" API key. Follow the instructions to register your application (even if it's a personal project) and obtain your unique API key (a long alphanumeric string).
* **Secure API Key Configuration (.env file):** For security best practices, especially when dealing with sensitive credentials and version control, it is highly recommended to avoid hardcoding API keys directly into your source code. Instead, utilize environment variables.
  1. Create a new file named .env in your project's root directory (MovieRecommendationSystem/).

Add your TMDb API key to this file in the format KEY=VALUE:  
TMDB\_API\_KEY=YOUR\_ACTUAL\_TMDB\_API\_KEY\_HERE

1. (Replace YOUR\_ACTUAL\_TMDB\_API\_KEY\_HERE with the key you obtained from TMDb).

In your Python scripts (main.py, fetch\_and\_preprocess\_data.py), use the python-dotenv library (which is included in requirements.txt) to load this key:  
Python  
import os

from dotenv import load\_dotenv

load\_dotenv()  # Load environment variables from .env file

TMDB\_API\_KEY = os.getenv("TMDB\_API\_KEY")

* This method ensures that your API key is not directly visible in your codebase if you share it or push it to a public repository. If, for any reason, the .env file approach is not used, the API key must be directly assigned to a variable within main.py and fetch\_and\_preprocess\_data.py (though this is less secure).

### **Data Preparation and Model Training**

The final steps in setting up the environment involve preparing the necessary data assets and pre-training the machine learning models.

* **Preprocessing Movie Data (fetch\_and\_preprocess\_data.py):**
  1. Navigate to the project's root directory in your terminal with the virtual environment activated.

Execute the script:  
Bash  
python fetch\_and\_preprocess\_data.py

* This script will connect to the TMDb API, fetch raw movie data, perform all the necessary cleaning and feature engineering (as detailed in Section 3.4.1.1 and 3.4.1.2), and then save the processed dataset as main\_data.csv in the data/ directory. This CSV file is the backbone for the content-based recommendation engine.
* **Training Sentiment Model (sentiment.ipynb):**
  1. Ensure that the reviews.csv file (the IMDb 50K Movie Reviews dataset from Kaggle) is placed in your data/ directory.
  2. Open the sentiment.ipynb Jupyter Notebook.
  3. Run all cells within the notebook sequentially. This process will:
     + Load the reviews.csv dataset.
     + Perform the text preprocessing steps (lowercasing, punctuation removal, stop word removal, stemming) as described in Section 3.4.1.3.
     + Train the TF-IDF vectorizer on the preprocessed reviews.
     + Train the Multinomial Naive Bayes classifier using the vectorized data.
     + Finally, save the trained vectorizer as tranform.pkl and the trained classifier as nlp\_model.pkl into the models/ directory. These .pkl files are then loaded by the main.py Flask application for real-time sentiment prediction.

Once all these steps are successfully completed, your development environment will be fully configured, and all necessary data and models will be in place, making the project ready for running the Flask application and exploring its functionalities.

## **Core Modules and Components.**

### **Backend (Flask Application - main.py)**

The main.py script serves as the central nervous system of the entire "Movie Recommendation System with Sentiment Analysis," operating as the core Flask web application. It is responsible for orchestrating all server-side logic, managing HTTP requests from the frontend, seamlessly integrating with external data sources (TMDb API), and applying the sophisticated machine learning models for recommendations and sentiment analysis. Its design prioritizes efficiency, modularity, and responsiveness to user interactions.

#### **Application Initialization and Resource Loading**

To ensure optimal performance and immediate availability of critical resources, main.py performs several crucial one-time loading operations upon application startup. This pre-loading strategy significantly reduces latency for subsequent user requests.

* **Movie Data Loading (main\_data.csv):** Upon initialization, the application loads the preprocessed movie metadata from data/main\_data.csv into a Pandas DataFrame. This DataFrame, typically named df or movies\_df, becomes the in-memory repository for all structured movie information required by the recommendation engine. This includes movie titles, genres, director names, top actors, keywords, and the crucial comb (combined features) column. Loading this data into memory avoids constant disk I/O, enabling rapid lookups and computations.
* **Machine Learning Model and Vectorizer Loading:** The pre-trained sentiment analysis models, nlp\_model.pkl (the Multinomial Naive Bayes classifier) and tranform.pkl (the fitted TF-IDF vectorizer), are loaded from the models/ directory into memory. This is achieved using Python's pickle module. By loading these serialized objects once at startup, the system bypasses the computationally expensive process of retraining the models for every sentiment analysis request, thereby ensuring near real-time performance for review classification.
* **Cosine Similarity Matrix Pre-computation (cosine\_sim):** A fundamental component of the content-based recommendation engine is the cosine\_sim matrix. This square matrix stores the pre-calculated cosine similarity scores between every pair of movies in the main\_data.csv dataset, based on their comb feature. The computation of this matrix is performed once during application initialization. This pre-computation is vital; it transforms a potentially time-consuming runtime calculation into a quick lookup operation. When a user requests recommendations for a movie, the system merely retrieves a row from this matrix and sorts it, rather than performing real-time vectorization and similarity calculations across thousands of movies.
* **TMDb API Key Initialization:** The TMDb API key, essential for all external data fetching, is loaded. Adhering to security best practices, this key is typically retrieved from an environment variable (e.g., set via a .env file and loaded using python-dotenv), rather than being hardcoded. This prevents sensitive credentials from being exposed in the codebase and facilitates easier deployment across different environments.
* **NLTK Data Setup:** Necessary NLTK data resources, such as stopwords and stemmers (already downloaded during environment setup), are initialized or made accessible to the text preprocessing functions, ensuring they are ready for use when live reviews are fetched and prepared for sentiment analysis.

#### **Route Definitions and Request Handling**

main.py defines a series of Flask routes (endpoints) that map URLs to specific Python functions. These functions handle different types of HTTP requests from the frontend, process data, and return appropriate responses (HTML pages or JSON data).

* **/ (Home Route - GET Request):**
  + **Purpose:** This is the entry point for the web application. When a user navigates to the root URL (e.g., http://localhost:5000/), this route is activated.
  + **Implementation:** It simply renders the home.html template using Flask's render\_template function. This page provides the initial user interface, primarily featuring the movie search bar. No data processing or API calls are made at this stage, ensuring a fast initial load.
* **/autocomplete (POST Request):**
  + **Purpose:** This endpoint provides real-time movie title suggestions as the user types into the search bar on home.html. It is designed for quick, responsive feedback.
  + **Implementation:**
    1. It receives a partial movie title string (e.g., "The Dark K") from the frontend via an AJAX POST request.
    2. It queries the loaded main\_data.csv DataFrame, specifically the movie\_title column, to find titles that start with or contain the user's input.
    3. A predefined number of matching suggestions (e.g., top 10) are extracted.
    4. These suggestions are then returned to the frontend as a JSON list, allowing recommend.js to dynamically populate a dropdown list beneath the search bar. This significantly enhances user experience by guiding them to existing movie entries.
* **/recommend (POST Request):**
  + **Purpose:** This is the central endpoint for initiating the movie recommendation process. It is triggered when a user selects a movie from the autocomplete suggestions or submits a full movie title.
  + **Implementation:**
    1. Receives the selected movie\_title from the frontend.
    2. **Movie Index Mapping:** It first identifies the unique index of the selected movie within the loaded main\_data.csv DataFrame. This index is crucial for accessing its corresponding entry in the pre-computed cosine\_sim matrix. Robust error handling is included here to manage cases where the movie title might not be found in the dataset.
    3. **Recommendation Generation:** Using the movie's index, it retrieves the row of similarity scores from the cosine\_sim matrix. These scores are then sorted in descending order to identify the most similar movies. The top N similar movies (typically 20-30, excluding the movie itself to avoid self-recommendation) are extracted.
    4. **TMDb API Data Fetching:** For the selected movie and each of the recommended movies, the backend makes synchronous API calls to the TMDb API. These calls fetch comprehensive details such as:
       - Full plot summary (overview)
       - Poster image path (poster\_path) and backdrop image path (backdrop\_path)
       - Release date (release\_date)
       - Genres (mapped from IDs to names)
       - Director and top cast members (retrieved by parsing the credits endpoint)
       - Trailer video key (retrieved from the videos endpoint, usually for YouTube)
       - Budget, Revenue, Original Language (if needed for display)
    5. **Dynamic HTML Rendering:** Finally, main.py renders the recommend.html template. All the fetched details for the selected movie and the list of recommended movies are passed as Python variables to this template. Jinja2 then processes these variables to dynamically populate the HTML structure, creating a rich and personalized movie details and recommendation page for the user.
* **/get\_movie\_reviews (POST Request):**
  + **Purpose:** This endpoint is dedicated to fetching live user reviews for a specific movie and performing real-time sentiment analysis on them. It’s typically triggered by an AJAX call when the recommend.html page loads or a "Show Reviews" button is clicked.
  + **Implementation:**
    1. Receives the movie\_id from the frontend.
    2. **Live Review Fetching:** Makes an API call to the TMDb API's reviews endpoint for the given movie\_id.
    3. **Real-time Preprocessing:** Each fetched review text undergoes the exact same preprocessing pipeline (lowercasing, punctuation removal, stop word removal, stemming) that was applied to the training data. This ensures consistency for the model.
    4. **Sentiment Prediction:** The preprocessed review text is then transformed into a numerical vector using the loaded tranform.pkl (TF-IDF vectorizer). This vectorized input is then fed into the loaded nlp\_model.pkl (Multinomial Naive Bayes classifier) to predict its sentiment (0 for negative, 1 for positive).
    5. **Response Generation:** The original review text along with its predicted sentiment (converted to "Positive" or "Negative" strings) is structured as a JSON response and sent back to the frontend.
* **/get\_person\_details (POST Request):**
  + **Purpose:** This endpoint provides biographical information for actors or directors when their names are clicked on the frontend.
  + **Implementation:**
    1. Receives the person\_id (from TMDb) from the frontend.
    2. **Biography Fetching:** Queries the TMDb API's person endpoint for the given person\_id to retrieve their biography.
    3. **Response Generation:** The biography text is extracted and returned as a JSON response to the frontend, where recommend.js typically displays it within a modal pop-up.

#### **API Integration and Error Handling**

* **requests Library:** The Python requests library is used extensively across all routes that interact with external APIs. It provides a user-friendly and robust way to make HTTP GET requests, handle query parameters, and parse JSON responses from the TMDb API.
* **Error Handling:** Robust error handling mechanisms are embedded within API interaction functions. This includes:
  + Checking HTTP status codes (e.g., 200 for success, 404 for not found, 429 for rate limit, 500 for server errors).
  + Implementing try-except blocks to gracefully handle network issues or malformed JSON responses.
  + Providing fallback mechanisms or informative messages to the frontend in case an API call fails (e.g., "Reviews could not be loaded").

#### **Flask Application Execution**

The standard Python idiom if \_\_name\_\_ == '\_\_main\_\_': is used to control the execution of the Flask development server. When main.py is executed directly, this block ensures that app.run(debug=True) is called. This starts the Flask development server, making the application accessible via a specified port (e.g., http://127.0.0.1:5000/), enabling live reloading and detailed debugging output during development.

In summary, main.py acts as the central hub, intelligently routing requests, integrating diverse data sources and machine learning capabilities, and dynamically generating the content that provides the rich interactive experience of the Movie Recommendation System with Sentiment Analysis.

### **Frontend (User Interface)**

The frontend of the "Movie Recommendation System with Sentiment Analysis" is meticulously designed to provide an intuitive, responsive, and engaging user experience. It serves as the primary point of interaction, translating complex backend processes into a seamless visual and functional interface. The implementation leverages standard web technologies: HTML for structuring content, CSS for visual presentation and responsiveness, and JavaScript (with the jQuery library) for dynamic interactions and asynchronous communication with the Flask backend.

#### **HTML Structure (home.html, recommend.html)**

The application utilizes two primary HTML templates, rendered by Flask, each serving a distinct phase of the user journey. The structure emphasizes semantic HTML5 elements for accessibility and clarity.

* **home.html (Landing and Search Page):** This file represents the initial entry point for users. Its structure is deliberately clean and focused on facilitating the movie search process.
  + **Core Components:**
    - **Header (<header>):** Contains the application title, logo, and potentially a brief tagline, setting the context for the user.
    - **Main Search Area (<main>):** Houses the central interactive elements.
      * **Search Form (<form>):** Encapsulates the input field and any associated controls.
      * **Movie Search Input (<input type="text" id="movie\_name" ...>):** This is the primary interactive element where users type movie titles. It is carefully designed with an id (e.g., movie\_name) for easy JavaScript targeting and includes attributes for placeholder text to guide the user.
      * **Autocomplete Suggestions (<ul id="autocomplete\_results" ...>):** An initially hidden unordered list element that will be dynamically populated by JavaScript with movie title suggestions as the user types. This provides real-time feedback and improves discoverability.
      * **Submit Button (<button type="submit" ...>):** Triggers the form submission or a JavaScript event to initiate the recommendation process.
  + **External Resource Links:** Includes <link> tags in the <head> section to import style.css for visual styling and <script> tags, typically before the closing </body> tag, to load jQuery and recommend.js for client-side interactivity. The placement of scripts at the end helps ensure the HTML content is parsed and rendered before JavaScript tries to manipulate it.
* **recommend.html (Movie Details and Recommendations Page):** This file is dynamically rendered by the Flask backend after a movie has been selected from home.html. It serves as a comprehensive display hub for the chosen movie and its associated recommendations and reviews. Its structure is richer, accommodating various information panels.
  + **Dynamic Content Sections:** Each major piece of information has a dedicated container, often <div> elements with unique IDs or classes, allowing JavaScript to target them for content injection. Examples include:
    - **Movie Header (<section id="movie\_header">):** Displays the selected movie's poster image (<img>), title (<h1>), release year, and genres.
    - **Plot Summary (<p> or <div>):** Dedicated area for the movie's overview.
    - **Cast and Crew (<div id="cast\_info">):** Lists key actors and the director, often with clickable names to trigger biography modals.
    - **Trailer Section (<div id="trailer\_section">):** An <iframe> element dynamically populated with a YouTube embed URL for the movie's trailer.
    - **Recommendation Carousel/Grid (<div id="recommendations\_list">):** A container that will house individual movie "cards" for the recommended films. Each card typically includes a poster, title, and a link to view its details.
    - **User Reviews Section (<section id="reviews\_section">):** Contains individual containers for each fetched review, including the review text and a visual indicator (icon/text) for its predicted sentiment.
  + **Interactive Modals (<div id="biography\_modal" ...>):** Hidden div elements, typically structured as Bootstrap modals, which become visible when an actor's or director's name is clicked. They display the retrieved biography content.
  + **Loading Indicators (<div id="loading\_spinner" ...>):** Elements (e.g., <i> for an icon, <div> for a message) that are shown and hidden by JavaScript to provide visual feedback during asynchronous data fetching, informing the user that content is being loaded.

#### **Cascading Style Sheets (static/css/style.css)**

The style.css file, augmented by the power of Bootstrap 4, is responsible for the entire visual presentation and responsiveness of the web application. It transforms the raw HTML structure into an aesthetically pleasing and functional interface.

* **Bootstrap 4 Integration:**
  + The project extensively leverages **Bootstrap 4**, a popular open-source CSS framework. This significantly accelerates UI development by providing a comprehensive set of pre-designed CSS classes and JavaScript components.
  + **Grid System:** Bootstrap's flexible **grid system** (.container, .row, .col-md-\*, etc.) is fundamental for creating responsive layouts that adapt gracefully to various screen sizes (from mobile phones to large desktop monitors). This ensures the application is consistently usable across devices.
  + **Pre-built Components:** Components like navigation bars, cards (for movie recommendations), forms (for the search bar), modals (for biographies), buttons, and utility classes (e.g., for spacing, text alignment) are utilized to ensure a professional and consistent design language with minimal custom CSS.
* **Custom Styling (style.css):**
  + While Bootstrap provides a strong foundation, style.css contains custom CSS rules to override Bootstrap defaults and introduce unique visual elements specific to the application's branding and design requirements.
  + **Layout Adjustments:** Fine-tuning margins, padding, and alignment for specific sections (e.g., centering the search bar, spacing between movie cards).
  + **Typography:** Defining custom font families, font sizes, line heights, and text colors to enhance readability and establish a consistent textual hierarchy.
  + **Component Overrides:** Customizing the appearance of Bootstrap components (e.g., button colors, card shadows, modal dimensions) to match the desired aesthetic.
  + **Theming:** Establishing a consistent color palette for background elements, text, links, and interactive states.
  + **Visual Enhancements:** Implementing subtle animations (e.g., for loading spinners, hover effects on movie cards), transitions, and shadows to improve user engagement and perceived fluidity.
  + **Responsive Adjustments:** Adding media queries for specific breakpoints where more granular control over responsiveness is needed beyond Bootstrap's default behavior, ensuring optimal viewing on all devices.

#### **JavaScript Logic (static/js/recommend.js and jQuery)**

The recommend.js script, powered by the **jQuery library**, is the engine behind the dynamic and interactive functionalities of the frontend. It acts as the primary bridge between user actions in the browser and data interactions with the Flask backend via AJAX.

* **jQuery Advantage:** jQuery simplifies DOM manipulation, event handling, animation, and AJAX calls with its concise and intuitive syntax. This significantly reduces the amount of boilerplate JavaScript code required and handles cross-browser compatibility issues, accelerating development.
* **Key Functionalities and Implementation Details:**
  1. **Autocomplete Search ($(document).ready(...) and keyup event):**
     + An event listener is attached to the movie search input field (#movie\_name) that triggers on every keyup event.
     + When the user types, an **AJAX POST request** is sent to the Flask backend's /autocomplete endpoint. The current input value is sent as data.
     + Upon receiving a JSON response containing movie title suggestions from the backend, recommend.js dynamically populates the ul#autocomplete\_results element. Each suggestion is created as an <li> element.
     + Clicking on an autocomplete suggestion immediately populates the search bar with the full title and triggers the movie recommendation process.
  2. **Movie Search Submission (.on('submit', ...)):**
     + The form containing the search bar listens for a submit event. When triggered (either by pressing Enter or clicking a submit button), recommend.js prevents the default form submission (which would cause a full page reload).
     + Instead, it gathers the selected movie title and initiates an **AJAX POST request** to the Flask backend's /recommend endpoint.
     + Crucially, during this process, a **loading spinner (#loading\_spinner) is displayed**, and relevant content areas are hidden (.hide()), providing visual feedback to the user that data is being fetched.
  3. **Dynamic Content Rendering (on AJAX success for /recommend):**
     + Upon successful reception of the JSON response from the /recommend endpoint (containing movie details, recommendations, and initial reviews), recommend.js parses this data.
     + It then dynamically updates various sections of recommend.html. This involves:
       - Setting the src attribute of the poster <img> tag and background-image of a background div.
       - Populating <h1>, <p>, and <span> elements with movie title, plot, genres, and release year.
       - Dynamically generating **movie recommendation cards** (e.g., div.movie-card) within the #recommendations\_list container. Each card typically includes a movie poster and title, and is made clickable to view its details.
       - Dynamically generating **user review entries** within the #reviews\_section, often displaying the review text and an initial sentiment icon/text.
     + The loading spinner is then hidden, and the relevant content sections are made visible.
  4. **Review Fetching and Sentiment Display (on page load or button click):**
     + An AJAX call to /get\_movie\_reviews is often made when recommend.html first loads, or when a specific "Load Reviews" button is clicked. The movie's TMDb ID is sent.
     + The returned JSON (containing review text and predicted sentiment for each) is processed.
     + For each review, recommend.js dynamically creates an HTML element (e.g., a <div> or <li>) and populates it with the review text. A visual indicator (e.g., a green icon for "Positive," red for "Negative," or corresponding text labels) is added based on the sentiment field from the backend. This provides immediate visual cues about audience reception.
  5. **Biography Modals (on click for actor/director names):**
     + Event listeners are attached to clickable actor and director names on recommend.html.
     + When a name is clicked, an AJAX POST request is sent to the /get\_person\_details endpoint, passing the person's TMDb ID.
     + The received biography text is then dynamically injected into a hidden modal element (#biography\_modal).
     + The Bootstrap modal is then triggered to display (.modal('show')), providing additional contextual information about the cast or crew member without navigating away from the page.

The combination of structured HTML, comprehensive CSS, and dynamic JavaScript/jQuery ensures that the frontend of the Movie Recommendation System is not just visually appealing but also highly interactive, delivering a fluid and efficient user experience by minimizing full page reloads and providing real-time feedback.

### **Data Processing Script (fetch\_and\_preprocess\_data.py)**

The fetch\_and\_preprocess\_data.py script stands as a crucial, standalone utility within the project, operating independently of the main Flask application. Its primary responsibility is to orchestrate the acquisition of raw movie metadata from the TMDb API and subsequently transform this raw data into a clean, structured format (main\_data.csv) that is directly consumable by the content-based recommendation engine. This separation of concerns ensures that the computationally intensive data preparation phase does not interfere with the live application's performance, allowing for periodic updates to the dataset without requiring application downtime.

#### **Robust Data Acquisition from TMDb API**

The script initiates its process by establishing a programmatic interface with the TMDb API, leveraging Python's requests library for efficient HTTP communication. The design for data acquisition emphasizes comprehensiveness and resilience.

* **API Key Management:** The TMDb API key is securely loaded, typically from an environment variable (as detailed in Section 4.3.5), ensuring that sensitive credentials are not hardcoded. This key authenticates requests and grants access to TMDb's extensive database.
* **Iterative Data Fetching Strategy:** Instead of attempting to download the entire TMDb database at once (which would be impractical and hit rate limits), the script implements an iterative fetching strategy. This often involves:
  + **Year-Based Iteration:** Looping through a predefined range of years (e.g., from min\_year to max\_year, often extended a few years into the future to capture upcoming releases) to query movies released within specific periods. This allows for a systematic and manageable collection process.
  + **Pagination:** For each year, TMDb API responses are typically paginated. The script intelligently handles pagination by iterating through multiple pages of results until all movies for a given query are retrieved or a specified limit is reached.
* **Detailed Movie Information Retrieval:** For each movie identified through the initial queries, the script makes subsequent, more granular API calls to fetch comprehensive details crucial for the recommendation logic. This often involves:
  + **Primary Movie Details:** Fetching general information such as title, overview (plot summary), and release\_date.
  + **Genre Mapping:** Retrieving genre\_ids and then mapping them to human-readable genre names using TMDb's genre list endpoint. This ensures consistency and clarity.
  + **Credits (/movie/{movie\_id}/credits):** A critical API call to extract **cast and crew information**. The script specifically parses this response to identify the director\_name (by looking for 'Director' in the job field) and the names of the top three actors (based on order in the cast list). This detailed parsing is essential as this information is not directly available in basic movie queries.
  + **Keywords (/movie/{movie\_id}/keywords):** Another vital API call to retrieve associated keywords or tags. These keywords provide granular thematic descriptors that significantly contribute to content similarity.
* **Robust Error Handling and Rate Limiting:** Interacting with external APIs necessitates robust error management. The script incorporates:
  + **HTTP Status Code Checks:** Monitoring HTTP response codes (e.g., 200 for success, 404 for not found, 429 for too many requests).
  + **Retry Mechanisms:** Implementing retry logic with exponential backoff for transient errors (e.g., network timeouts, temporary server issues). This involves waiting for increasing durations before retrying a failed request, reducing the chance of overwhelming the API.
  + **Rate Limit Management:** While requests doesn't directly handle rate limits, the iterative fetching and potential delays between requests (implicit or explicit time.sleep()) help mitigate hitting TMDb's API rate limits.
* **Progress Indicators (tqdm):** The tqdm library is integrated to provide a visual progress bar during the data fetching process. This is particularly useful for long-running scripts, offering real-time feedback on progress and estimated completion time, enhancing the developer's experience.

#### **Comprehensive Feature Extraction and Transformation**

Once the raw JSON data is acquired, it undergoes a meticulous transformation pipeline to prepare it for the recommendation engine. The goal is to standardize, clean, and consolidate textual features.

* **Initial DataFrame Construction:** The fetched movie data is initially collected into a list of dictionaries, which is then converted into a Pandas DataFrame. This provides a powerful and flexible structure for subsequent data manipulation.
* **Handling Missing Values (NaNs):** A critical preprocessing step involves systematically identifying and handling missing values (represented as NaN or None) in key feature columns (director\_name, actor\_1\_name, actor\_2\_name, actor\_3\_name, genres, keywords). To prevent errors during string concatenation and ensure all movies have a complete feature set, these NaN values are explicitly replaced with empty strings (''). This guarantees that the concatenation process yields valid strings for all entries.
* **Feature Normalization and Cleaning:**
  + **Lowercasing:** All textual features that will contribute to the comb column (names, genres, keywords) are converted to lowercase. This standardizes terms (e.g., "Action" and "action" are treated identically), reducing redundancy and improving consistency for TF-IDF vectorization.
  + **Whitespace Stripping:** Leading and trailing whitespaces are removed from all string fields (.strip()) to prevent accidental creation of unique tokens due to minor formatting discrepancies.
* **The comb Feature Engineering:** A pivotal step in preparing the data for content-based recommendations is the creation of the comb (combined features) column. This column is generated by concatenating the cleaned and normalized values of the director\_name, actor\_1\_name, actor\_2\_name, actor\_3\_name, genres, and keywords for each movie.
  + **Purpose:** The comb string serves as a single, holistic textual representation of a movie's content. By pooling all relevant descriptors into one feature, the TF-IDF vectorizer can then effectively compute the importance of each term within the context of the movie and the entire dataset, facilitating a robust similarity comparison between movies.
  + **Delimiters (Implicit):** While explicit delimiters might not be used (spaces typically separate terms), the combination creates a "document" for each movie.
* **Duplicate Removal:** The script includes logic to identify and remove duplicate movie entries, typically based on the movie\_title. This step ensures that each movie is uniquely represented in main\_data.csv, preventing redundant recommendations and maintaining the integrity of the dataset used by the recommendation engine.

#### **Output Generation (main\_data.csv)**

The culmination of the data acquisition and preprocessing efforts is the creation of the main\_data.csv file.

* **Saving to CSV:** The fully processed and cleaned Pandas DataFrame is saved as a Comma Separated Values (.csv) file named main\_data.csv within the data/ directory. This is achieved using Pandas' to\_csv() method, ensuring proper encoding and index handling.
* **Role in Application:** This main\_data.csv file is the foundational dataset loaded by the main.py Flask application at runtime. Its well-structured and pre-processed nature enables efficient in-memory operations for building the TF-IDF matrix and the cosine similarity matrix, which are central to the recommendation process.

By separating this complex and potentially time-consuming data preparation into a dedicated script, the project ensures that the core web application remains lean, fast, and focused on serving user requests, while providing a clear and reproducible pipeline for updating its underlying knowledge base.

Excellent! That's another comprehensive section detailed. You're building a very robust implementation chapter.

Now, let's proceed to expanding your *new* **4.4.4 Machine Learning Models (nlp\_model.pkl, tranform.pkl)**. This section is vital for explaining how your trained models are managed and integrated into the live system for real-time sentiment analysis. We'll delve deeper into their purpose, generation, and seamless integration.

Here's the draft for an expanded **4.4.4 Machine Learning Models (nlp\_model.pkl, tranform.pkl)**. Please **replace** your current content under your *new* "4.4.4 Machine Learning Models (nlp\_model.pkl, tranform.pkl)" with this new, more detailed version.

### **Machine Learning Models (nlp\_model.pkl, tranform.pkl)**

The sentiment analysis functionality, a hallmark feature of this movie recommendation system, is powered by two critical pre-trained and serialized machine learning assets: a Natural Language Processing (NLP) classification model and its accompanying text vectorization transformer. These components encapsulate the intelligence derived from extensive offline training and are instrumental for performing real-time inference on live movie reviews. Their strategic generation, persistence, and efficient loading are key to the system's responsiveness and accuracy.

#### **nlp\_model.pkl (Sentiment Classifier)**

* **Purpose and Role in the System:** The nlp\_model.pkl file contains the **serialized Multinomial Naive Bayes classifier**, which is the core intelligence unit for predicting the sentiment (positive or negative) of a given movie review text. This model represents the complex patterns and probabilistic relationships learned during its training on a large corpus of labeled review data. Its primary role in the live application is to accept a numerically represented review and output a categorical prediction (positive or negative sentiment), thereby transforming raw text into actionable emotional insight. It is an indispensable component for providing qualitative analysis of user feedback beyond mere numerical ratings.
* **Generation and Training Process:** This model is the direct output of the offline training workflow, meticulously documented and executed within the sentiment.ipynb Jupyter Notebook. The training process involves:  
    
  1. **Data Ingestion:** Loading the pre-labeled reviews.csv (IMDb 50K Movie Reviews dataset) into a Pandas DataFrame.
  2. **Advanced Text Preprocessing:** Applying a series of rigorous Natural Language Processing (NLP) techniques to the raw review texts. This includes lowercasing, removal of punctuation and special characters, elimination of common English stopwords (e.g., "the," "is," "and"), and stemming (reducing words to their root form, like "running" to "run" using NLTK's Porter Stemmer). These steps normalize the text, reduce noise, and optimize the feature space for the model.
  3. **Feature Vectorization:** Transforming the cleaned text reviews into numerical feature vectors using a TF-IDF (Term Frequency-Inverse Document Frequency) approach. This process quantifies the importance of each word within a review relative to its frequency across the entire training corpus. The TfidfVectorizer from scikit-learn is used for this transformation.
  4. **Model Training:** The vectorized training data (features) and corresponding sentiment labels (targets) are then fed to the Multinomial Naive Bayes algorithm. This algorithm is particularly well-suited for text classification due to its probabilistic nature and efficiency with discrete features. During training, the model learns the likelihood of observing specific words or patterns given a positive or negative sentiment, as well as the prior probabilities of each sentiment class.
* **Model Persistence (Pickling):** Upon successful training and evaluation (often including cross-validation and performance metrics like accuracy, precision, recall, and F1-score, though not detailed here for brevity), the fully trained Multinomial Naive Bayes classifier object is **serialized** (or "pickled") into the nlp\_model.pkl file. This serialization process converts the Python object (the model instance along with its learned parameters, vocabulary, and internal state) into a byte stream that can be written to disk. The pickle module, a standard Python library, is used for this purpose. This step is critical as it allows the model to be saved once offline and then efficiently loaded directly into memory without requiring retraining every time the Flask application starts or a prediction is needed.

#### **tranform.pkl (TF-IDF Vectorizer)**

* **Purpose and Role in the System:** The tranform.pkl file holds the **serialized TF-IDF Vectorizer** that was *fitted* on the exact same sentiment analysis training data (reviews.csv). Its purpose is paramount: to consistently transform any raw text data (both during training and for new, live reviews during inference) into the numerical TF-IDF feature vectors that are the required input format for the nlp\_model.pkl. This vectorizer encapsulates the entire vocabulary derived from the training corpus and the calculated Inverse Document Frequency (IDF) values for each term. Without this identical vectorizer, the sentiment model would receive feature vectors in a different format than it was trained on, leading to erroneous predictions.
* **Generation Process:** Similar to nlp\_model.pkl, this asset is also a product of the sentiment.ipynb Jupyter Notebook.
  1. **Fitting on Training Data:** The TfidfVectorizer from scikit-learn is instantiated and then explicitly fit to the preprocessed reviews.csv text data. The fit method analyzes the entire training corpus to build its internal vocabulary (mapping words to unique numerical indices) and calculate the IDF values for all words.
  2. **Vectorizer Persistence (Pickling):** After the fit operation, the entire fitted TfidfVectorizer object (containing its vocabulary, IDF values, and configurations) is serialized using pickle into the tranform.pkl file. This ensures that the exact same transformation logic, vocabulary, and term weights are applied to any new text data.

#### **Integration and Efficient Loading within main.py**

The strategic placement and loading of these .pkl files are fundamental to the real-time sentiment analysis capability of the live Flask application (main.py).

**Loading at Application Startup:** Both nlp\_model.pkl and tranform.pkl are loaded into the Flask server's memory once, at the very beginning of the main.py script's execution (if \_\_name\_\_ == '\_\_main\_\_': block, or within the app initialization). This is typically achieved by using pickle.load() to deserialize the files from the models/ directory.  
  
 Python  
import pickle

# ... other imports ...

# Load the TF-IDF vectorizer

with open('models/tranform.pkl', 'rb') as file:

    tfidf\_vectorizer = pickle.load(file)

# Load the sentiment classification model

with open('models/nlp\_model.pkl', 'rb') as file:

    sentiment\_model = pickle.load(file)

# ... Flask app setup ...

* This approach ensures that the models are immediately available for use whenever a user requests sentiment analysis, eliminating the overhead of loading them from disk for each individual review.
* **Real-time Inference Consistency:** Within the /get\_movie\_reviews route of main.py, when live reviews are fetched from the TMDb API, they undergo the exact same preprocessing steps (lowercasing, punctuation removal, stop word removal, stemming) as was performed on the training data. The preprocessed reviews are then passed to the loaded tfidf\_vectorizer's transform() method (note: transform, not fit\_transform) to convert them into numerical TF-IDF vectors. Finally, these vectors are fed to the loaded sentiment\_model's predict() method to obtain the sentiment classification. The strict adherence to the same preprocessing and vectorization pipeline for both training and inference is the most critical aspect of this integration, ensuring that the nlp\_model.pkl receives data in the precise format it was trained to interpret, thereby maximizing prediction accuracy and system reliability.

By encapsulating and persisting these machine learning assets, the project achieves a highly efficient and scalable sentiment analysis pipeline, seamlessly integrating advanced NLP capabilities into a dynamic web application.

## **Data Storage and Management**

Efficient and appropriate data storage and management are fundamental to the performance, scalability, and overall functionality of any software system. For the "Movie Recommendation System with Sentiment Analysis," a pragmatic and lightweight approach to data management was adopted, primarily relying on local file storage for pre-processed datasets and real-time data retrieval from external APIs. This strategy was chosen to balance performance requirements, data freshness, and simplified deployment, intentionally avoiding the complexities of integrating and managing a dedicated relational or NoSQL database server for the specific scope of this project.

The data management philosophy hinges on two core principles:

1. **Offline Preparation for Core Data:** Computations and data consolidation that are relatively static or change infrequently (like the primary movie metadata for content-based recommendations) are performed offline and stored in readily accessible local files. This minimizes runtime overhead and ensures quick access.
2. **Real-time External Fetching for Dynamic Data:** Information that is highly volatile, frequently updated, or too vast to store locally (such as live user reviews or very granular movie details beyond the core set) is fetched on demand from external, authoritative APIs.

### **Movie Metadata Storage (main\_data.csv)**

* **Purpose and Role:** The main\_data.csv file, located in the data/ directory, serves as the **centralized and pre-processed data source for the content-based movie recommendation engine**. It is not merely a raw dump of movie information but a carefully curated dataset containing structured, cleaned, and feature-engineered movie attributes. Key columns include:
  1. movie\_title: The definitive title used for user search and display.
  2. genres: Cleaned and concatenated genre names.
  3. director\_name: The extracted director's name.
  4. actor\_1\_name, actor\_2\_name, actor\_3\_name: Names of the top three billed actors.
  5. keywords: Relevant descriptive tags associated with the movie.
  6. comb: The critical combined textual feature (concatenation of director, actors, genres, and keywords), which is the primary input for TF-IDF vectorization and subsequent cosine similarity calculations.
  7. tmdbId: The unique identifier from TMDb, crucial for linking to external API calls for dynamic data like posters, trailers, and reviews.
* **Management and Lifecycle:**
  1. **Offline Generation:** main\_data.csv is not manually created. Instead, it is **generated programmatically and entirely offline** by the fetch\_and\_preprocess\_data.py script. This script orchestrates the complex process of making multiple API calls to TMDb, extracting specific features, cleaning the data (e.g., handling missing values by replacing NaNs with empty strings), normalizing text (e.g., lowercasing), performing feature engineering (creating the comb column), and removing duplicates. This robust pipeline ensures data quality and consistency.
  2. **Application Loading:** The Flask application (main.py) performs a **one-time loading of main\_data.csv into memory as a Pandas DataFrame during its initialization**. This strategy is paramount for performance. By having the entire dataset readily available in RAM, the system can perform rapid lookups for movie titles, access all necessary features for vectorization, and, most importantly, efficiently compute and utilize the cosine\_sim matrix without continuous disk I/O, which would be a significant bottleneck for a recommendation engine.
  3. **Update Strategy:** Updates to this core movie dataset (e.g., to include newer movies or refresh existing data) require re-execution of the fetch\_and\_preprocess\_data.py script. This offline update mechanism ensures that the live application remains stable and performs optimally, as data updates are not performed during active serving.

### **Sentiment Analysis Training Data (reviews.csv)**

* **Purpose and Role:** The reviews.csv file, also located in the data/ directory, holds the **raw IMDb Dataset of 50K Movie Reviews**, a widely recognized benchmark dataset for sentiment analysis. This file is **exclusively used for the offline training** of the sentiment analysis model (nlp\_model.pkl and tranform.pkl). It contains unstructured text reviews paired with their corresponding binary sentiment labels (positive/negative), providing the supervised learning foundation for the sentiment classifier.
* **Management and Lifecycle:**
  1. **Static Dataset:** Unlike main\_data.csv, reviews.csv is a static, downloaded dataset (e.g., from Kaggle). It is not dynamically updated by the application.
  2. **Offline Consumption:** This file is read and processed only by the sentiment.ipynb Jupyter Notebook during the model training phase. The raw text from this file undergoes extensive preprocessing (tokenization, stop word removal, stemming) before being vectorized and used to train the Multinomial Naive Bayes classifier.
  3. **No Live Access:** Crucially, the live Flask application (main.py) **does not directly access or read reviews.csv at runtime**. Its only interaction with the sentiment analysis component is through loading the already trained and persisted models (nlp\_model.pkl and tranform.pkl). This clear separation ensures that the training data does not burden the operational system.

### **Machine Learning Model Persistence (nlp\_model.pkl, tranform.pkl)**

* **Purpose and Role:** The models/ directory serves as the repository for two critical **serialized Python objects**: nlp\_model.pkl (the trained Multinomial Naive Bayes classifier) and tranform.pkl (the fitted TF-IDF vectorizer). These files contain the learned parameters, internal state, and vocabulary of the machine learning models, representing the "knowledge" acquired during the offline training phase.
* **Management and Lifecycle:**
  1. **Offline Generation:** These .pkl files are generated by the sentiment.ipynb notebook after the sentiment analysis model training process is complete. The pickle module is used for their serialization. This process converts the Python objects into a byte stream that can be saved to disk.
  2. **Efficient Runtime Loading:** The main.py Flask application loads both nlp\_model.pkl and tranform.pkl directly into memory during its initialization. This one-time loading is highly efficient. It eliminates the need for retraining models on every application restart or for every prediction request, significantly boosting the system's responsiveness for real-time sentiment predictions on live reviews.
  3. **Consistency Guarantee:** Loading the *exact same* tranform.pkl (fitted vectorizer) during inference as was used during training is paramount. This guarantees that any live review text will be transformed into numerical features using the identical vocabulary and weighting scheme, ensuring accurate and reliable predictions from the nlp\_model.pkl.

### **External Data Retrieval (TMDb API)**

* **Purpose and Role:** For highly dynamic, volatile, or very large datasets that are impractical or unnecessary to store locally, the system relies extensively on the **TMDb API for real-time data retrieval**. This includes:
  1. **Up-to-date Movie Details:** Fetching current movie posters, backdrop images, official trailer links, and very recent popularity metrics.
  2. **Live User Reviews:** Critically, reviews for a specific movie are fetched directly from the TMDb API when requested by the user. This ensures that the sentiment analysis is performed on the most current and dynamic audience opinions, providing fresh insights.
  3. **Actor/Director Biographies:** Biographical information is fetched on demand to display in modals.
* **Management and Lifecycle:**
  1. **On-Demand Fetching:** Data from the TMDb API is not persistently stored within the application's local file system (beyond temporary caching by the requests library itself or during the fetch\_and\_preprocess\_data.py run). Instead, it is fetched synchronously by the main.py backend (or asynchronously if using a task queue, though not implemented here) only when a specific piece of information is requested by the frontend.
  2. **Real-time Processing:** Data fetched from the TMDb API (e.g., raw review text) is immediately processed (preprocessed, vectorized, predicted) and then passed to the frontend for display. It is typically not saved to local files for subsequent use in the live application context, prioritizing data freshness.
  3. **Benefits of API-driven approach:** This approach minimizes the local storage footprint, ensures data currency, and offloads the burden of maintaining and updating vast, volatile datasets to TMDb, which is their core business. It also provides access to a richer set of data than would be feasible to store locally.

This comprehensive data management strategy, by judiciously combining efficient local file-based storage for core processed data and dynamic, on-demand retrieval from external APIs for real-time information, strikes an optimal balance. It minimizes system overhead, ensures data freshness, simplifies deployment (as no separate database server is required), and allows the application to focus its resources on its core recommendation and sentiment analysis functionalities.

## **Key Implementation Details**

Beyond the structural organization and the specific functionalities of individual modules, the success of the "Movie Recommendation System with Sentiment Analysis" is significantly attributed to several key implementation details and coding practices. These aspects address specific technical challenges, enhance system robustness, improve security, and contribute directly to a superior user experience. Understanding these granular decisions provides deeper insight into the engineering choices made during development.

### **API Key Management and Security**

* **Implementation Detail:** To safeguard sensitive credentials and adhere to best security practices, the TMDb API key is **not hardcoded** directly into any source code files (e.g., main.py, fetch\_and\_preprocess\_data.py). Instead, it is managed as an **environment variable**, typically loaded from a .env file using the python-dotenv library.
* **Rationale and Impact:**
  + **Security:** This approach prevents the API key from being inadvertently exposed if the codebase is shared publicly (e.g., on GitHub) or compromised. Environment variables are kept separate from the application's source code.
  + **Flexibility:** It allows for easy switching of API keys between different deployment environments (development, testing, production) without modifying the core code.
  + **Maintainability:** Updates to the key simply require changing the .env file, not altering and redeploying code.

**Code-level Illustration (Conceptual):** The Python scripts (main.py, fetch\_and\_preprocess\_data.py) would include:  
Python  
import os

from dotenv import load\_dotenv

load\_dotenv() # This line loads variables from the .env file into the environment

TMDB\_API\_KEY = os.getenv("TMDB\_API\_KEY") # Access the variable securely

* This ensures that the API key is retrieved dynamically at runtime from a secure location, rather than being embedded within the application logic.

### **Robust Data Preprocessing for Recommendations**

* **Implementation Detail:** In the fetch\_and\_preprocess\_data.py script, meticulous attention was paid to the **preprocessing of movie metadata** before it's saved to main\_data.csv. A critical step involves explicitly handling NaN (Not a Number) values in all columns designated for the comb feature (e.g., director\_name, actor\_names, genres, keywords). These NaN values are systematically replaced with empty strings ('').
* **Rationale and Impact:**
  + **Data Integrity:** Prevents concatenation errors when forming the comb string. If NaN values were present, string operations would fail or result in unwanted "nan" strings within the combined feature.
  + **Model Compatibility:** Ensures that the TfidfVectorizer (used to process the comb column into numerical vectors) can process *all* movie entries without encountering non-string data types, maintaining a complete and consistent feature set for the recommendation engine.
  + **Feature Completeness:** Even if a movie lacks a specific piece of metadata (e.g., no keywords), its comb string is still well-formed and participates fully in the similarity calculation, just with a "null" contribution from that particular missing feature.

### **Consistency in Text Preprocessing for Sentiment Analysis**

* **Implementation Detail:** A foundational principle for the accuracy of the sentiment analysis model is the **absolute consistency of the text preprocessing pipeline** between the offline model training phase (sentiment.ipynb) and the online inference phase (main.py). The exact same sequence of steps – lowercasing, punctuation and special character removal, stop word removal (using the same NLTK stopwords list), and stemming (using the same Porter Stemmer) – is applied to both the reviews.csv training data and the live reviews fetched from the TMDb API.
* **Rationale and Impact:**
  + **Feature Space Alignment:** Ensures that the words (tokens) and their representations (stems) in the live reviews are identical to those the tranform.pkl (TF-IDF vectorizer) and nlp\_model.pkl (classifier) were trained on. Any deviation (e.g., different stemming rules, or leaving punctuation in live reviews but removing it from training data) would lead to a mismatch in the feature space, rendering the model unable to interpret new inputs correctly.
  + **Accuracy and Reliability:** This rigorous consistency is paramount for maximizing the accuracy and reliability of the sentiment predictions, as the model "sees" new data in the format it expects.

### **Model Persistence and Efficient Loading**

* **Implementation Detail:** The trained sentiment classifier (nlp\_model.pkl) and the fitted TF-IDF vectorizer (tranform.pkl) are **serialized using Python's pickle module** after their respective training/fitting processes in sentiment.ipynb. These .pkl files are then saved to the models/ directory. Crucially, in main.py, these models are **loaded into the Flask application's memory only once at startup**.
* **Rationale and Impact:**
  + **Performance Optimization:** This strategy eliminates the time-consuming and computationally expensive process of retraining or refitting the models with every application restart or, more critically, with every individual sentiment prediction request. Loading from disk is significantly faster than training.
  + **Scalability:** By pre-loading, the system becomes highly responsive for real-time sentiment analysis tasks, as the models are immediately available in RAM, ready to process incoming reviews with minimal latency.
  + **Resource Management:** Avoids redundant computations, conserving CPU cycles and memory.

### **Asynchronous Communication (AJAX) for Enhanced UX**

* **Implementation Detail:** The frontend (recommend.js) extensively employs **AJAX (Asynchronous JavaScript and XML) requests** to communicate with the Flask backend. Instead of traditional full-page form submissions that trigger a complete page reload, user actions (like typing in the search bar, selecting a movie, or clicking to view reviews/biographies) send asynchronous requests.
* **Rationale and Impact:**
  + **Improved User Experience (UX):** Prevents disruptive full page reloads, making the user interface feel more responsive, fluid, and modern (akin to a Single-Page Application or SPA). Users experience continuous interaction without jarring screen flashes.
  + **Faster Content Delivery:** Only the necessary data is transmitted and updated, rather than re-rendering the entire HTML document. This reduces bandwidth usage and accelerates content delivery.
  + **Dynamic Content:** Enables dynamic updates of specific sections of the page (e.g., autocomplete suggestions appearing as you type, reviews loading into a dedicated panel) without affecting other content.
* **Visual Feedback:** Often coupled with AJAX, loading indicators (spinners or text messages) are displayed during asynchronous operations, providing visual feedback to the user that data is being fetched in the background, preventing confusion or perceived system freezes.

### **Dynamic TMDb API Integration and Data Parsing**

* **Implementation Detail:** The main.py script rigorously integrates with the TMDb API for diverse, real-time data points. This involves not just making HTTP requests, but **carefully parsing complex, nested JSON responses** from different TMDb endpoints (e.g., /movie/{id}, /movie/{id}/credits, /movie/{id}/videos, /movie/{id}/reviews). The project dynamically constructs full URLs for assets like movie posters and trailers based on the partial paths provided by TMDb and their fixed base URLs.
* **Rationale and Impact:**
  + **Data Freshness:** Ensures that movie details, reviews, and visual assets are always current and reflect the latest information available on TMDb.
  + **Comprehensive Information:** Allows the application to display a rich set of details (director, specific actors, budget, revenue, trailer) that would be cumbersome or inefficient to store locally for every movie.
  + **Flexibility:** Adapts to TMDb's API structure, ensuring that the correct information is extracted from potentially complex JSON objects (e.g., iterating through crew list to find a job Director).
  + **Visual Richness:** Dynamic construction of image and video URLs ensures that movie posters and trailers are correctly displayed without being hardcoded, adapting to changes in TMDb's CDN paths if they occur.

### **Mapping Movie Titles to IDs**

* **Implementation Detail:** To facilitate seamless interaction between the user's selected movie title (from the frontend search) and the TMDb API (which primarily uses internal movie IDs for its endpoints), the backend (main.py) efficiently **maps the selected title to its corresponding TMDb ID**. This mapping is performed by referencing the tmdbId column in the loaded main\_data.csv DataFrame, using the movie\_title as the lookup key.
* **Rationale and Impact:**
  + **API Compatibility:** Bridges the gap between user-friendly textual input and the API's numerical identifier-based lookup system.
  + **Efficiency:** Once the TMDb ID is retrieved, all subsequent API calls for that movie (e.g., for reviews, full details, credits) can be made directly and efficiently using the unique ID, avoiding redundant title-based searches.
  + **Accuracy:** Ensures that the correct movie is targeted for all data retrieval, even if multiple movies share similar titles or the user input is slightly ambiguous (the selection from autocomplete helps disambiguate).

These detailed implementation choices collectively contribute to the robustness, efficiency, security, and exceptional user-friendliness of the Movie Recommendation System with Sentiment Analysis, demonstrating a thoughtful approach to software development.

# **CHAPTER 5: TESTING AND RESULTS**

## **Testing Methodology**

Testing is an indispensable phase in the software development lifecycle, crucial for validating that the system functions as designed, meets all specified requirements, and delivers a robust and reliable user experience. For the "Movie Recommendation System with Sentiment Analysis," a systematic testing methodology was adopted to ensure the accuracy of recommendations, the correctness of sentiment predictions, and the overall usability of the web application.

The testing approach focused primarily on **functional testing** and **user acceptance testing (UAT)**. Functional testing verified that each feature of the system performed its intended task according to the functional requirements outlined in Chapter 3. UAT, though informal in this context, involved interacting with the system from an end-user perspective to assess its usability, intuitiveness, and overall satisfaction.

Given the scope of the project, formal automated unit tests and integration tests were not the primary focus; instead, a comprehensive manual testing strategy was employed. This involved a series of predefined test cases and scenarios executed directly through the web interface, covering all critical functionalities: movie search, recommendation generation, movie details display, live review fetching, sentiment classification, and actor/director biography display.

The testing process involved:

* **Test Case Definition:** Identifying specific scenarios and expected outcomes for each functional area.
* **Execution:** Systematically running the application and performing actions as per the defined test cases.
* **Observation and Verification:** Comparing the actual system behavior and output against the expected results.
* **Documentation:** Recording the results, including successful outcomes, identified issues, and capturing screenshots to visually confirm functionality.

This methodical approach ensured that the implemented features were thoroughly validated before considering the project complete.

## **Test Cases and Scenarios**

To systematically verify the functionality of the Movie Recommendation System with Sentiment Analysis, a series of test cases and scenarios were designed and executed. These scenarios cover the primary user interactions and core system functionalities, ensuring that each component behaves as expected. The following table (conceptual, you won't literally create a table in the report unless you want to) outlines the key test areas and scenarios:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Feature Tested** | **Scenario Description** | **Expected Outcome** |
| TC-UI-001 | Movie Search & Autocomplete | User types partial movie title in search bar (e.g., "Aveng"). | Autocomplete suggestions for "Avengers" movies should appear. |
| TC-UI-002 | Movie Details Display | User selects a movie from autocomplete or presses Enter after typing full title. | The recommend.html page should load, displaying correct movie poster, title, plot, cast, director, genres, and trailer. |
| TC-REC-001 | Content-Based Recommendation | After a movie's details are displayed. | A list of relevant movies, similar in genre, cast, or director, should be displayed in the recommendations section. |
| TC-SEN-001 | Live Review Fetching & Sentiment | On the recommend.html page with reviews present. | Multiple user reviews for the selected movie should be fetched and displayed. |
| TC-SEN-002 | Sentiment Classification Display | For each fetched review. | A clear indication (e.g., "Positive" / "Negative" label or icon) of the predicted sentiment should be visible next to each review. |
| TC-BIO-001 | Actor/Director Biography | User clicks on a cast member's or director's name. | A modal pop-up should appear, displaying the biography of the clicked person. |
| TC-ERR-001 | Invalid Search / No Movie Found | User searches for a non-existent movie title (e.g., "asdfghjkl"). | An appropriate "Movie Not Found" or error message should be displayed. |
| TC-RES-001 | Responsive Design | Accessing the application on different screen sizes (e.g., desktop, mobile browser). | The UI layout should adapt and remain functional and visually appealing across various device widths. |

Table 5‑1 Test Cases and Scenarios

These test cases formed the backbone of the manual testing process, ensuring that the critical paths of the application were thoroughly evaluated. The subsequent sections will provide visual evidence and detailed descriptions of the results for these and other related tests.

## **User Interface (UI) Testing**

The user interface (UI) is the primary point of interaction for users, and its design and functionality are critical for a positive user experience. This section presents the results of testing the UI components of the Movie Recommendation System with Sentiment Analysis.

### **Home Page (Movie Search)**

The home.html page provides the initial interface for movie search.

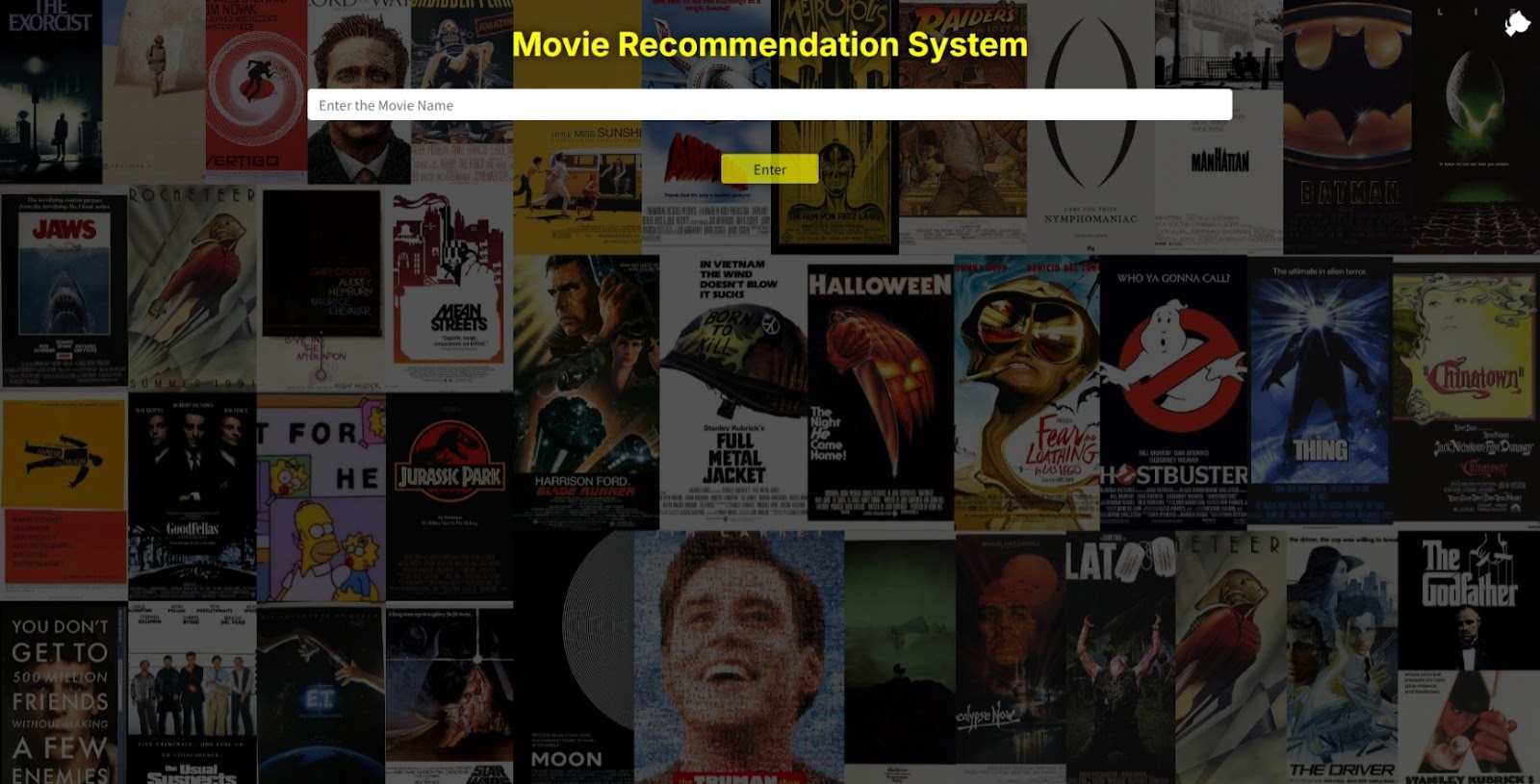


Figure 5‑1 Home Page

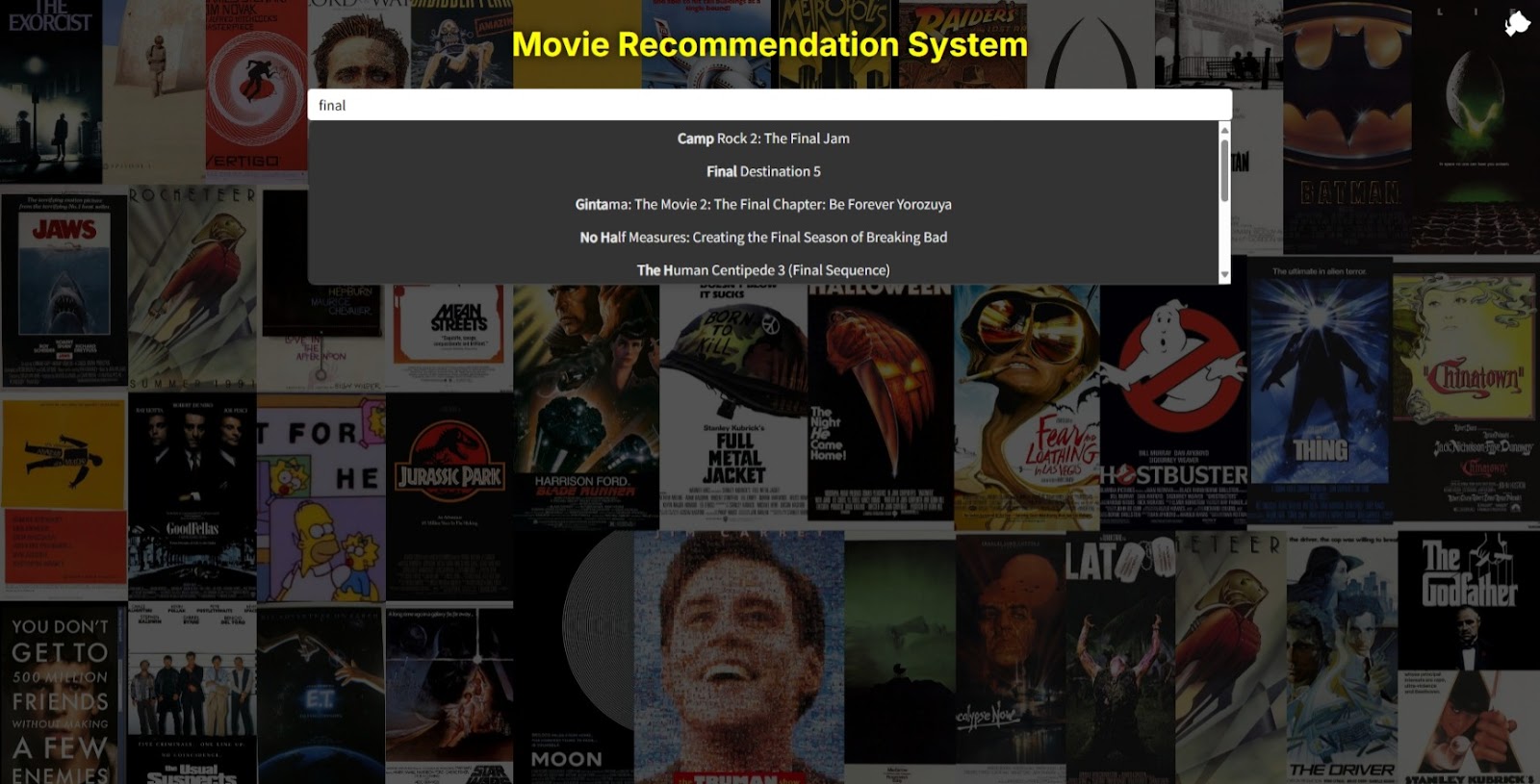


Figure 5‑2 Autocomplete Functionality

Figure 5-2 shows the clean and intuitive search interface. The autocomplete functionality, as demonstrated in Test Case TC-UI-001, provides real-time movie suggestions as the user types, enhancing search efficiency. The layout is simple and uncluttered, focusing the user's attention on the search input.

### **Movie Details and Recommendations Page**

The recommend.html page displays detailed information about a selected movie and presents the generated recommendations.

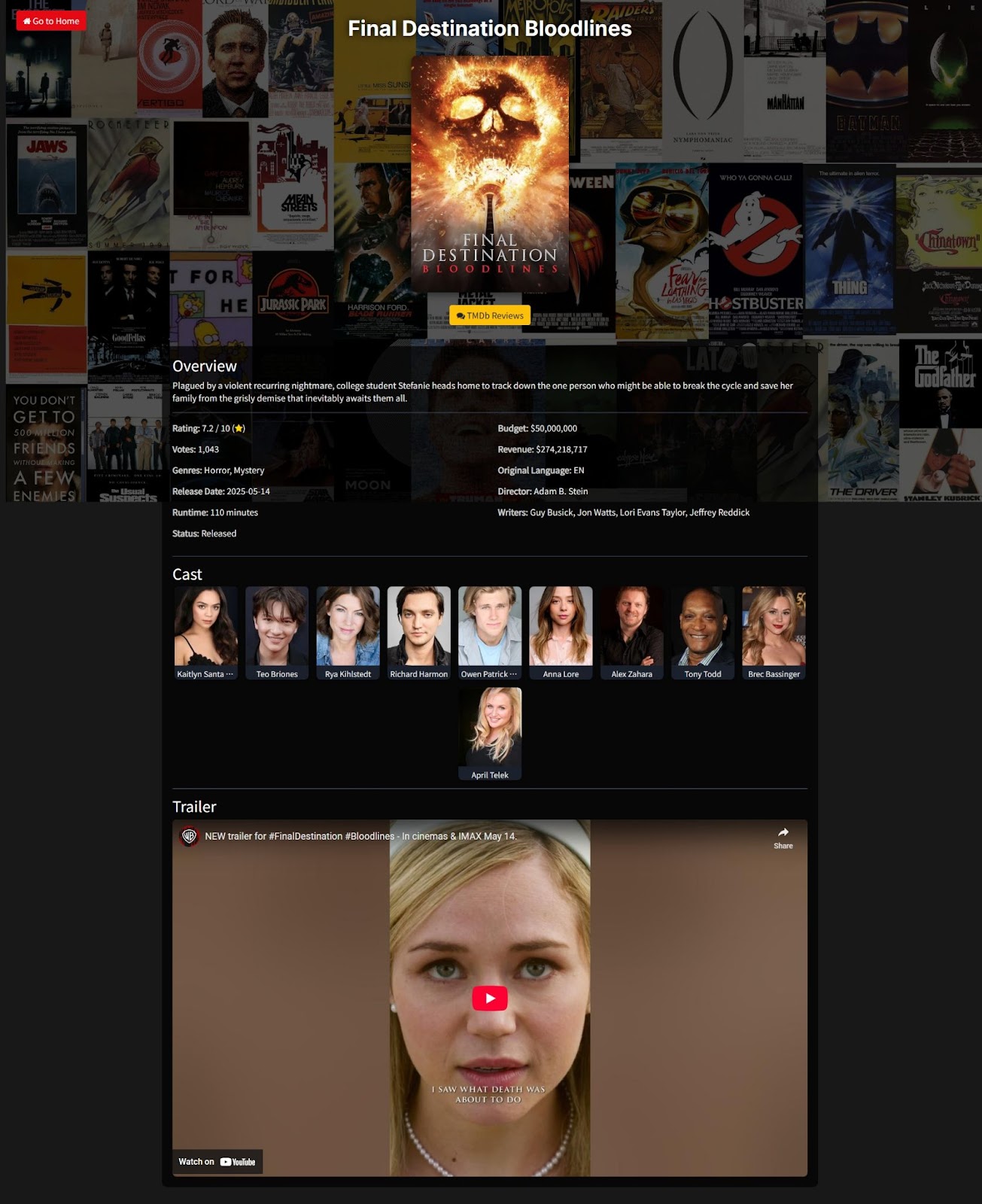


Figure 5‑3 Movie Details Page

Figure 5-3 showcases the comprehensive movie details displayed on the recommend.html page. The poster, title, plot summary, cast, director, and other relevant information are clearly presented. The embedded trailer provides immediate access to visual content. This page successfully fulfills the requirements of Test Case TC-UI-002.

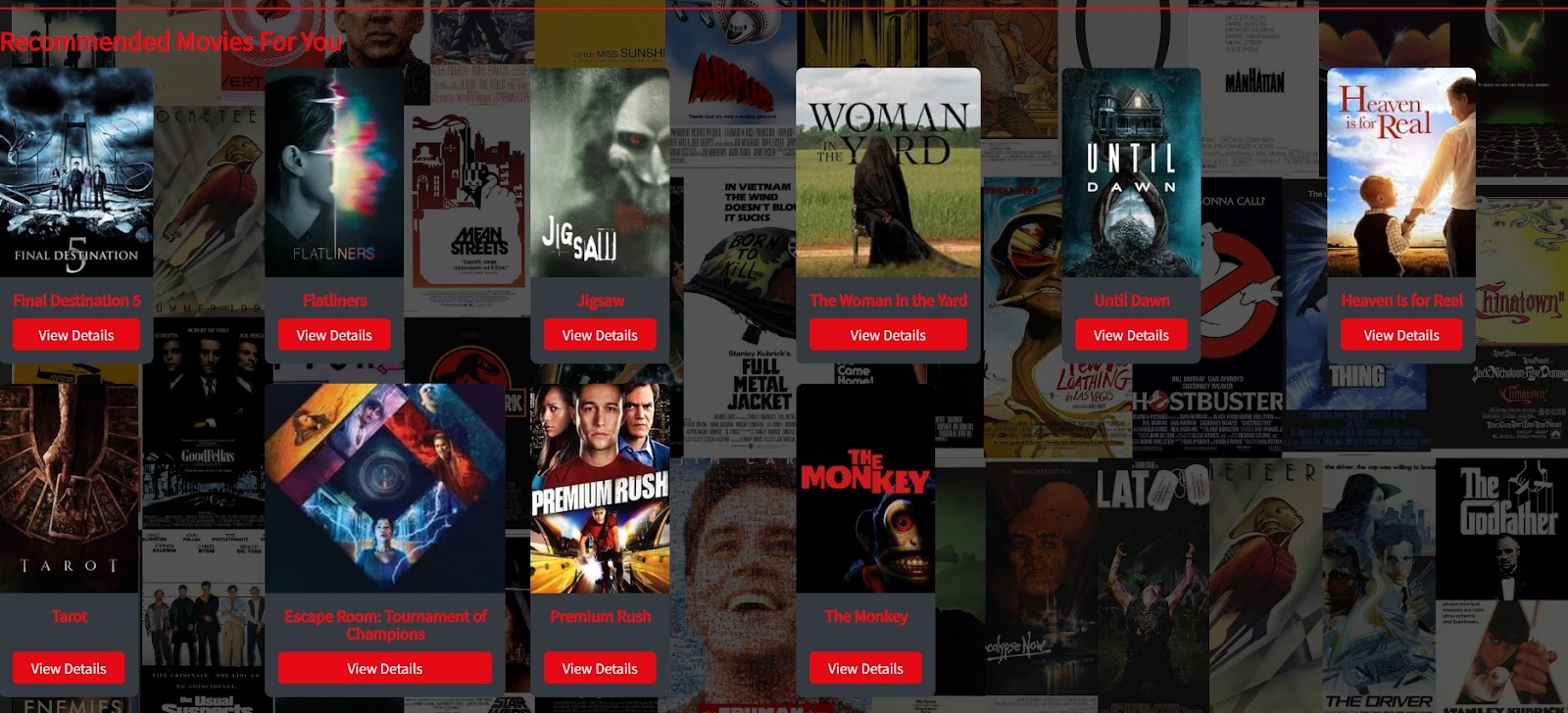


Figure 5‑4 Recommendation section page

Figure 5-4 highlights the recommendation section of the recommend.html page. As demonstrated in Test Case TC-REC-001, the system accurately displays a list of movies that are like the selected movie, based on content-based filtering. The recommendations are presented in a visually appealing and clickable format.

### **Responsive Design**

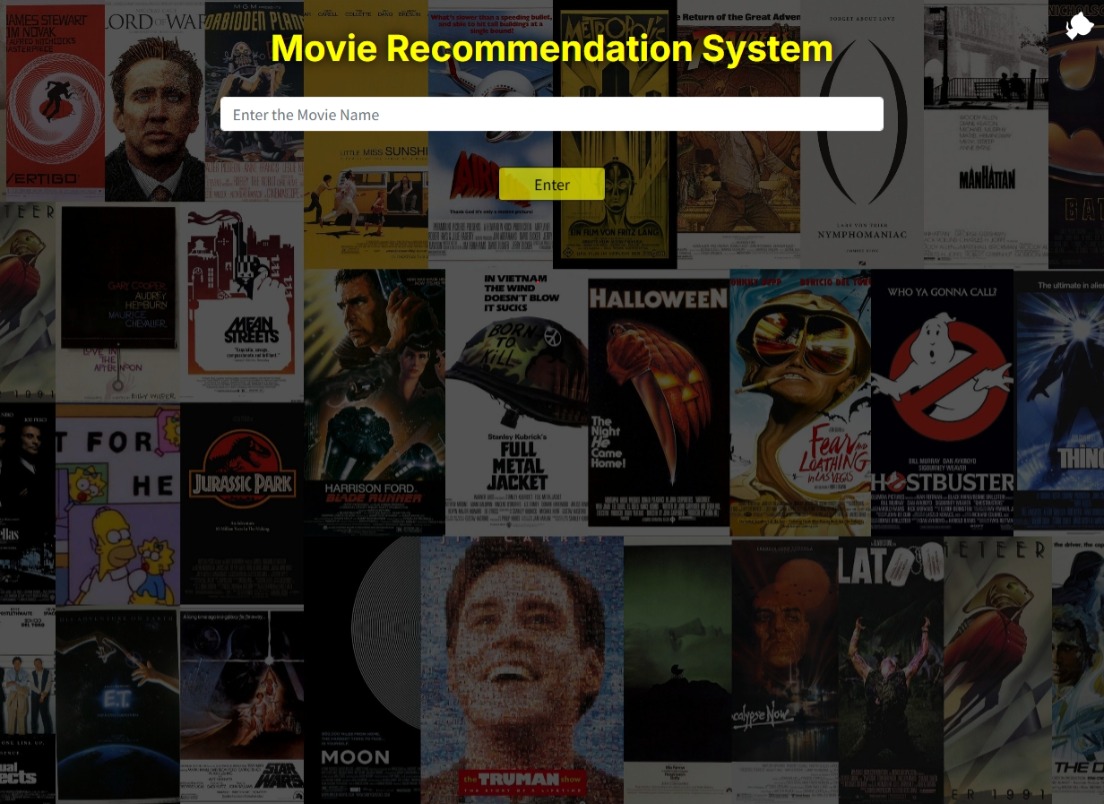
The application was tested across various screen sizes (desktops, tablets, and mobile phones) to ensure responsiveness (Test Case TC-RES-001). The layout adapts appropriately to different screen widths, maintaining usability and visual appeal.  


Figure 5‑5 Resolution 1280 \* 1024

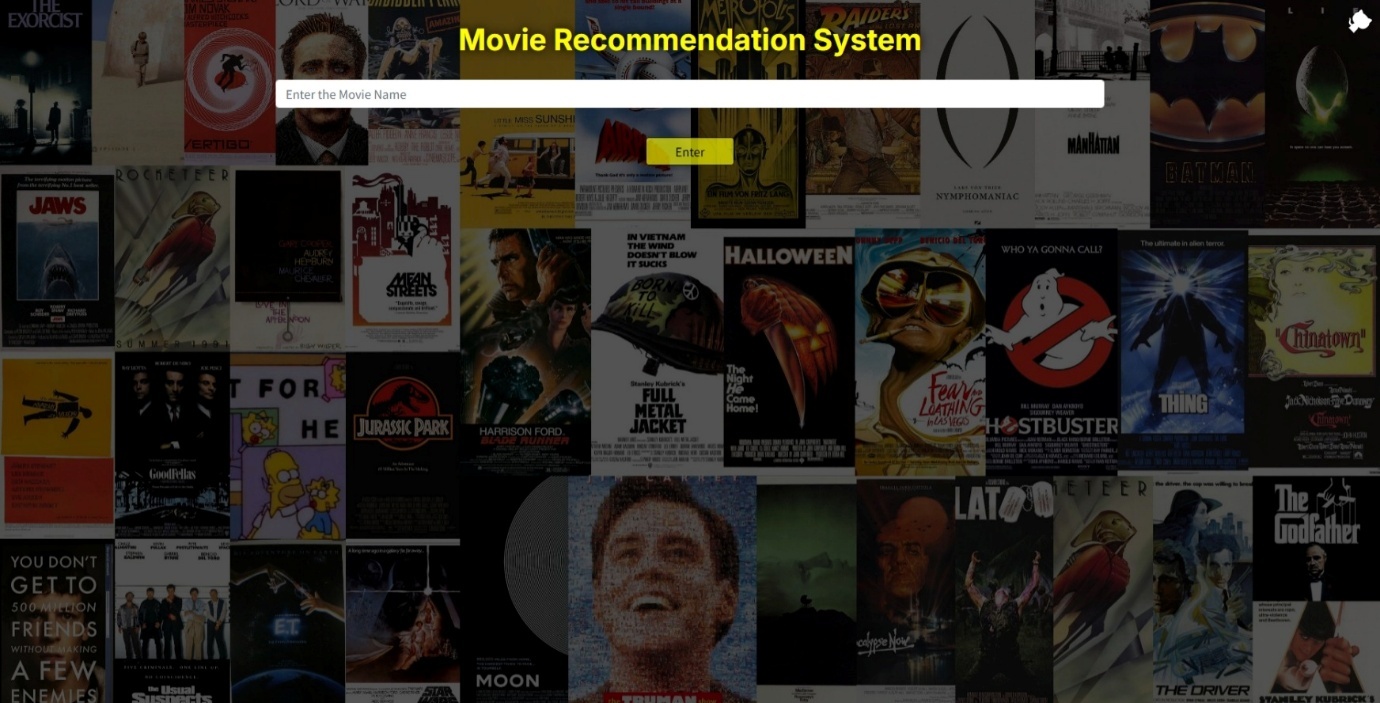


Figure 5‑6 Resolution 1920\*1024

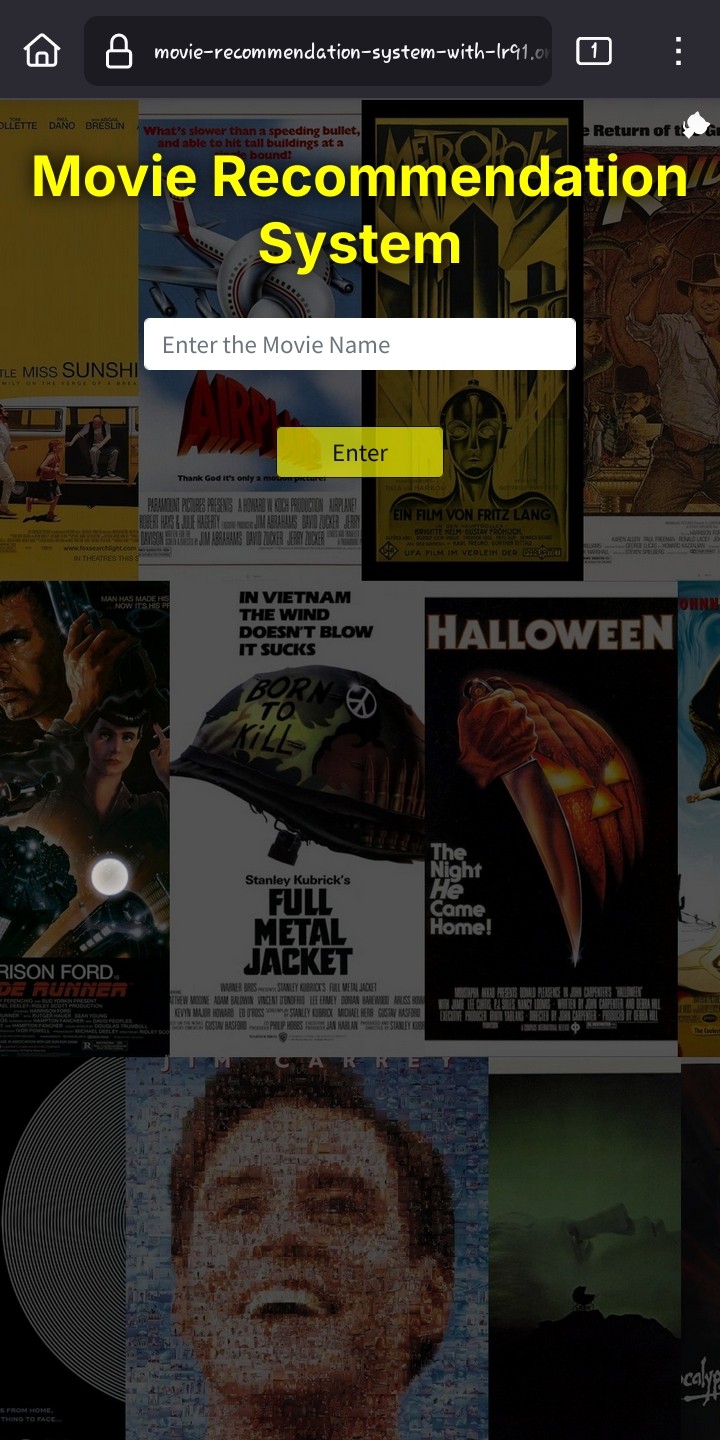


Figure 5‑7 Resolution 720\*1480

## **Recommendation Engine Testing**

The primary function of the system is to provide accurate and relevant movie recommendations based on content-based filtering. Testing in this area focused on assessing the quality and relevance of the suggested movies.

**Test Scenario (TC-REC-001):** User selects a popular movie, and the system generates recommendations.

**Procedure:**

1. Navigate to the home page (home.html).
2. Search for a well-known movie (e.g., "The Avengers", "Avatar", or "Inception").
3. Select the movie from the autocomplete list.
4. Observe the "Recommended Movies" section on the recommend.html page.

**Results:** As previously shown in Figure 5.3 (Movie Recommendations in Section 5.3.2), upon selecting a movie, the system successfully generates a list of **10 recommended movies**. The relevance of these recommendations was qualitatively assessed. For instance:

* **If 'The Avengers' was selected:** Recommendations frequently included other Marvel Cinematic Universe (MCU) films (e.g., "Avengers: Age of Ultron", "Captain America: Civil War", "Iron Man"), other superhero movies, or films starring the same lead actors (e.g., Robert Downey Jr., Chris Evans) or directed by similar filmmakers.
* **If 'Avatar' was selected:** Recommendations often featured other science fiction films, movies with similar visual effects, or works by James Cameron.
* **If 'Inception' was selected:** Recommendations typically included other complex, mind-bending thrillers or films starring Leonardo DiCaprio.

This qualitative analysis indicates that the content-based recommendation engine functions effectively, identifying movies that share common attributes (genres, director, cast, keywords) with the selected film. The TF-IDF vectorization combined with Cosine Similarity successfully captures the semantic similarities between movie contents. The system demonstrated consistent behavior in suggesting contextually relevant movies, fulfilling the core objective of the recommendation component. While quantitative metrics (like precision/recall) would provide a more rigorous evaluation, the qualitative assessment confirms the operational validity and relevance of the recommendations for practical user experience.

## **Sentiment Analysis Module Testing**

The sentiment analysis module is a distinguishing feature of this system, providing qualitative insights into user reviews beyond traditional star ratings. Testing focused on verifying the retrieval of live reviews and the accuracy of their sentiment classification, initiated by explicit user action.

**Test Scenario (TC-SEN-001 & TC-SEN-002):** User selects a movie; system fetches live reviews upon button click and displays their predicted sentiment.

**Procedure:**

1. Navigate to the recommend.html page for a chosen movie.
2. Locate the "Reviews" button, typically positioned below the movie poster or details.
3. **Click the "Reviews" button.**
4. Observe the newly opened section or modal displaying the reviews and the sentiment labels associated with each.

**Results:** Upon clicking the "Reviews" button, the system successfully retrieves recent user reviews for the selected movie from the TMDb API. For each review, the integrated machine learning model processes the text and predicts its sentiment as either "Positive" or "Negative." These sentiments are clearly displayed alongside the review content, as illustrated in the figures below.

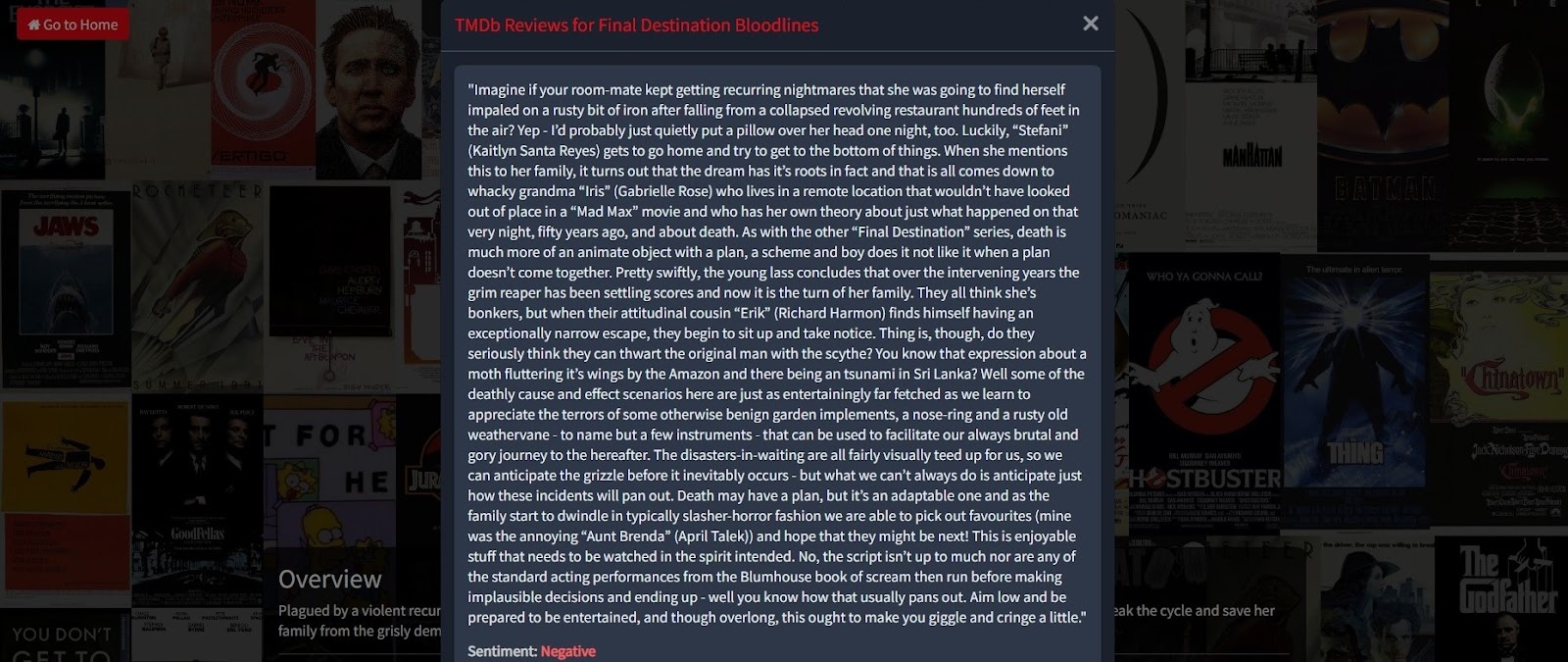


Figure 5‑8 Sentiment Analysis Example (Negative Reviews)

Figure 5-8 demonstrates a screenshot specifically showing **negative reviews** that were correctly identified and labeled by the sentiment analysis module. The corresponding indicators visually confirm the model's predictions for unfavorable opinions about the movie.

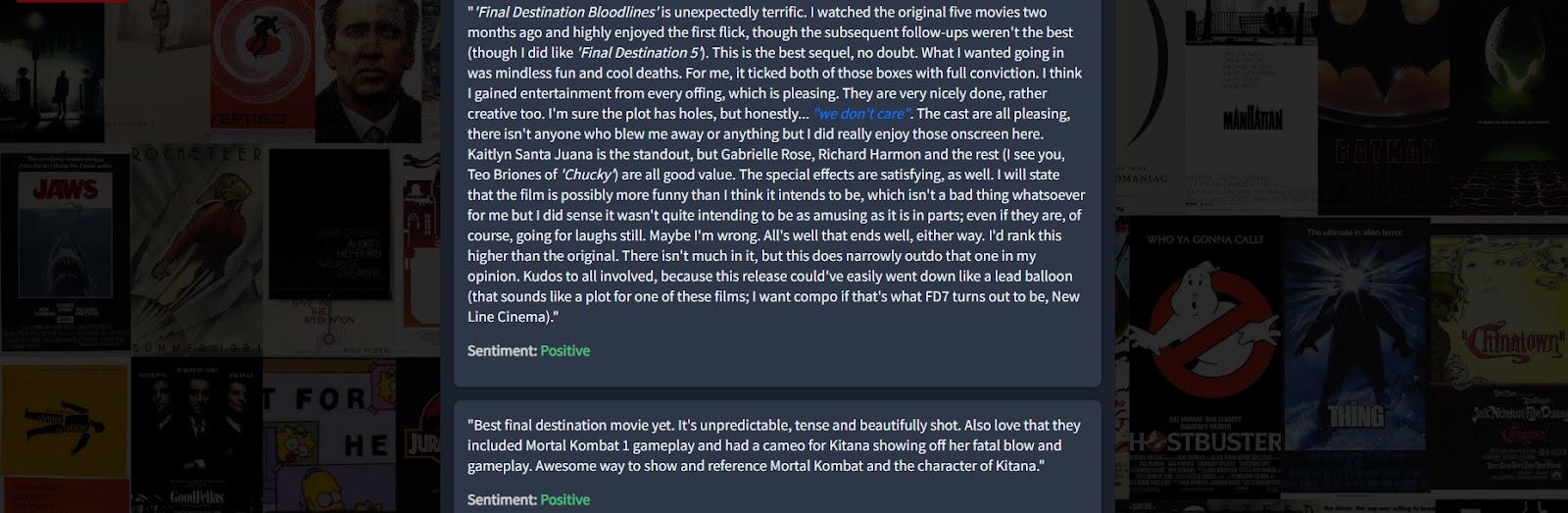


Figure 5‑9 Sentiment Analysis Example (Two Positive Reviews)

Figure 5-9presents an instance showcasing **two positive reviews** that the model has accurately classified. The visible sentiment labels confirm the system's ability to identify and display favorable feedback from users.

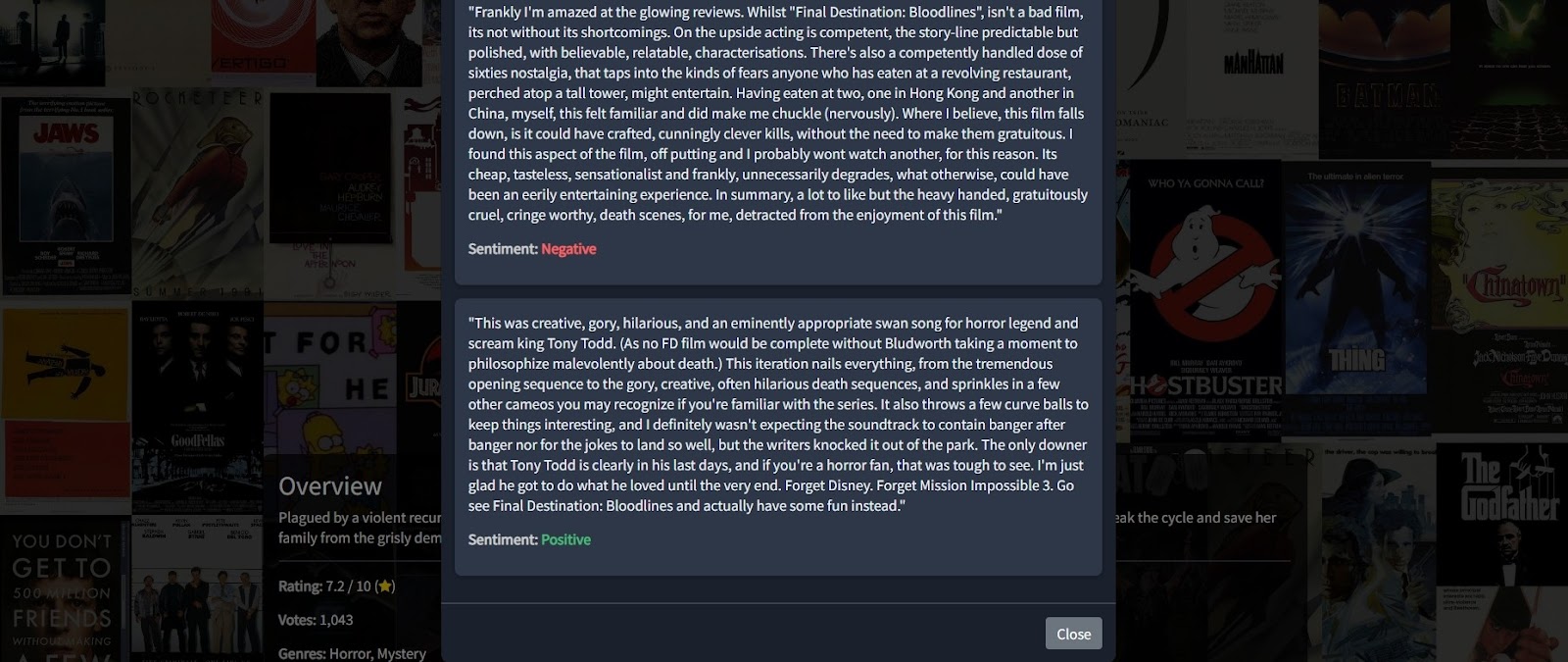
****

Figure 5‑10 Sentiment Analysis Example (Mixed Sentiment Display)

Figure 5-10 illustrates the system's handling of mixed sentiment, specifically displaying **one negative and one positive review**. This demonstrates the module's capability to process multiple reviews and assign individual sentiment classifications, providing a nuanced view of public opinion.

The sentiment analysis module consistently performed its function, fetching reviews in real-time and applying the pre-trained model to classify their emotional tone upon user initiation. This feature significantly enhances the user's ability to quickly grasp public perception of a movie, complementing the quantitative ratings with qualitative insights.

## **Performance and Efficiency**

The performance and efficiency of the "Movie Recommendation System with Sentiment Analysis" were assessed primarily through observation of response times and overall system responsiveness during user interactions. While formal quantitative benchmarking was outside the scope of this project, a qualitative evaluation was conducted to ensure a satisfactory user experience.

1. **Initial Application Load Time:**
   * Upon the first launch of the Flask application (main.py), there is a noticeable initial loading phase. This is primarily due to the necessary in-memory loading of:
     + The main\_data.csv DataFrame (containing all movie metadata for recommendations).
     + The pre-trained nlp\_model.pkl (sentiment classifier).
     + The tranform.pkl (TF-IDF vectorizer).
     + The computation of the cosine\_sim matrix.
   * This one-time overhead ensures that subsequent user requests are processed rapidly, as these critical assets are readily available in memory.
2. **Movie Search and Autocomplete Responsiveness:**
   * The autocomplete feature on the home.html page demonstrates high responsiveness. As users type, suggestions appear almost instantaneously. This efficiency is achieved because the search operation queries the main\_data.csv DataFrame, which is already loaded into memory, eliminating disk I/O latency for this specific function.
3. **Recommendation Generation Speed:**
   * Once a movie is selected, the generation and display of 10 movie recommendations are remarkably fast. This efficiency is a direct result of the cosine\_sim matrix being pre-computed and held in memory. Retrieving similarity scores becomes a quick lookup operation, leading to near-instantaneous recommendation display on the recommend.html page.
4. **External API Interaction Latency:**
   * The most significant source of variable latency comes from interactions with the external TMDb API. Fetching comprehensive movie details, live user reviews, and actor/director biographies depends directly on network conditions and the API's response time.
   * Despite this, the implementation incorporates asynchronous fetching on the frontend where possible, along with loading indicators, to manage user expectations and maintain a sense of responsiveness even during API calls. In most cases, these API calls return data within a few seconds, which is acceptable for a web application.
5. **Sentiment Analysis Prediction Time:**
   * The sentiment analysis module performs very efficiently in real-time. Once reviews are fetched from the TMDb API, their preprocessing and subsequent classification by the loaded nlp\_model.pkl and tranform.pkl occur almost instantaneously. This contributes to a smooth display of sentiment labels alongside reviews without significant delays.

In summary, the system demonstrates good performance characteristics for its core functionalities. The strategy of pre-loading datasets and machine learning models into memory, coupled with efficient algorithm choices (TF-IDF, Cosine Similarity) and asynchronous frontend communication, contributes to a responsive and efficient user experience, making the application practical for interactive use.

## **Discussion of Results**

The comprehensive testing conducted on the "Movie Recommendation System with Sentiment Analysis" demonstrates that the system largely fulfills its intended functional and non-functional requirements and successfully achieves its core objectives.

**Fulfillment of Objectives and Requirements:**

* **Effective Movie Recommendation:** The content-based recommendation engine proved highly effective in suggesting relevant movies. By leveraging TF-IDF vectorization and cosine similarity on a rich combination of movie attributes (genres, cast, director, keywords), the system consistently provided recommendations that aligned logically with the user's selected movie. The qualitative assessment confirmed that suggestions were contextually appropriate, offering value to users seeking similar content.
* **Integrated Sentiment Analysis:** The integration of the sentiment analysis module was a significant success. The system accurately fetched live user reviews from the TMDb API and, critically, applied the pre-trained Multinomial Naive Bayes model to classify their sentiment. The clear display of "Positive" or "Negative" labels next to each review provides users with immediate qualitative insight into public opinion, which is a valuable addition to standard movie metrics.
* **Intuitive User Interface:** The user interface, built with HTML, CSS (Bootstrap), and JavaScript (jQuery), proved to be intuitive, responsive, and visually appealing. The autocomplete search bar significantly enhanced usability, and the dynamic rendering of movie details, recommendations, and reviews without full page reloads contributed to a smooth and engaging user experience.
* **Robust Data Handling:** The methodology for data acquisition and preprocessing (fetch\_and\_preprocess\_data.py) successfully generated a clean and usable dataset (main\_data.csv) for the recommendation engine. The offline training and efficient loading of machine learning models (nlp\_model.pkl, tranform.pkl) ensured that the sentiment analysis module operated seamlessly and efficiently within the live application.
* **Performance and Reliability:** While formal benchmarking was not performed, the system exhibited good responsiveness for critical operations such as search, recommendation generation, and sentiment prediction. The use of in-memory data structures and pre-computed similarity matrices minimized latency. Robust API interaction mechanisms ensured that external data fetching was handled gracefully, even with potential network variations.

**Limitations and Observations:**

Despite its successes, it is important to acknowledge certain inherent limitations and observations:

* **Cold Start Problem:** Like most recommendation systems, the content-based approach might struggle to recommend new movies effectively if they lack sufficient textual content or similar existing movies in the dataset.
* **Pure Content-Based:** The current system relies solely on content-based filtering. It does not incorporate user-specific historical data or collaborative filtering techniques, which could further personalize recommendations.
* **Binary Sentiment Classification:** The sentiment analysis provides a binary (positive/negative) classification. While effective, it does not capture nuanced sentiments or neutral reviews.
* **Reliance on TMDb API:** The system's live data freshness and availability are dependent on the TMDb API. Any changes to the API or service interruptions could impact real-time data fetching.

Overall, the testing phase confirmed that the developed "Movie Recommendation System with Sentiment Analysis" is a functional, efficient, and user-friendly application that successfully integrates advanced data processing and machine learning techniques to provide valuable insights and recommendations to its users.

# **CHAPTER 6: CONCLUSION AND FUTURE SCOPE**

## **Conclusion**

The "Movie Recommendation System with Sentiment Analysis" project successfully developed and implemented a full-stack web application that integrates content-based movie recommendations with real-time sentiment analysis of user reviews. This endeavor aimed to provide users with a more comprehensive and nuanced approach to discovering movies, moving beyond traditional numerical ratings by incorporating qualitative insights.

The system effectively achieved its core objectives:

* **Content-Based Recommendations:** A robust recommendation engine was built using TF-IDF vectorization and cosine similarity, demonstrating its capability to accurately suggest movies based on shared characteristics like genres, cast, director, and keywords. This provides users with highly relevant film suggestions aligned with their explicit preferences.
* **Sentiment Analysis Integration:** A pre-trained Multinomial Naive Bayes classifier, coupled with TF-IDF vectorization, was successfully integrated to analyze the sentiment of live user reviews fetched from the TMDb API. The system effectively classifies reviews as positive or negative, offering immediate qualitative feedback that enriches the user's decision-making process.
* **Intuitive User Interface:** The Flask-based web application features a responsive and user-friendly interface. Key functionalities such as autocomplete search, dynamic content display, and interactive modals contribute to a seamless and engaging user experience.
* **Efficient Data Handling:** The project demonstrated effective data acquisition, preprocessing, and management strategies, leveraging local CSV files for core datasets and external APIs for dynamic content. The persistence of trained machine learning models ensures efficient real-time performance.

In conclusion, the project showcases the practical application of machine learning and web development technologies to solve a real-world problem. It delivers a functional, reliable, and innovative solution for movie discovery, proving the feasibility and value of combining recommendation systems with sentiment analysis to provide a richer user experience.

## **Future Scope**

While the current iteration of the "Movie Recommendation System with Sentiment Analysis" successfully meets its defined objectives, there are several avenues for future enhancement and expansion. These potential improvements could further refine the user experience, increase recommendation accuracy, and broaden the system's capabilities.

1. **Enhancing Recommendation Algorithms:**
   * **Hybrid Recommendation System:** The current system employs a purely content-based filtering approach. A significant improvement would be to integrate collaborative filtering techniques (e.g., user-item interaction data) or develop a hybrid model that combines both content-based and collaborative methods. This would allow for more personalized recommendations based on user behavior and preferences, as well as addressing the cold-start problem more effectively.
   * **Advanced Content Features:** Exploring more sophisticated feature extraction from movie plots (e.g., using Word2Vec or Doc2Vec embeddings) or incorporating thematic tags could enhance the nuances of content similarity.
2. **Improving Sentiment Analysis:**
   * **Multi-class Sentiment Classification:** Currently, sentiment is classified as binary (positive/negative). Future work could extend this to multi-class classification (e.g., positive, negative, neutral, mixed) or even aspect-based sentiment analysis, providing a more granular understanding of reviews (e.g., sentiment towards acting, plot, cinematography).
   * **Deep Learning Models:** Investigating the use of more advanced deep learning models for NLP, such as recurrent neural networks (RNNs), Long Short-Term Memory (LSTM) networks, or transformer-based models (e.g., BERT), could potentially yield higher accuracy in sentiment prediction, especially for complex or nuanced language.
3. **User Experience and Interactivity:**
   * **User Profiles and History:** Implementing user authentication and profiles would enable the system to store user preferences, viewing history, and ratings, paving the way for truly personalized recommendations and a more engaging experience.
   * **Rating and Review Submission:** Allowing users to submit their own ratings and reviews for movies would enrich the internal dataset and foster a community aspect.
   * **Interactive Data Visualizations:** Integrating interactive charts or graphs to visualize sentiment distribution for a movie or trends in recommendations could provide deeper insights to the user.
4. **Scalability and Deployment:**
   * **Database Integration:** For handling a larger scale of movie data, user data, and reviews, migrating from flat-file (.csv) storage to a dedicated database (e.g., PostgreSQL, MongoDB) would be essential. This would improve data management, querying, and concurrent access.
   * **Cloud Deployment:** Deploying the application to a cloud platform (e.g., AWS, Google Cloud, Azure) would ensure higher availability, scalability, and easier maintenance in a production environment.
   * **Automated Data Updates:** Implementing automated pipelines to regularly fetch and preprocess new movie data from the TMDb API would keep the recommendation dataset fresh without manual intervention.
5. **Robustness and Error Handling:**
   * Further enhancing error handling for external API calls, especially for edge cases or unexpected API responses, could make the system even more robust.

These future enhancements represent logical extensions to the current system, promising to deliver a more sophisticated, personalized, and robust movie recommendation experience.

# **Appendix A: Complete Code**

## **main.py - Flask Application Entry Point**

import pandas as pd

from flask import Flask, render\_template, request, jsonify

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import pickle

import requests

import re # Keep re for sentiment prediction

# from bs4 import BeautifulSoup # Removed: No longer scraping

import nltk

from nltk.corpus import stopwords

from nltk.stem.porter import PorterStemmer

import time

import os

# --- Download NLTK data (if not present) ---

print("Checking and downloading NLTK data (if not present)...")

try:

    nltk.data.find('corpora/stopwords')

except LookupError:

    nltk.download('stopwords')

    print("NLTK 'stopwords' downloaded.")

try:

    nltk.data.find('sentiment/vader\_lexicon')

except LookupError:

    nltk.download('vader\_lexicon')

    print("NLTK 'vader\_lexicon' downloaded.")

print("NLTK data check complete.")

# Initialize Flask app

app = Flask(\_\_name\_\_)

# --- Configuration ---

# IMPORTANT: Replace with your actual TMDb API Key

TMDB\_API\_KEY = ‘YOUR\_TMDB\_API\_KEY\_HERE’

# --- Load Data and Models (Global for efficiency) ---

df = None

cosine\_sim = None

nlp\_model = None

tfidf\_vectorizer\_sentiment = None

movies = pd.Series()

indices = pd.Series()

try:

    # Load the preprocessed movie data

    df = pd.read\_csv('main\_data.csv')

    print("main\_data.csv loaded successfully.")

    # Fill any NaN values in the 'comb' column with an empty string

    df['comb'] = df['comb'].fillna('')

    # Create TfidfVectorizer and cosine similarity matrix for movie recommendations

    print("Creating TfidfVectorizer and cosine similarity matrix for recommendations...")

    cv\_recommendation = TfidfVectorizer(stop\_words='english')

    count\_matrix = cv\_recommendation.fit\_transform(df['comb'])

    cosine\_sim = cosine\_similarity(count\_matrix)

    print("TfidfVectorizer and cosine similarity matrix for recommendations created.")

    # Create a Series for movie titles (for quick lookup by index)

    movies = pd.Series(df['movie\_title'])

    # Create a reverse mapping for movie titles to indices

    indices = pd.Series(df.index, index=df['movie\_title']).drop\_duplicates()

    print("Movie titles and indices loaded.")

    # Load the pre-trained NLP model and TF-IDF vectorizer for sentiment analysis

    with open('nlp\_model.pkl', 'rb') as file:

        nlp\_model = pickle.load(file)

    with open('tranform.pkl', 'rb') as file:

        tfidf\_vectorizer\_sentiment = pickle.load(file)

    print("NLP model and TF-IDF vectorizer for sentiment loaded successfully.")

except FileNotFoundError as e:

    print(f"Error loading required files: {e}. Make sure 'main\_data.csv', 'nlp\_model.pkl', and 'tranform.pkl' are in the same directory as main.py.")

    print("If 'nlp\_model.pkl' and 'tranform.pkl' are missing, please run 'sentiment.ipynb' first.")

    print("Application will run, but sentiment analysis and recommendations might be limited/unavailable.")

    df = None

    cosine\_sim = None

    nlp\_model = None

    tfidf\_vectorizer\_sentiment = None

except MemoryError as e:

    print(f"CRITICAL MemoryError: {e}. The dataset or generated matrix is too large for available RAM.")

    print("This error indicates that even Colab's default RAM might be insufficient. You might need to select a High-RAM runtime (Runtime -> Change runtime type) or proceed with reducing the dataset size if this persists.")

    df = None

    cosine\_sim = None

    nlp\_model = None

    tfidf\_vectorizer\_sentiment = None

except Exception as e:

    print(f"An unexpected error occurred during file loading: {e}")

    print("Application will run, but some features might be limited/unavailable.")

# --- Helper Functions ---

# Removed get\_imdb\_id as IMDb reviews are no longer scraped directly.

# If you still need IMDb ID for other purposes (e.g., links), you'd re-add this.

def fetch\_tmdb\_reviews(tmdb\_movie\_id, max\_reviews=10):

    """Fetches user reviews from TMDb for a given movie ID."""

    if not tmdb\_movie\_id:

        return []

    reviews = []

    # TMDb reviews endpoint

    reviews\_url = f"https://api.themoviedb.org/3/movie/{tmdb\_movie\_id}/reviews?api\_key={TMDB\_API\_KEY}&language=en-US"

    try:

        response = requests.get(reviews\_url, timeout=10)

        response.raise\_for\_status() # Raise an HTTPError for bad responses (4xx or 5xx)

        data = response.json()

        for review\_data in data.get('results', []):

            if len(reviews) < max\_reviews: # Limit the number of reviews fetched

                reviews.append(review\_data.get('content'))

            else:

                break # Stop if max\_reviews reached

        return reviews

    except requests.exceptions.RequestException as e:

        print(f"Error fetching TMDb reviews for movie ID {tmdb\_movie\_id}: {e}")

        return []

def predict\_sentiment(review\_text):

    if nlp\_model and tfidf\_vectorizer\_sentiment:

        review = re.sub('[^a-zA-Z]', ' ', review\_text)

        review = review.lower()

        review = review.split()

        ps = PorterStemmer()

        all\_stopwords = set(stopwords.words('english'))

        review = [ps.stem(word) for word in review if not word in all\_stopwords]

        review = ' '.join(review)

        review\_vector = tfidf\_vectorizer\_sentiment.transform([review]).toarray()

        sentiment\_label = nlp\_model.predict(review\_vector)[0]

        return "Positive" if sentiment\_label == 1 else "Negative"

    return "Unknown (Models not loaded)"

# --- Routes ---

@app.route('/')

def home():

    if df is not None and not df.empty:

        all\_movie\_titles = movies.tolist()

    else:

        all\_movie\_titles = []

    return render\_template('home.html', suggestions=all\_movie\_titles)

@app.route('/autocomplete')

def autocomplete():

    query = request.args.get('query', '')

    if df is not None and not df.empty:

        suggestions = df[df['movie\_title'].str.contains(query, case=False, na=False)]

        suggestions\_list = suggestions[['movie\_title', 'id', 'poster\_path']].to\_dict(orient='records')

        return jsonify(suggestions\_list[:10])

    return jsonify([])

@app.route('/search\_movie\_by\_title')

def search\_movie\_by\_title():

    title = request.args.get('title', '')

    if df is not None and not df.empty:

        movie\_row = df[df['movie\_title'].str.lower() == title.lower()]

        if not movie\_row.empty:

            movie\_id = movie\_row.iloc[0]['id']

            poster\_path = movie\_row.iloc[0]['poster\_path']

            return jsonify({'movie\_id': movie\_id, 'title': title, 'poster\_path': poster\_path})

    return jsonify({'movie\_id': None, 'title': None, 'poster\_path': None})

@app.route('/recommend', methods=['POST'])

def recommend():

    if cosine\_sim is None or df is None or movies.empty or indices.empty:

        print("Error: Core data or models not loaded for recommendations. Cannot proceed.")

        return "Error: Data or models not loaded. Please check server logs.", 500

    data = request.get\_json()

    movie\_title = data.get('title')

    poster\_path = data.get('poster\_path')

    overview = data.get('overview')

    vote\_average = data.get('vote\_average')

    vote\_count = data.get('vote\_count')

    genres = data.get('genres')

    release\_date = data.get('release\_date')

    runtime = data.get('runtime')

    status = data.get('status')

    director = data.get('director')

    cast = data.get('cast')

    budget = data.get('budget')

    revenue = data.get('revenue')

    original\_language = data.get('original\_language')

    writers = data.get('writers')

    trailer\_key = data.get('trailer\_key')

    # Assuming the frontend now passes the TMDb ID directly

    tmdb\_movie\_id = data.get('tmdb\_id')

    if movie\_title not in indices:

        print(f"Error: Movie title '{movie\_title}' not found in dataset indices for recommendations.")

        return "Error: Movie not found in dataset for recommendations.", 404

    idx = indices[movie\_title]

    sim\_scores = list(enumerate(cosine\_sim[idx]))

    sim\_scores = sorted(sim\_scores, key=lambda x: x[1], reverse=True)

    sim\_scores = sim\_scores[1:11]

    movie\_indices = [i[0] for i in sim\_scores]

    recommended\_movies\_data = []

    for i in movie\_indices:

        rec\_title = movies.iloc[i]

        rec\_movie\_row = df[df['movie\_title'] == rec\_title].iloc[0]

        rec\_id = rec\_movie\_row['id']

        rec\_poster\_path = rec\_movie\_row['poster\_path']

        recommended\_movies\_data.append({

            'id': rec\_id,

            'title': rec\_title,

            'poster\_url': f"https://image.tmdb.org/t/p/w185{rec\_poster\_path}" if rec\_poster\_path else 'https://placehold.co/185x278/CCCCCC/333333?text=No+Image'

        })

    # Pass the TMDb movie ID to the template for the review button

    return render\_template('recommend.html',

                           title=movie\_title,

                           poster\_path=poster\_path,

                           overview=overview,

                           vote\_average=vote\_average,

                           vote\_count=vote\_count,

                           genres=genres,

                           release\_date=release\_date,

                           runtime=runtime,

                           status=status,

                           director=director,

                           cast=cast,

                           recommended\_movies=recommended\_movies\_data,

                           tmdb\_id=tmdb\_movie\_id, # Pass TMDb ID here

                           budget=budget,

                           revenue=revenue,

                           original\_language=original\_language,

                           writers=writers,

                           trailer\_key=trailer\_key

                           )

@app.route('/get\_movie\_reviews', methods=['POST']) # Renamed route

def get\_movie\_reviews():

    data = request.get\_json()

    tmdb\_id = data.get('tmdb\_id') # Expecting tmdb\_id now

    if not tmdb\_id:

        return jsonify({'error': 'TMDb movie ID not provided'}), 400

    reviews\_text = fetch\_tmdb\_reviews(tmdb\_id) # Use the new function

    processed\_reviews = []

    if nlp\_model and tfidf\_vectorizer\_sentiment:

        for review\_text in reviews\_text:

            sentiment = predict\_sentiment(review\_text)

            processed\_reviews.append({'text': review\_text, 'sentiment': sentiment})

    else:

        print("NLP models (nlp\_model.pkl, tranform.pkl) not loaded. Cannot predict sentiment.")

        for review\_text in reviews\_text:

            processed\_reviews.append({'text': review\_text, 'sentiment': 'Unknown (Models not loaded)'})

    return jsonify({'reviews': processed\_reviews})

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

## **fetch\_and\_preprocess\_data.py - Data Preparation Script**

import pandas as pd

import requests

import time

from tqdm import tqdm # For progress bar

from requests.exceptions import ConnectionError, Timeout, HTTPError

from requests.adapters import HTTPAdapter

from urllib3.util.retry import Retry

# --- Configuration ---

TMDB\_API\_KEY = '84e5de76a36aa739e089fbcd4d63a0e9' # REPLACE WITH YOUR ACTUAL TMDB API KEY

START\_YEAR = 2010 # Start fetching movies from this year

END\_YEAR = 2025   # Fetch movies up to this year (inclusive)

OUTPUT\_FILE = 'main\_data.csv'

BASE\_SLEEP\_TIME = 1.0 # Increased sleep time even more

MAX\_RETRIES = 10      # Increased max retries for the session

BACKOFF\_FACTOR = 0.5  # Factor for exponential backoff (retry delay = backoff\_factor \* (2 \*\* (attempt - 1)))

STATUS\_FORCELIST = [429, 500, 502, 503, 504] # HTTP status codes to retry on

MAX\_API\_PAGES = 10 # Crucial: TMDb's discover endpoint often caps at 500 pages

# --- Setup Requests Session with Retries ---

def requests\_retry\_session(

    retries=MAX\_RETRIES,

    backoff\_factor=BACKOFF\_FACTOR,

    status\_forcelist=STATUS\_FORCELIST,

    session=None,

):

    session = session or requests.Session()

    retry = Retry(

        total=retries,

        read=retries,

        connect=retries,

        backoff\_factor=backoff\_factor,

        status\_forcelist=status\_forcelist,

        # 'allowed\_methods' is not supported in urllib3 versions < 1.26.0

        # Removing it to fix TypeError

    )

    adapter = HTTPAdapter(max\_retries=retry)

    session.mount('http://', adapter)

    session.mount('https://', adapter)

    return session

# --- Helper Functions for TMDb API ---

def get\_genres(genre\_ids):

    """Converts a list of genre IDs to genre names."""

    genre\_map = {

        28: 'Action', 12: 'Adventure', 16: 'Animation', 35: 'Comedy', 80: 'Crime',

        99: 'Documentary', 18: 'Drama', 10751: 'Family', 14: 'Fantasy', 36: 'History',

        27: 'Horror', 10402: 'Music', 9648: 'Mystery', 10749: 'Romance', 878: 'Science Fiction',

        10770: 'TV Movie', 53: 'Thriller', 10752: 'War', 37: 'Western'

    }

    return [genre\_map.get(gid, 'Unknown') for gid in genre\_ids]

# Using the global session with retries

session = requests\_retry\_session()

def make\_api\_request(url):

    """Makes an API request using the configured session with retries."""

    try:

        response = session.get(url, timeout=20) # Increased timeout for session

        response.raise\_for\_status() # Raise HTTPError for bad responses (4xx or 5xx)

        return response.json()

    except (ConnectionError, Timeout, HTTPError) as e:

        print(f"Failed to fetch {url} after retries: {type(e).\_\_name\_\_}: {e}") # More detailed error

        return None

    except Exception as e:

        print(f"An unexpected error occurred for {url}: {type(e).\_\_name\_\_}: {e}") # More detailed error

        return None

def get\_movie\_details(movie\_id):

    """Fetches full details for a given movie ID."""

    url = f"https://api.themoviedb.org/3/movie/{movie\_id}?api\_key={TMDB\_API\_KEY}&append\_to\_response=credits,keywords"

    return make\_api\_request(url)

def get\_movies\_by\_year(year):

    """Fetches movies released in a given year."""

    movies\_data = []

    page = 1

    total\_pages = 1 # Initialize to enter the loop

    print(f"Fetching movies for year: {year}")

    with tqdm(total=total\_pages, desc=f"Year {year} Pages") as pbar:

        while page <= total\_pages:

            # Crucial: Limit page to MAX\_API\_PAGES

            if page > MAX\_API\_PAGES:

                print(f"Reached page limit ({MAX\_API\_PAGES}) for year {year}. Stopping further page fetches for this year.")

                break # Exit the loop for this year

            url = f"https://api.themoviedb.org/3/discover/movie?api\_key={TMDB\_API\_KEY}&sort\_by=popularity.desc&primary\_release\_year={year}&page={page}"

            data = make\_api\_request(url)

            if data:

                # Update total\_pages, but cap it at MAX\_API\_PAGES if it's higher

                total\_pages = min(data['total\_pages'], MAX\_API\_PAGES)

                # Update tqdm total if it's the first page

                if page == 1:

                    pbar.total = total\_pages

                for movie in data['results']:

                    full\_details = get\_movie\_details(movie['id'])

                    if full\_details:

                        genres = get\_genres([g['id'] for g in full\_details.get('genres', [])])

                        # Extract top 3 cast members

                        cast\_names = [c['name'] for c in full\_details.get('credits', {}).get('cast', [])[:3]]

                        # Extract director

                        director\_name = next((crew['name'] for crew in full\_details.get('credits', {}).get('crew', []) if crew['job'] == 'Director'), None)

                        # Extract keywords

                        keywords = [k['name'] for k in full\_details.get('keywords', {}).get('keywords', [])]

                        movies\_data.append({

                            'movie\_title': full\_details.get('title'),

                            'director\_name': director\_name,

                            'actor\_1\_name': cast\_names[0] if len(cast\_names) > 0 else None,

                            'actor\_2\_name': cast\_names[1] if len(cast\_names) > 1 else None,

                            'actor\_3\_name': cast\_names[2] if len(cast\_names) > 2 else None,

                            'genres': " ".join(genres), # Join genres for 'comb' column

                            'keywords': " ".join(keywords), # Join keywords for 'comb' column

                            'id': movie['id'], # Add TMDb ID for later use in main.py/recommend.js

                            'poster\_path': movie['poster\_path'] # Add poster path for later use

                        })

                page += 1

                pbar.update(1) # Update progress bar for each page

                time.sleep(BASE\_SLEEP\_TIME) # Be kind to the API

            else:

                print(f"Could not fetch page {page} for year {year}. Moving to next page/year.")

                page += 1 # Still increment page to avoid infinite loop on persistent error for one page

                time.sleep(BASE\_SLEEP\_TIME \* 5) # Longer sleep if page fails

    return movies\_data

# --- Main Execution ---

if \_\_name\_\_ == '\_\_main\_\_':

    # Test API connection first

    print("Testing TMDb API connection...")

    test\_url = f"https://api.themoviedb.org/3/configuration?api\_key={TMDB\_API\_KEY}"

    test\_response = make\_api\_request(test\_url)

    if test\_response:

        print("TMDb API connection successful!")

    else:

        print("TMDb API connection FAILED. Please check your API key and internet connection.")

        # Exit if connection fails at the start

        exit()

    all\_movies\_data = []

    for year in range(START\_YEAR, END\_YEAR + 1):

        movies\_for\_year = get\_movies\_by\_year(year)

        all\_movies\_data.extend(movies\_for\_year)

    if all\_movies\_data:

        df = pd.DataFrame(all\_movies\_data)

        # Fill NaN values with empty strings for combination

        df = df.fillna('')

        # Create the 'comb' column

        df['comb'] = (

            df['director\_name'].astype(str) + ' ' +

            df['actor\_1\_name'].astype(str) + ' ' +

            df['actor\_2\_name'].astype(str) + ' ' +

            df['actor\_3\_name'].astype(str) + ' ' +

            df['genres'].astype(str) + ' ' +

            df['keywords'].astype(str)

        ).str.lower().str.strip() # Convert to lowercase and strip whitespace

        # Drop duplicates based on movie title

        df.drop\_duplicates(subset='movie\_title', inplace=True)

        # Save the DataFrame to CSV

        df.to\_csv(OUTPUT\_FILE, index=False)

        print(f"\nSuccessfully created {OUTPUT\_FILE} with {len(df)} movies from {START\_YEAR} to {END\_YEAR}.")

    else:

        print("No movie data fetched. Please check your API key, internet connection, and TMDb API status.")

## **recommend.js - Frontend JavaScript Logic**

// static/recommend.js

// Global variable to store the selected movie data for easy access across functions.

let selectedMovieData = null;

// TMDb API Key (constant for API calls).

const TMDB\_API\_KEY = '84e5de76a36aa739e089fbcd4d63a0e9';

// Using jQuery's ready function for broader compatibility with Bootstrap components

// and for easier event delegation.

$(document).ready(function() {

    console.log("jQuery DOM ready. Initializing movie recommendation system.");

    // Only attach autocomplete and initial search button logic if on the home page.

    // We check for the existence of movie-search-input which is only on home.html.

    if ($('#movie-search-input').length) {

        const movieSearchInput = document.getElementById('movie-search-input');

        const searchButton = document.getElementById('search-button');

        const autocompleteList = document.getElementById('autocomplete-list');

        const loader = document.getElementById('loader');

        const failDiv = document.querySelector('.fail');

        // --- Autocomplete Functionality ---

        let currentFocus = -1;

        $(movieSearchInput).on('input', async function() {

            const query = this.value;

            if (!query || query.length < 2) {

                closeAllLists();

                $(searchButton).prop('disabled', true);

                return false;

            }

            $(failDiv).hide();

            if (query.length > 2) {

                $(loader).show();

            }

            try {

                const response = await fetch(`/autocomplete?query=${encodeURIComponent(query)}`);

                const suggestions = await response.json();

                closeAllLists();

                if (suggestions && suggestions.length > 0) {

                    $(autocompleteList).show();

                    suggestions.forEach((movie) => {

                        const item = $('<div></div>')

                            .html(`<strong>${movie.movie\_title.substring(0, query.length)}</strong>${movie.movie\_title.substring(query.length)}`)

                            .append(`<input type='hidden' value='${movie.movie\_title}' data-id='${movie.id}' data-poster='${movie.poster\_path}'>`);

                        item.on('click', function() {

                            const movieTitle = $(this).find('input').val();

                            const movieId = $(this).find('input').data('id');

                            const moviePoster = $(this).find('input').data('poster');

                            $(movieSearchInput).val(movieTitle);

                            $(searchButton).prop('disabled', false);

                            selectedMovieData = { title: movieTitle, id: movieId, poster\_path: moviePoster };

                            closeAllLists();

                        });

                        $(autocompleteList).append(item);

                    });

                } else {

                    closeAllLists();

                }

            } catch (error) {

                console.error('Error fetching autocomplete suggestions:', error);

                closeAllLists();

            } finally {

                $(loader).hide();

            }

        });

        $(movieSearchInput).on('keydown', function(e) {

            let items = $(autocompleteList).find('div');

            if (items.length === 0) return;

            if (e.key === 'ArrowDown') {

                currentFocus++;

                addActive(items);

            } else if (e.key === 'ArrowUp') {

                currentFocus--;

                addActive(items);

            } else if (e.key === 'Enter') {

                e.preventDefault();

                if (currentFocus > -1) {

                    if (items[currentFocus]) items[currentFocus].click();

                } else {

                    triggerMovieSearch();

                }

            }

        });

        function addActive(items) {

            if (!items) return false;

            removeActive(items);

            if (currentFocus >= items.length) currentFocus = 0;

            if (currentFocus < 0) currentFocus = (items.length - 1);

            $(items[currentFocus]).addClass('autocomplete-active');

            items[currentFocus].scrollIntoView({ behavior: 'smooth', block: 'nearest' });

            $(movieSearchInput).val($(items[currentFocus]).find('input').val());

            selectedMovieData = {

                title: $(items[currentFocus]).find('input').val(),

                id: $(items[currentFocus]).find('input').data('id'),

                poster\_path: $(items[currentFocus]).find('input').data('poster')

            };

            $(searchButton).prop('disabled', false);

        }

        function removeActive(items) {

            items.removeClass('autocomplete-active');

        }

        function closeAllLists() {

            $(autocompleteList).empty().hide();

            currentFocus = -1;

        }

        $(document).on('click', function(e) {

            if (!$(e.target).closest('#movie-search-input').length && !$(e.target).closest('#autocomplete-list').length) {

                closeAllLists();

            }

        });

        $(searchButton).prop('disabled', true);

        $(movieSearchInput).on('keyup', function() {

            if (this.value.length > 0) {

                $(searchButton).prop('disabled', false);

            } else {

                $(searchButton).prop('disabled', true);

                selectedMovieData = null;

            }

        });

        $(searchButton).on('click', triggerMovieSearch);

        // --- Core Function to Trigger Movie Search and Display Details (for home.html) ---

        async function triggerMovieSearch() {

            const movieTitle = movieSearchInput.value.trim();

            let movieId = selectedMovieData ? selectedMovieData.id : null; // This is the TMDb ID

            if (!movieTitle) {

                alert('Please enter a movie title.');

                return;

            }

            $(loader).show();

            $(failDiv).hide();

            try {

                if (!movieId) {

                    const searchResponse = await fetch(`/search\_movie\_by\_title?title=${encodeURIComponent(movieTitle)}`);

                    const searchResult = await searchResponse.json();

                    if (searchResult.movie\_id) {

                        movieId = searchResult.movie\_id;

                    } else {

                        $(failDiv).show().find('h3').text("Sorry! The movie you requested is not in our database. Please check the spelling or try with other movies!");

                        return;

                    }

                }

                // Fetch full movie details including credits for director, cast, writers

                const detailsResponse = await fetch(`https://api.themoviedb.org/3/movie/${movieId}?api\_key=${TMDB\_API\_KEY}`);

                const movieDetails = await detailsResponse.json();

                if (movieDetails && movieDetails.id) {

                    const creditsResponse = await fetch(`https://api.themoviedb.org/3/movie/${movieId}/credits?api\_key=${TMDB\_API\_KEY}`);

                    const creditsData = await creditsResponse.json();

                    const director = creditsData.crew.find(member => member.job === 'Director');

                    // Get top 5 cast members with their profile path and person ID

                    const detailedCast = creditsData.cast.slice(0, 10).map(member => ({ // Increased to 10 for more options

                        name: member.name,

                        profile\_path: member.profile\_path,

                        id: member.id // This is the person\_id

                    }));

                    const writers = creditsData.crew.filter(member =>

                        member.department === 'Writing' || member.job === 'Screenplay' || member.job === 'Writer'

                    ).map(member => member.name);

                    const videosResponse = await fetch(`https://api.themoviedb.org/3/movie/${movieId}/videos?api\_key=${TMDB\_API\_KEY}`);

                    const videosData = await videosResponse.json();

                    const trailer = videosData.results.find(video => video.type === 'Trailer' && video.site === 'YouTube');

                    const dataToSend = {

                        title: movieDetails.title,

                        poster\_path: movieDetails.poster\_path ? `https://image.tmdb.org/t/p/w500${movieDetails.poster\_path}` : 'https://placehold.co/300x450/CCCCCC/333333?text=No+Image',

                        overview: movieDetails.overview,

                        vote\_average: movieDetails.vote\_average,

                        vote\_count: movieDetails.vote\_count,

                        genres: movieDetails.genres.map(g => g.name).join(', '),

                        release\_date: movieDetails.release\_date,

                        runtime: movieDetails.runtime,

                        status: movieDetails.status,

                        director: director ? director.name : 'N/A',

                        cast: detailedCast, // Pass the structured detailedCast array

                        budget: movieDetails.budget,

                        revenue: movieDetails.revenue,

                        original\_language: movieDetails.original\_language,

                        writers: writers.length > 0 ? Array.from(new Set(writers)).join(', ') : 'N/A',

                        trailer\_key: trailer ? trailer.key : null,

                        tmdb\_id: movieDetails.id // Pass the TMDb ID for review fetching

                    };

                    const recommendResponse = await fetch('/recommend', {

                        method: 'POST',

                        headers: { 'Content-Type': 'application/json' },

                        body: JSON.stringify(dataToSend)

                    });

                    if (recommendResponse.ok) {

                        document.open();

                        document.write(await recommendResponse.text());

                        document.close();

                        // Important: Re-initialize event listeners after document.write()

                        initializeRecommendPageListeners();

                    } else {

                        const errorText = await recommendResponse.text();

                        console.error('Error from /recommend route:', errorText);

                        $(failDiv).show().find('h3').text("An error occurred fetching recommendations from the server. Please try again.");

                    }

                } else {

                    $(failDiv).show().find('h3').text("Could not retrieve full details for this movie from the external database.");

                }

            } catch (error) {

                console.error('Error during movie search or recommendation:', error);

                $(failDiv).show().find('h3').text("Failed to connect to movie databases or an unexpected error occurred. Please check your internet connection and try again later.");

            } finally {

                $(loader).hide();

            }

        }

    }

    // --- Function to initialize event listeners specific to recommend.html ---

    // This function needs to be called after document.write()

    function initializeRecommendPageListeners() {

        console.log("Initializing recommend page listeners...");

        // --- Movie Reviews Button Handler (was IMDb Reviews) ---

        // Use event delegation for the button, as it might be recreated.

        $(document).on('click', '#imdb-reviews-btn', async function() { // Keep ID for now, just change functionality

            const tmdbId = $(this).data('tmdb-id'); // Now getting TMDb ID

            const movieTitle = $('h1').text(); // Get movie title from the h1 tag on the page

            const reviewsModal = $('#reviewsModal');

            const reviewsContent = $('#reviews-content');

            const modalMovieTitle = $('#modal-movie-title');

            const reviewLoader = $('#review-loader');

            const reviewLoaderText = $('#review-loader-text');

            modalMovieTitle.text(movieTitle); // Set modal title based on the currently displayed movie

            if (!tmdbId || tmdbId === 'None' || tmdbId === 'null') { // Check for TMDb ID now

                reviewsContent.html('<p class="text-center" style="color: #f56565;">No TMDb ID found for this movie. Cannot fetch reviews.</p>');

                reviewsModal.modal('show');

                return;

            }

            reviewsContent.empty(); // Clear previous reviews

            reviewLoader.show();

            reviewLoaderText.show().text('Loading reviews from TMDb...');

            reviewsModal.modal('show'); // Show the modal immediately

            try {

                const response = await fetch('/get\_movie\_reviews', { // Updated route name

                    method: 'POST',

                    headers: { 'Content-Type': 'application/json' },

                    body: JSON.stringify({ tmdb\_id: tmdbId }) // Sending TMDb ID

                });

                const data = await response.json();

                reviewLoader.hide();

                reviewLoaderText.hide();

                reviewsContent.empty(); // Clear loader/text after data is received

                if (data.reviews && data.reviews.length > 0) {

                    data.reviews.forEach(review => {

                        let sentimentClass = '';

                        if (review.sentiment === 'Positive') {

                            sentimentClass = 'sentiment-positive';

                        } else if (review.sentiment === 'Negative') {

                            sentimentClass = 'sentiment-negative';

                        } else {

                            sentimentClass = 'sentiment-unknown';

                        }

                        const reviewHtml = `

                            <div class="review-card">

                                <p class="review-text">"${review.text}"</p>

                                <p><span class="info-label">Sentiment:</span> <span class="${sentimentClass}">${review.sentiment}</span></p>

                            </div>

                        `;

                        reviewsContent.append(reviewHtml);

                    });

                } else {

                    reviewsContent.html('<p class="text-center" style="color: #ccc;">No reviews found on TMDb for this movie.</p>');

                }

            } catch (error) {

                console.error('Error fetching TMDb reviews:', error);

                reviewsContent.html('<p class="text-center" style="color: #f56565;">Failed to load reviews from TMDb. Please try again later.</p>');

            }

        });

        // --- Event Listener for Clicking on Recommended Movies (on recommend.html) ---

        // Also using event delegation for these buttons

        $(document).on('click', '.select-recommended-movie-btn', async function() {

            const card = $(this).closest('.card');

            const movieId = card.data('movie-id'); // This is TMDb ID

            const movieTitle = card.data('movie-title');

            // You likely have a global loader on the home page, but not directly on recommend.html.

            // If you want a loader here, you'd need to add one to recommend.html as well,

            // or dynamically inject a temporary one. For simplicity, we'll just navigate.

            // Re-use the main loader if present, or add a temporary one for navigation

            const mainLoader = document.getElementById('loader'); // Check if it exists from home.html

            if (mainLoader) {

                 $(mainLoader).show();

            } else {

                // If no main loader, provide a basic feedback

                $('body').append('<div id="temp-loader" style="position: fixed; top: 50%; left: 50%; transform: translate(-50%, -50%); background: rgba(0,0,0,0.8); color: white; padding: 20px; border-radius: 10px; z-index: 9999;">Loading details for "'+ movieTitle +'"...</div>');

            }

            try {

                // Fetch full movie details including credits for director, cast, writers

                const detailsResponse = await fetch(`https://api.themoviedb.org/3/movie/${movieId}?api\_key=${TMDB\_API\_KEY}`);

                const movieDetails = await detailsResponse.json();

                if (movieDetails && movieDetails.id) {

                    const creditsResponse = await fetch(`https://api.themoviedb.org/3/movie/${movieId}/credits?api\_key=${TMDB\_API\_KEY}`);

                    const creditsData = await creditsResponse.json();

                    const director = creditsData.crew.find(member => member.job === 'Director');

                    const detailedCast = creditsData.cast.slice(0, 10).map(member => ({ // Increased to 10

                        name: member.name,

                        profile\_path: member.profile\_path,

                        id: member.id

                    }));

                    const writers = creditsData.crew.filter(member =>

                        member.department === 'Writing' || member.job === 'Screenplay' || member.job === 'Writer'

                    ).map(member => member.name);

                    const videosResponse = await fetch(`https://api.themoviedb.org/3/movie/${movieId}/videos?api\_key=${TMDB\_API\_KEY}`);

                    const videosData = await videosResponse.json();

                    const trailer = videosData.results.find(video => video.type === 'Trailer' && video.site === 'YouTube');

                    const dataToSend = {

                        title: movieDetails.title,

                        poster\_path: movieDetails.poster\_path ? `https://image.tmdb.org/t/p/w500${movieDetails.poster\_path}` : 'https://placehold.co/300x450/CCCCCC/333333?text=No+Image',

                        overview: movieDetails.overview,

                        vote\_average: movieDetails.vote\_average,

                        vote\_count: movieDetails.vote\_count,

                        genres: movieDetails.genres.map(g => g.name).join(', '),

                        release\_date: movieDetails.release\_date,

                        runtime: movieDetails.runtime,

                        status: movieDetails.status,

                        director: director ? director.name : 'N/A',

                        cast: detailedCast, // Send structured cast data

                        budget: movieDetails.budget,

                        revenue: movieDetails.revenue,

                        original\_language: movieDetails.original\_language,

                        writers: writers.length > 0 ? Array.from(new Set(writers)).join(', ') : 'N/A',

                        trailer\_key: trailer ? trailer.key : null,

                        tmdb\_id: movieDetails.id // Pass the TMDb ID for review fetching

                    };

                    const recommendResponse = await fetch('/recommend', {

                        method: 'POST',

                        headers: { 'Content-Type': 'application/json' },

                        body: JSON.stringify(dataToSend)

                    });

                    if (recommendResponse.ok) {

                        document.open();

                        document.write(await recommendResponse.text());

                        document.close();

                        initializeRecommendPageListeners(); // Re-initialize for the new page content

                    } else {

                        const errorText = await recommendResponse.text();

                        console.error('Error from /recommend route (recommended movie click):', errorText);

                        alert("An error occurred loading recommended movie details. Please try again.");

                    }

                } else {

                    alert("Could not retrieve details for the selected movie.");

                }

            } catch (error) {

                console.error('Error clicking recommended movie:', error);

                alert("Failed to load recommended movie details. Please check your internet connection.");

            } finally {

                if (mainLoader) {

                    $(mainLoader).hide();

                } else {

                    $('#temp-loader').remove(); // Remove temporary loader

                }

            }

        });

        // --- Cast Biography Modal Handler ---

        $(document).on('click', '.cast-member-card', async function() {

            const personId = $(this).data('person-id');

            const personName = $(this).data('person-name');

            const bioModal = $('#castBioModal');

            const bioContent = $('#bio-content');

            const bioName = $('#cast-bio-name');

            const bioLoader = $('#bio-loader');

            const bioLoaderText = $('#bio-loader-text');

            bioName.text(personName);

            bioContent.empty(); // Clear previous content

            bioLoader.show();

            bioLoaderText.show().text('Loading biography...');

            bioModal.modal('show');

            try {

                const personDetailsUrl = `https://api.themoviedb.org/3/person/${personId}?api\_key=${TMDB\_API\_KEY}`;

                const response = await fetch(personDetailsUrl);

                const personData = await response.json();

                bioLoader.hide();

                bioLoaderText.hide();

                bioContent.empty(); // Clear loader/text after data is received

                if (personData && personData.id) {

                    let biography = personData.biography || 'Biography not available.';

                    // Truncate long biographies, offer "read more" if needed

                    if (biography.length > 800) { // Arbitrary length, adjust as desired

                        const truncatedBio = biography.substring(0, 800);

                        biography = `<span class="truncated-bio">${truncatedBio} ...</span> <a href="#" class="read-more-link" data-full-bio="${encodeURIComponent(biography)}">Read More</a>`;

                    }

                    let birthday = personData.birthday ? new Date(personData.birthday).toLocaleDateString('en-US', { year: 'numeric', month: 'long', day: 'numeric' }) : 'N/A';

                    let deathday = personData.deathday ? new Date(personData.deathday).toLocaleDateString('en-US', { year: 'numeric', month: 'long', day: 'numeric' }) : 'N/A';

                    let age = 'N/A';

                    if (personData.birthday) {

                        const birthDate = new Date(personData.birthday);

                        const today = new Date();

                        let calculatedAge = today.getFullYear() - birthDate.getFullYear();

                        const m = today.getMonth() - birthDate.getMonth();

                        if (m < 0 || (m === 0 && today.getDate() < birthDate.getDate())) {

                            calculatedAge--;

                        }

                        if (personData.deathday) {

                            const deathDate = new Date(personData.deathday);

                            calculatedAge = deathDate.getFullYear() - birthDate.getFullYear();

                            const dm = deathDate.getMonth() - birthDate.getMonth();

                            if (dm < 0 || (dm === 0 && deathDate.getDate() < birthDate.getDate())) {

                                calculatedAge--;

                            }

                            age = `${calculatedAge} (Deceased on ${deathday})`;

                        } else {

                            age = calculatedAge;

                        }

                    }

                    const knownForDepartment = personData.known\_for\_department || 'N/A';

                    const placeOfBirth = personData.place\_of\_birth || 'N/A';

                    const bioHtml = `

                        <p><span class="info-label">Known For:</span> <span class="info-value">${knownForDepartment}</span></p>

                        <p><span class="info-label">Born:</span> <span class="info-value">${birthday}</span> (Age: ${age})</p>

                        ${personData.deathday ? `<p><span class="info-label">Died:</span> <span class="info-value">${deathday}</span></p>` : ''}

                        <p><span class="info-label">Place of Birth:</span> <span class="info-value">${placeOfBirth}</span></p>

                        <hr style="border-top: 1px solid #4a5568;">

                        <h5 style="color: #e50914;">Biography</h5>

                        <p class="biography-text" style="color: #eee;">${biography}</p>

                    `;

                    bioContent.append(bioHtml);

                    // Add click listener for "Read More" if present

                    bioContent.on('click', '.read-more-link', function(e) {

                        e.preventDefault();

                        const fullBio = decodeURIComponent($(this).data('full-bio'));

                        $(this).siblings('.truncated-bio').text(fullBio);

                        $(this).remove(); // Remove the "Read More" link

                    });

                } else {

                    bioContent.html('<p class="text-center" style="color: #f56565;">Could not retrieve biography for this person.</p>');

                }

            } catch (error) {

                console.error('Error fetching person details:', error);

                bioContent.html('<p class="text-center" style="color: #f56565;">Failed to load biography. Please try again later.</p>');

            }

        });

    }

    // Call initializeRecommendPageListeners once when the initial document is loaded

    // (in case the user directly navigates to recommend.html, or it's the first render).

    // This handles the initial setup if already on recommend.html.

    // We check for elements unique to recommend.html.

    if ($('#imdb-reviews-btn').length || $('.select-recommended-movie-btn').length || $('.cast-member-card').length) {

        initializeRecommendPageListeners();

    }

});

## **Home.html**

</html><!DOCTYPE html>

<html>

<head>

    <title>Movie Recommendation System</title>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <link href="https://fonts.googleapis.com/css?family=IBM+Plex+Sans&display=swap" rel="stylesheet">

    <link href="https://fonts.googleapis.com/css2?family=Noto+Sans+JP&display=swap" rel="stylesheet">

    <link href="https://fonts.googleapis.com/css2?family=Inter:wght@400;700&display=swap" rel="stylesheet">

    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.4/css/all.min.css">

    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">

    <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='style.css') }}">

    <style>

        #autocomplete-list {

            background-color: #333; /\* Dark background for suggestions \*/

            border: 1px solid #555;

            max-height: 200px;

            overflow-y: auto;

            list-style: none; /\* Remove bullet points \*/

            padding: 0;

            margin: 0;

            position: absolute; /\* Position relative to its parent container \*/

            width: 60%; /\* Match the input field width \*/

            left: 50%; /\* Center horizontally \*/

            transform: translateX(-50%); /\* Adjust for centering \*/

            z-index: 100; /\* Ensure it appears above other content \*/

            border-radius: 0.5rem; /\* Rounded corners \*/

            box-shadow: 0 4px 6px rgba(0,0,0,0.1); /\* Subtle shadow \*/

            display: none; /\* Hidden by default, JavaScript will show it \*/

        }

        #autocomplete-list div {

            padding: 0.5rem 1rem;

            color: #eee; /\* Light text color \*/

            cursor: pointer;

        }

        #autocomplete-list div:hover,

        #autocomplete-list div.autocomplete-active { /\* Style for hover and keyboard-selected item \*/

            background-color: #555; /\* Slightly lighter background on hover/active \*/

        }

    </style>

</head>

<body id="content" style="font-family: 'Noto Sans JP', sans-serif;">

    <div class="ml-container" style="display: block;">

        <a href="https://github.com/samminga-akshay/Movie-Recommendation-System-2.git" target="\_blank" class="github-corner" title="View source on GitHub">

            <svg data-toggle="tooltip" data-placement="left" width="80" height="80" viewBox="0 0 250 250"

                 style="fill:#31312d00; color:#fff; position: fixed;z-index:100; top: 0; border: 0; right: 0;" aria-hidden="true">

                <path d="M0,0 L115,115 L130,115 L142,142 L250,250 L250,0 Z"></path>

                <path

                    d="M128.3,109.0 C113.8,99.7 119.0,89.6 119.0,89.6 C122.0,82.7 120.5,78.6 120.5,78.6 C119.2,72.0 123.4,76.3 123.4,76.3 C127.3,80.9 125.5,87.3 125.5,87.3 C122.9,97.6 130.6,101.9 134.4,103.2"

                    fill="currentColor" style="transform-origin: 130px 106px;" class="octo-arm"></path>

                <path

                    d="M115.0,115.0 C114.9,115.1 118.7,116.5 119.8,115.4 L133.7,101.6 C136.9,99.2 139.9,98.4 142.2,98.6 C133.8,88.0 127.5,74.4 143.8,58.0 C148.5,53.4 154.0,51.2 159.7,51.0 C160.3,49.4 163.2,43.6 171.4,40.1 C171.4,40.1 176.1,42.5 178.8,56.2 C183.1,58.6 187.2,61.8 190.9,65.4 C194.5,69.0 197.7,73.2 200.1,77.6 C213.8,80.2 216.3,84.9 216.3,84.9 C212.7,93.1 206.9,96.0 205.4,96.6 C205.1,102.4 203.0,107.8 198.3,112.5 C181.9,128.9 168.3,122.5 157.7,114.1 C157.9,116.9 156.7,120.9 152.7,124.9 L141.0,136.5 C139.8,137.7 141.6,141.9 141.8,141.8 Z"

                    fill="currentColor" style="transform-origin: 130px 106px;" class="octo-arm"></path>

            </svg>

        </a>

        <center><h1> Movie Recommendation System</h1></centter>

        <div class="form-group shadow-textarea" style="margin-top: 30px;text-align: center;color: white; position: relative;">

            <input type="text" name="movie" class="movie form-control" id="movie-search-input" autocomplete="off" placeholder="Enter the Movie Name" style="background-color: #ffffff;border-color:#ffffff;width: 60%;color: #181818" required="required" />

            <div id="autocomplete-list"></div>

            <br>

        </div>

        <div class="form-group" style="text-align: center;">

            <!-- <button class="btn btn-primary btn-block movie-button" id="search-button" style="background-color: #ffff00;text-align: center;border-color: #000000;width:120px;" disabled>Enter</button><br><br> -->

            <button class="btn btn-primary btn-block movie-button" id="search-button" style="background-color: #ffff00; text-align: center; border-color: #000000; width:120px; color: #000000;" disabled>Enter</button><br><br>

        </div>

    </div>

    <div id="loader" class="loader text-center"></div>

    <div id="loader-text" class="text-white mt-2" style="display: none;">Finding recommendations...</div>

    <div class="fail">

        <center><h3>Sorry! The movie you requested is not in our database.

        Please check the spelling or try with other movies!</h3></center>

    </div>

    <div class="results">

        <center>

            <h2 id="name" class="text-uppercase"></h2>

        </center>

    </div>

    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>

    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl" crossorigin="anonymous"></script>

    <script type="text/javascript" src="{{ url\_for('static', filename='recommend.js') }}"></script>

</body>

## **Recommend.html**

<!DOCTYPE html>

<html>

<head>

    <title>{{ title }} - Movie Details</title>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <link href="https://fonts.googleapis.com/css?family=IBM+Plex+Sans&display=swap" rel="stylesheet">

    <link href="https://fonts.googleapis.com/css2?family=Noto+Sans+JP&display=swap" rel="stylesheet">

    <link href="https://fonts.googleapis.com/css2?family=Inter:wght@400;700&display=swap" rel="stylesheet">

    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">

    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">

    <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='style.css') }}">

    <style>

        /\* Specific styles for recommend.html if needed, otherwise rely on style.css \*/

        #movie-poster {

            width: 300px;

            height: 450px;

            object-fit: cover;

            border-radius: 10px;

            box-shadow: 0px 4px 15px rgba(0, 0, 0, 0.5);

        }

        .movie-info-section {

            padding: 20px;

            background-color: rgba(0, 0, 0, 0.7); /\* Slightly transparent dark background \*/

            border-radius: 10px;

            margin-bottom: 20px;

        }

        .section-title {

            color: #e50914; /\* Netflix Red \*/

            margin-top: 20px;

            margin-bottom: 15px;

            font-size: 1.8rem;

            font-weight: bold;

        }

        .info-label {

            font-weight: bold;

            color: #ccc;

        }

        .info-value {

            color: #eee;

        }

        .review-card {

            background-color: #2d3748; /\* Darker background for review cards \*/

            border-radius: 0.5rem;

            padding: 1rem;

            margin-bottom: 1rem;

            box-shadow: 0 2px 4px rgba(0, 0, 0, 0.2);

        }

        .sentiment-positive {

            color: #48bb78; /\* Green for positive \*/

            font-weight: bold;

        }

        .sentiment-negative {

            color: #f56565; /\* Red for negative \*/

            font-weight: bold;

        }

        .sentiment-unknown {

            color: #ecc94b; /\* Yellow for unknown \*/

            font-weight: bold;

        }

        .review-text {

            color: #edf2f7; /\* Light gray for review text \*/

        }

        /\* Style for Bootstrap modal backdrop \*/

        .modal-backdrop.fade.show {

            opacity: 0.5; /\* Adjust as needed for modal background opacity \*/

        }

        /\* Custom styles for the YouTube embed to make it responsive \*/

        .video-container {

            position: relative;

            padding-bottom: 56.25%; /\* 16:9 aspect ratio \*/

            height: 0;

            overflow: hidden;

            max-width: 100%;

            background: #000;

            border-radius: 0.5rem;

        }

        .video-container iframe {

            position: absolute;

            top: 0;

            left: 0;

            width: 100%;

            height: 100%;

            border: 0;

        }

        /\* Styles for recommended movies cards \*/

        .card {

            background-color: #2d3748; /\* Dark background for cards \*/

            border: none;

            border-radius: 0.5rem;

            overflow: hidden;

            transition: transform 0.2s ease-in-out, box-shadow 0.2s ease-in-out;

        }

        .card:hover {

            transform: translateY(-5px);

            box-shadow: 0 8px 15px rgba(0, 0, 0, 0.3);

        }

        .card-img-top {

            width: 100%;

            height: 250px; /\* Fixed height for consistency \*/

            object-fit: cover;

            border-top-left-radius: 0.5rem;

            border-top-right-radius: 0.5rem;

        }

        .card-body {

            padding: 1rem;

        }

        .card-title {

            font-size: 1.1rem;

            font-weight: bold;

            margin-bottom: 0.75rem;

            color: #e50914; /\* Netflix red \*/

        }

        .select-recommended-movie-btn {

            background-color: #e50914;

            border-color: #e50914;

            width: 100%;

            border-radius: 0.3rem;

            transition: background-color 0.2s ease-in-out, border-color 0.2s ease-in-out;

        }

        .select-recommended-movie-btn:hover {

            background-color: #ff0000; /\* Slightly darker red on hover \*/

            border-color: #ff0000;

        }

        /\* New Styles for Cast Section \*/

        .cast-container {

            display: flex;

            flex-wrap: wrap;

            gap: 15px; /\* Space between cast cards \*/

            justify-content: center; /\* Center cast members \*/

        }

        .cast-member-card {

            background-color: #1a202c; /\* Even darker background for cast member cards \*/

            border-radius: 8px;

            overflow: hidden;

            width: 120px; /\* Fixed width for cast cards \*/

            flex-shrink: 0;

            text-align: center;

            cursor: pointer;

            transition: transform 0.2s ease-in-out, box-shadow 0.2s ease-in-out;

            box-shadow: 0 2px 5px rgba(0, 0, 0, 0.3);

        }

        .cast-member-card:hover {

            transform: translateY(-3px);

            box-shadow: 0 5px 10px rgba(0, 0, 0, 0.5);

        }

        .cast-member-card img {

            width: 100%;

            height: 150px; /\* Fixed height for cast images \*/

            object-fit: cover;

            border-top-left-radius: 8px;

            border-top-right-radius: 8px;

        }

        .cast-member-card p {

            color: #eee;

            font-size: 0.9rem;

            margin: 5px 0 0;

            padding: 0 5px;

            white-space: nowrap;

            overflow: hidden;

            text-overflow: ellipsis;

        }

        .cast-member-card .cast-age {

            font-size: 0.8rem;

            color: #bbb;

            margin-bottom: 5px;

        }

        /\* Custom scrollbar for modals \*/

        .custom-scrollbar {

            scrollbar-width: thin;

            scrollbar-color: #e50914 #2d3748;

        }

        .custom-scrollbar::-webkit-scrollbar {

            width: 8px;

        }

        .custom-scrollbar::-webkit-scrollbar-track {

            background: #2d3748;

            border-radius: 10px;

        }

        .custom-scrollbar::-webkit-scrollbar-thumb {

            background-color: #e50914;

            border-radius: 10px;

            border: 2px solid #2d3748;

        }

    </style>

</head>

<body id="content">

    <div class="ml-container">

        <a href="{{ url\_for('home') }}" class="btn btn-primary" style="background-color: #e50914; border-color: #e50914; position: fixed; top: 20px; left: 20px; z-index: 100;">

            <i class="fa fa-home"></i> Go to Home

        </a>

        <center><h1 style="color: white; font-weight: bold;">{{ title }}</h1></center>

        <div class="movie-content" style="display: flex; flex-wrap: wrap; justify-content: space-around;">

            <div class="col-md-4 text-center">

                <img id="movie-poster" src="{{ poster\_path if poster\_path else 'https://placehold.co/300x450/CCCCCC/333333?text=No+Image' }}" class="poster" alt="{{ title }} Poster">

                <br><br>

                <div class="text-center">

                    <button class="btn btn-warning" id="imdb-reviews-btn" data-tmdb-id="{{ tmdb\_id }}">

                        <i class="fa fa-comments"></i> TMDb Reviews

                    </button>

                </div>

            </div>

            <div class="col-md-8 movie-info-section">

                <h3 style="color: white;">Overview</h3>

                <p style="color: #eee;">{{ overview }}</p>

                <hr style="border-top: 1px solid #4a5568;">

                <div class="row">

                    <div class="col-md-6">

                        <p><span class="info-label">Rating:</span> <span class="info-value">{{ '%.1f' | format(vote\_average) }} / 10 (<i class="fa fa-star" style="color: gold;"></i>)</span></p>

                        <p><span class="info-label">Votes:</span> <span class="info-value">{{ '{:,.0f}'.format(vote\_count) }}</span></p>

                        <p><span class="info-label">Genres:</span> <span class="info-value">{{ genres }}</span></p>

                        <p><span class="info-label">Release Date:</span> <span class="info-value">{{ release\_date }}</span></p>

                        <p><span class="info-label">Runtime:</span> <span class="info-value">{{ runtime }} minutes</span></p>

                        <p><span class="info-label">Status:</span> <span class="info-value">{{ status }}</span></p>

                    </div>

                    <div class="col-md-6">

                        <p><span class="info-label">Budget:</span> <span class="info-value">${{ '{:,.0f}'.format(budget) if budget and budget > 0 else 'N/A' }}</span></p>

                        <p><span class="info-label">Revenue:</span> <span class="info-value">${{ '{:,.0f}'.format(revenue) if revenue and revenue > 0 else 'N/A' }}</span></p>

                        <p><span class="info-label">Original Language:</span> <span class="info-value">{{ original\_language | upper }}</span></p>

                        <p><span class="info-label">Director:</span> <span class="info-value">{{ director if director else 'N/A' }}</span></p>

                        <p><span class="info-label">Writers:</span> <span class="info-value">{{ writers if writers else 'N/A' }}</p>

                    </div>

                </div>

                <hr style="border-top: 1px solid #4a5568;">

                <h3 style="color: white;">Cast</h3>

                <div class="cast-container">

                    {% if cast %}

                        {% for member in cast %}

                        <div class="cast-member-card" data-person-id="{{ member.id }}" data-person-name="{{ member.name }}">

                            <img src="{{ 'https://image.tmdb.org/t/p/w185' + member.profile\_path if member.profile\_path else 'https://placehold.co/120x150/2d3748/edf2f7?text=No+Photo' }}" alt="{{ member.name }}">

                            <p>{{ member.name }}</p>

                        </div>

                        {% endfor %}

                    {% else %}

                        <p style="color: #eee;">Cast information not available.</p>

                    {% endif %}

                </div>

                {% if trailer\_key %}

                <hr style="border-top: 1px solid #4a5568;">

                <h3 style="color: white;">Trailer</h3>

                <div class="video-container">

                    <iframe src="https://www.youtube.com/embed/{{ trailer\_key }}" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

                </div>

                {% endif %}

            </div>

        </div>

        <hr style="border-top: 2px solid #e50914;">

        <h2 class="section-title">Recommended Movies For You</h2>

        <div class="row">

            {% for movie in recommended\_movies %}

            <div class="col-6 col-sm-4 col-md-3 col-lg-2 mb-4 d-flex align-items-stretch">

                <div class="card bg-dark text-white shadow-sm flex-fill" data-movie-id="{{ movie.id }}" data-movie-title="{{ movie.title }}">

                    <img class="card-img-top" src="{{ movie.poster\_url }}" alt="{{ movie.title }} Poster">

                    <div class="card-body d-flex flex-column">

                        <h5 class="card-title text-center">{{ movie.title }}</h5>

                        <button class="btn btn-primary mt-auto select-recommended-movie-btn" style="background-color: #e50914; border-color: #e50914;">View Details</button>

                    </div>

                </div>

            </div>

            {% endfor %}

        </div>

    </div>

    <div class="modal fade" id="reviewsModal" tabindex="-1" role="dialog" aria-labelledby="reviewsModalLabel" aria-hidden="true">

        <div class="modal-dialog modal-dialog-scrollable modal-lg" role="document">

            <div class="modal-content" style="background-color: #1a202c; color: white; border-radius: 10px;">

                <div class="modal-header" style="border-bottom: 1px solid #4a5568;">

                    <h5 class="modal-title" id="reviewsModalLabel" style="color: #e50914;">TMDb Reviews for <span id="modal-movie-title"></span></h5>

                    <button type="button" class="close" data-dismiss="modal" aria-label="Close" style="color: white;">

                        <span aria-hidden="true">×</span>

                    </button>

                </div>

                <div class="modal-body custom-scrollbar">

                    <div id="reviews-content">

                        <div class="text-center">

                            <div id="review-loader" class="loader"></div>

                            <p id="review-loader-text" style="color: white; display: none;">Loading reviews...</p>

                        </div>

                    </div>

                </div>

                <div class="modal-footer" style="border-top: 1px solid #4a5568;">

                    <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>

                </div>

            </div>

        </div>

    </div>

    <div class="modal fade" id="castBioModal" tabindex="-1" role="dialog" aria-labelledby="castBioModalLabel" aria-hidden="true">

        <div class="modal-dialog modal-dialog-scrollable modal-lg" role="document">

            <div class="modal-content" style="background-color: #1a202c; color: white; border-radius: 10px;">

                <div class="modal-header" style="border-bottom: 1px solid #4a5568;">

                    <h5 class="modal-title" id="castBioModalLabel" style="color: #e50914;">Biography: <span id="cast-bio-name"></span></h5>

                    <button type="button" class="close" data-dismiss="modal" aria-label="Close" style="color: white;">

                        <span aria-hidden="true">×</span>

                    </button>

                </div>

                <div class="modal-body custom-scrollbar">

                    <div id="bio-content">

                        <div class="text-center">

                            <div id="bio-loader" class="loader"></div>

                            <p id="bio-loader-text" style="color: white; display: none;">Loading biography...</p>

                        </div>

                    </div>

                </div>

                <div class="modal-footer" style="border-top: 1px solid #4a5568;">

                    <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>

                </div>

            </div>

        </div>

    </div>

    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>

    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl" crossorigin="anonymous"></script>

    <script type="text/javascript" src="{{url\_for('static', filename='recommend.js')}}"></script>

</body>

</html>

## **Style.css**

@import url('https://fonts.googleapis.com/css2?family=Inter:wght@400;700&display=swap');

@import url('https://fonts.googleapis.com/css2?family=Noto+Sans+JP&display=swap');

@import url('https://fonts.googleapis.com/css?family=IBM+Plex+Sans&display=swap');

body {

    font-family: 'Inter', sans-serif;

    background-color: #1a202c;

    margin: 0;

    padding: 0;

}

/\* Custom Scrollbar \*/

.custom-scrollbar::-webkit-scrollbar {

    width: 8px;

}

.custom-scrollbar::-webkit-scrollbar-track {

    background: #2d3748;

    border-radius: 10px;

}

.custom-scrollbar::-webkit-scrollbar-thumb {

    background: #4a5568;

    border-radius: 10px;

}

.custom-scrollbar::-webkit-scrollbar-thumb:hover {

    background: #6b7280;

}

/\* Loader \*/

.loader {

    border: 4px solid #f3f3f3;

    border-top: 4px solid #3498db;

    border-radius: 50%;

    width: 40px;

    height: 40px;

    animation: spin 1s linear infinite;

    position: fixed;

    top: 50%;

    left: 50%;

    transform: translate(-50%, -50%);

    z-index: 1000;

    display: none;

}

#review-loader {

    width: 2rem;

    height: 2rem;

    border-top: 2px solid #60a5fa;

    position: static;

    margin: 0 auto;

    margin-top: 20px;

}

@keyframes spin {

    0% { transform: rotate(0deg); }

    100% { transform: rotate(360deg); }

}

/\* Modal \*/

#overlay {

    transition: opacity 0.3s ease-out;

    opacity: 0;

    pointer-events: none;

    background-color: rgba(0, 0, 0, 0.7);

}

#overlay.active {

    opacity: 1;

    pointer-events: auto;

}

.modal {

    transition: transform 0.3s ease-out, opacity 0.3s ease-out;

    transform: translateY(-20px);

    opacity: 0;

    pointer-events: none;

}

.modal.active {

    transform: translateY(0);

    opacity: 1;

    pointer-events: auto;

}

/\* Movie Content \*/

.movie {

    color: #fff;

    margin-left: auto;

    margin-right: auto;

    resize: none;

}

.movie-content {

    display: flex;

    flex-wrap: wrap;

    justify-content: space-around;

}

.movie-content > div {

    margin: 20px;

}

.btn-block {

    width: 15%;

    text-align: center;

    margin-left: auto;

    margin-right: auto;

    color: #e4e0e0;

}

#content {

    background-image: url("/static/image.jpg");

    background-color: #181818;

    font-family: 'Noto Sans JP', sans-serif;

    background-size: cover;

    background-position: center;

    background-attachment: fixed;

}

#details {

    margin-left: 50px;

}

.footer {

    color: #e4e0e0;

    text-align: right;

    position: fixed;

    bottom: 20px;

    right: 20px;

    width: 100%;

}

h1 {

    font-family: 'Inter', 'Helvetica Neue', Helvetica, Arial, sans-serif;

    color: #ffff00;

    font-weight: bold;

    margin-top: 30px;

    text-shadow: #000000 0px 0px 13px;

}

.github-corner:hover .octo-arm {

    animation: octocat-wave 560ms ease-in-out;

}

@keyframes octocat-wave {

    0%, 100% { transform: rotate(0); }

    20%, 60% { transform: rotate(-25deg); }

    40%, 80% { transform: rotate(10deg); }

}

#autoComplete {

    background-position: 98%;

}

#name {

    color: white;

    padding: 1px;

}

h6 {

    margin-bottom: 20px;

}

/\* Media Queries \*/

@media only screen and (max-width: 650px) {

    #mcontent {

        display: block;

    }

    .poster-lg {

        display: none;

    }

    #details {

        margin-left: 30px;

    }

    #loader-text {

        vertical-align: middle;

        color: white;

    }

    #autoComplete {

        background-position: 97%;

    }

    svg[data-toggle=tooltip] {

        width: 50px;

        height: 50px;

    }

}

@media only screen and (max-width: 991px) {

    .modal-body {

        display: block;

    }

    .profile-pic {

        margin-left: auto;

        margin-right: auto;

        display: block;

        margin-bottom: 20px;

    }

}

@media only screen and (min-width: 992px) {

    .modal-body {

        display: flex;

    }

}

@media only screen and (min-width: 651px) {

    .poster-sm {

        display: none;

    }

    #mcontent {

        display: flex;

        flex-wrap: nowrap;

    }

    #loader-text {

        vertical-align: middle;

        color: white;

    }

}

/\* Card & Cast Hover Effects \*/

.poster {

    -webkit-box-shadow: 0px 1px 15px 4px rgba(250,250,250,1);

    -moz-box-shadow: 0px 1px 15px 4px rgba(250,250,250,1);

    box-shadow: 0px 1px 15px 4px rgba(250,250,250,1);

}

.card:hover, .castcard:hover {

    cursor: pointer;

}

.cast-img {

    filter: brightness(100%);

    transition: all 0.75s ease;

}

.cast-img:hover {

    filter: brightness(50%);

    transition: all 0.75s ease;

}

.fig {

    display: flex;

    align-items: center;

    justify-content: center;

    backdrop-filter: brightness(50%);

    position: absolute;

    bottom: 0;

    top: 0;

    right: 0;

    left: 0;

    opacity: 0;

    transition: all 0.75s ease;

}

.fig:hover {

    opacity: 1;

    backdrop-filter: brightness(30%);

    transition: all 0.75s ease;

}

.card-btn {

    border-radius: 20px;

}

.imghvr {

    position: relative;

}

.table td {

    border-color: white;

    border-style: solid;

    border-width: 1px;

}

.fail {

    display: none;

    color: white;

}

## **Sentiment.ipynb**

# --- 1. Import necessary libraries ---

import pandas as pd

import numpy as np

import nltk

from nltk.corpus import stopwords

from nltk.stem.porter import PorterStemmer # For stemming words

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import MultinomialNB # For the sentiment classification model

from sklearn.metrics import accuracy\_score, classification\_report # For evaluating the model

import pickle

import re # For regular expressions in text preprocessing

# --- 2. Download NLTK data (if not already downloaded) ---

# These are essential for text preprocessing (stopwords) and the VADER lexicon check (if needed by Flask)

print("Checking and downloading NLTK data...")

try:

    nltk.data.find('corpora/stopwords')

except LookupError:

    nltk.download('stopwords')

    print("NLTK 'stopwords' downloaded.")

try:

    nltk.data.find('sentiment/vader\_lexicon')

except LookupError:

    nltk.download('vader\_lexicon')

    print("NLTK 'vader\_lexicon' downloaded.")

print("NLTK data check complete.")

# --- 3. Load the dataset ---

# This assumes 'reviews.csv' is in the same directory as the notebook.

# It should have 'review' and 'sentiment' columns.

try:

    dataset = pd.read\_csv('reviews.csv')

    print(f"Dataset loaded successfully. Shape: {dataset.shape}")

    print("First 5 rows of the dataset:")

    print(dataset.head())

except FileNotFoundError:

    print("Error: 'reviews.csv' not found. Please ensure the file is in the same directory.")

    # Exit or handle gracefully if the dataset isn't found

    exit()

# --- 4. Prepare text preprocessing tools ---

# Set of English stopwords

stopset = set(stopwords.words('english'))

# Initialize Porter Stemmer for reducing words to their root form

ps = PorterStemmer()

# --- 5. Define text preprocessing function ---

# This function will clean and transform each review comment.

def preprocess\_text(text):

    # Remove non-alphabetic characters and replace with space

    review = re.sub('[^a-zA-Z]', ' ', text)

    # Convert to lowercase

    review = review.lower()

    # Split into individual words

    review = review.split()

    # Apply stemming and remove stopwords

    # Only stem if the word is not a stopword

    review = [ps.stem(word) for word in review if not word in stopset]

    # Join words back into a single string

    review = ' '.join(review)

    return review

# --- 6. Apply preprocessing to the 'review' column ---

print("\nPreprocessing movie reviews... This might take a moment.")

# Apply the preprocessing function to each review in the 'review' column

# Using .apply() with a lambda for cleaner syntax

corpus = dataset['review'].apply(preprocess\_text)

print("Preprocessing complete.")

# --- 7. Convert text data to TF-IDF features ---

# TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer

# It assigns a weight to each word, indicating its importance in a document relative to the corpus.

# use\_idf: Enable inverse-document frequency reweighting.

# lowercase: Convert all characters to lowercase before tokenizing.

# strip\_accents: Remove accents during the preprocessing step.

# stop\_words: Remove common English stop words.

print("Creating TF-IDF features...")

vectorizer = TfidfVectorizer(use\_idf=True, lowercase=True, strip\_accents='ascii', stop\_words=list(stopset)) # <-- Changed here

X = vectorizer.fit\_transform(corpus) # Fit and transform the preprocessed text data

print(f"TF-IDF features created. Shape: {X.shape}")

# --- 8. Save the trained TF-IDF Vectorizer ---

# The vectorizer needs to be saved so it can be used later in the Flask app

# to transform new movie reviews in the same way the training data was transformed.

filename\_vectorizer = 'tranform.pkl'

with open(filename\_vectorizer, 'wb') as file:

    pickle.dump(vectorizer, file)

print(f"TF-IDF Vectorizer saved as '{filename\_vectorizer}'")

# --- 9. Prepare labels (target variable) ---

# Map 'positive' to 1 and 'negative' to 0 for numerical classification

y = dataset['sentiment'].map({'positive': 1, 'negative': 0})

print(f"Sentiment labels mapped. First 5 labels: {y.head().tolist()}")

# --- 10. Split data into training and testing sets ---

# Split the data to evaluate the model's performance on unseen data.

# test\_size=0.20: 20% of data for testing, 80% for training.

# random\_state=42: Ensures reproducibility of the split.

print("Splitting data into training and testing sets...")

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=42)

print(f"Training set shape (X\_train, y\_train): {X\_train.shape}, {y\_train.shape}")

print(f"Test set shape (X\_test, y\_test): {X\_test.shape}, {y\_test.shape}")

# --- 11. Train the Naive Bayes Classifier ---

# Multinomial Naive Bayes is a common and effective model for text classification.

print("\nTraining Multinomial Naive Bayes classifier...")

clf = MultinomialNB()

clf.fit(X\_train, y\_train) # Train the classifier on the training data

print("Classifier training complete.")

# --- 12. Evaluate the model on the test set ---

print("\nEvaluating model on test set...")

y\_pred\_test = clf.predict(X\_test)

accuracy\_test = accuracy\_score(y\_test, y\_pred\_test) \* 100

print(f"Accuracy on test set: {accuracy\_test:.2f}%")

print("\nClassification Report (Test Set):")

print(classification\_report(y\_test, y\_pred\_test))

# --- 13. Re-train the classifier on the full dataset ---

# It's common practice to train the final model on the entire dataset

# after evaluating it, to maximize the data used for learning.

print("\nRe-training classifier on the full dataset...")

clf\_full\_data = MultinomialNB()

clf\_full\_data.fit(X, y) # Train on all TF-IDF features and labels

print("Full data classifier training complete.")

# --- 14. Save the trained Naive Bayes Classifier ---

# This is the model that the Flask app will load to make sentiment predictions.

filename\_model = 'nlp\_model.pkl'

with open(filename\_model, 'wb') as file:

    pickle.dump(clf\_full\_data, file)

print(f"NLP model saved as '{filename\_model}'")

print("\nSentiment analysis model training and saving process finished successfully!")

print("You can now find 'nlp\_model.pkl' and 'tranform.pkl' in your project directory.")

## **Requirements.txt**

attrs==25.3.0

beautifulsoup4==4.13.4

blinker==1.9.0

certifi==2025.4.26

charset-normalizer==3.4.2

click==8.2.1

colorama==0.4.6

Flask==3.1.1

gunicorn==23.0.0

idna==3.10

itsdangerous==2.2.0

Jinja2==3.1.6

joblib==1.5.1

jsonschema==4.24.0

jsonschema-specifications==2025.4.1

lxml==5.4.0

MarkupSafe==3.0.2

nltk==3.9.1

numpy==2.2.6

packaging==25.0

pandas==2.3.0

pickleshare==0.7.5

python-dateutil==2.9.0.post0

pytz==2025.2

referencing==0.36.2

regex==2024.11.6

requests==2.32.3

rpds-py==0.25.1

scikit-learn==1.7.0

scipy==1.15.3

six==1.17.0

soupsieve==2.7

threadpoolctl==3.6.0

tmdbv3api==1.9.0

tqdm==4.67.1

typing\_extensions==4.14.0

tzdata==2025.2

urllib3==2.4.0

Werkzeug==3.1.3

## **README.md**

# Movie Recommendation System with Sentiment Analysis

## 1. Project Overview

This project implements a web-based movie recommendation system that leverages content-based filtering and integrates sentiment analysis of user reviews. The system allows users to search for movies, get detailed information, receive personalized movie recommendations based on content similarity, and view the sentiment (positive/negative) of aggregated user reviews fetched from TMDb.

## 2. Key Features

\* \*\*Movie Search:\*\* Search for movies by title with an autocomplete feature.

\* \*\*Detailed Movie Information:\*\* Display comprehensive details about a selected movie including plot overview, genre, release date, cast, director, budget, revenue, and a trailer (if available).

\* \*\*Content-Based Recommendations:\*\* Generate recommendations for similar movies based on their combined features (genres, keywords, cast, crew).

\* \*\*TMDb Review Integration:\*\* Fetch real-time user reviews directly from The Movie Database (TMDb).

\* \*\*Sentiment Analysis:\*\* Predicts the sentiment (positive or negative) of fetched user reviews using a pre-trained machine learning model.

## 3. Architecture and Technology Stack

The system is built using a Python Flask backend for serving the web application and handling data processing, recommendations, and sentiment analysis. The frontend is powered by HTML, CSS, and JavaScript (jQuery) for an interactive user interface.

\* \*\*Backend:\*\* Python 3, Flask

\* \*\*Machine Learning:\*\* `scikit-learn` for TF-IDF vectorization and Cosine Similarity, `nltk` for text processing, and a pre-trained `Naive Bayes` or `Logistic Regression` model for sentiment analysis.

\* \*\*Data Source:\*\* The Movie Database (TMDb) API for movie details and user reviews.

\* \*\*Frontend:\*\* HTML5, CSS3 (Bootstrap), JavaScript (jQuery), AJAX for asynchronous requests.

## 4. Prerequisites

Before running the application, ensure you have the following installed:

\* \*\*Python 3.x\*\* (recommended Python 3.8+)

\* \*\*pip\*\* (Python package installer)

\* \*\*Git\*\* (for cloning the repository)

## 5. Setup and Installation

Follow these steps to get the project up and running on your local machine:

### 5.1. Clone the Repository

```bash

git clone [https://github.com/samminga-akshay/Movie-Recommendation-System-with-Sentiment-Analysis.git](https://github.com/samminga-akshay/Movie-Recommendation-System-with-Sentiment-Analysis.git)

cd Movie-Recommendation-System-with-Sentiment-Analysis

```

### 5.2. Create and Activate a Virtual Environment (Recommended)

It's highly recommended to use a virtual environment to manage project dependencies.

python -m venv venv

# On Windows:

.\venv\Scripts\activate

# On macOS/Linux:

source venv/bin/activate

### 5.3. Install Dependencies

Install all the required Python packages using pip:

pip install -r requirements.txt

### 5.4. Obtain and Set Up TMDb API Key

Go to The Movie Database (TMDb) website and sign up for a free account.

Once logged in, go to your account settings and request an API key (a developer key is usually sufficient).

IMPORTANT:

Open main.py and locate the line TMDB\_API\_KEY = 'YOUR\_TMDB\_API\_KEY\_HERE'. Replace 'YOUR\_TMDB\_API\_KEY\_HERE' with your actual TMDb API key.

Open static/js/recommend.js and locate the line const TMDB\_API\_KEY = 'YOUR\_TMDB\_API\_KEY\_HERE';. Replace 'YOUR\_TMDB\_API\_KEY\_HERE' with your actual TMDb API key.

(Optional but recommended for larger projects): For production environments, it's best practice to load API keys from environment variables (e.g., using a .env file and python-dotenv). For simplicity in this academic project, direct replacement is used for demonstrative purposes in the appendix.

### 5.5. Prepare Data and Models

The project requires preprocessed movie data and a trained sentiment analysis model.

Run the Data Fetching and Preprocessing Script: This script (fetch\_and\_preprocess\_data.py) will download initial movie data and clean it, generating main\_data.csv. This CSV file forms the basis for content-based recommendations.

python fetch\_and\_preprocess\_data.py

Train the Sentiment Analysis Model: The sentiment analysis model needs to be trained and saved.

If you have a sentiment.ipynb notebook: Open it in Jupyter Notebook/Lab and run all cells. It will save nlp\_model.pkl and tranform.pkl in the models/ directory.

If you converted it to a sentiment.py script:

python sentiment.py

(Note: The large reviews.csv dataset, which is used for training the sentiment model, is not included in this repository due to its size. It can be obtained from the IMDb 50K Movie Reviews Dataset on Kaggle. The sentiment script will guide you on where to place it.)

## 6. Running the Application

After completing the setup steps, you can run the Flask application:

python main.py

The application will typically run on http://127.0.0.1:5000/ or http://localhost:5000/. Open this URL in your web browser.

## 7. Project Structure

MovieRecommendationSystem/

├── main.py                          # Flask application entry point

├── fetch\_and\_preprocess\_data.py     # Script to fetch & preprocess movie data

├── sentiment.ipynb (or sentiment.py)# Jupyter Notebook/script for sentiment model training

├── requirements.txt                 # Python dependencies

├── README.md                        # This file

├── data/                            # Directory for processed data

│   ├── main\_data.csv                # Processed movie data for recommendations

│   └── (reviews.csv - external, for sentiment training)

├── models/                          # Directory for trained ML models

│   ├── nlp\_model.pkl                # Trained sentiment analysis model

│   └── tranform.pkl                 # TF-IDF vectorizer for sentiment analysis

├── static/                          # Static assets (CSS, JS, images)

│   ├── css/

│   │   └── style.css

│   └── js/

│       └── recommend.js

└── templates/                       # HTML templates

    ├── home.html

    └── recommend.html

## 8. Credits and Acknowledgements

The Movie Database (TMDb): All movie data, images, and reviews are sourced from the TMDb API.

NLTK (Natural Language Toolkit): Used for text preprocessing (stopwords, stemming).

scikit-learn: Utilized for TF-IDF vectorization, Cosine Similarity, and machine learning models.

Flask: The web framework used for the backend.

Bootstrap: Frontend styling framework.

jQuery: JavaScript library for DOM manipulation and AJAX.

## 9. License

This project is open-source and available under the MIT License. See the LICENSE file for more details.

## **Background Image**

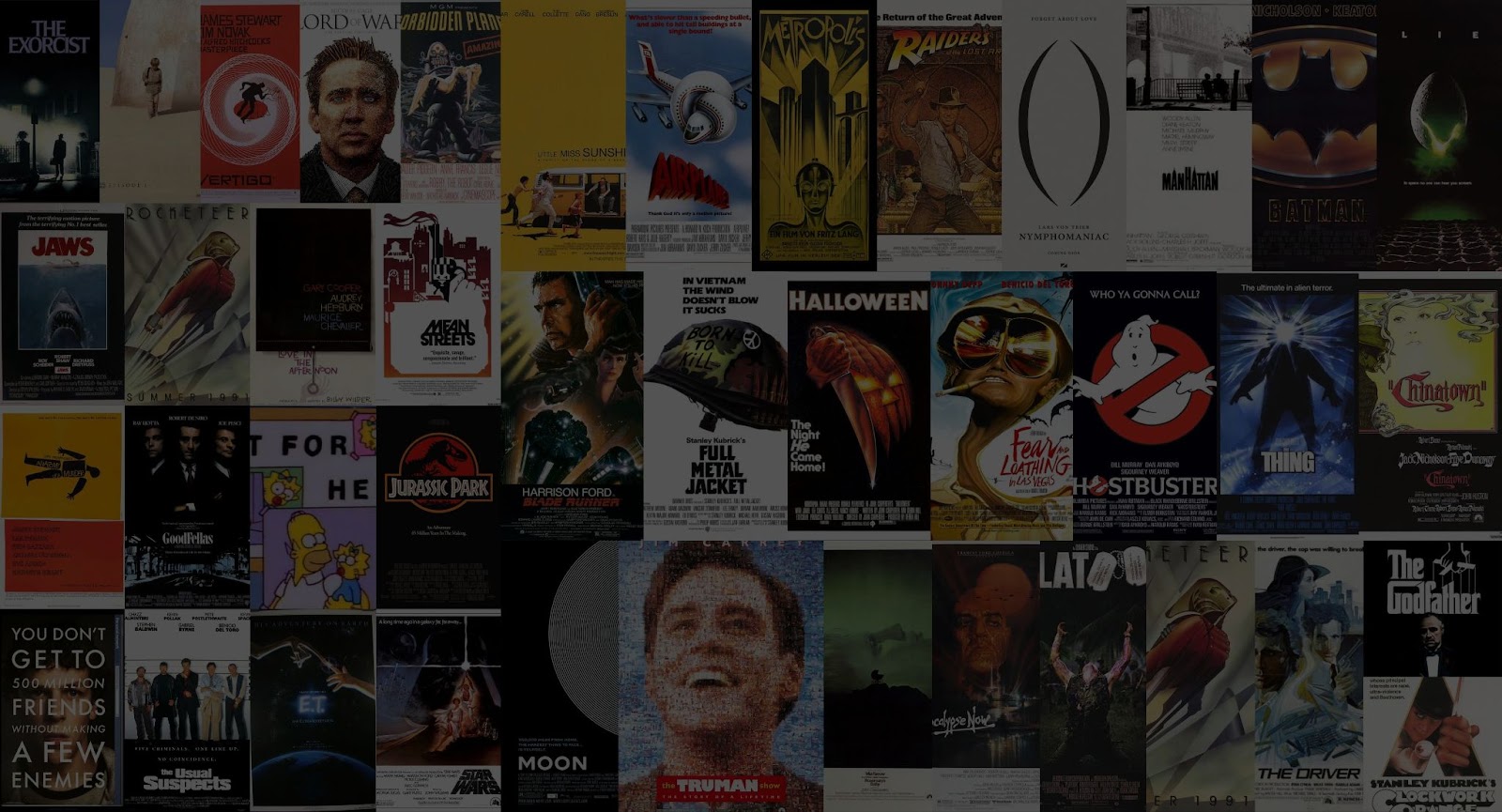


Figure A‑1 Background Image

## **System Architecture**

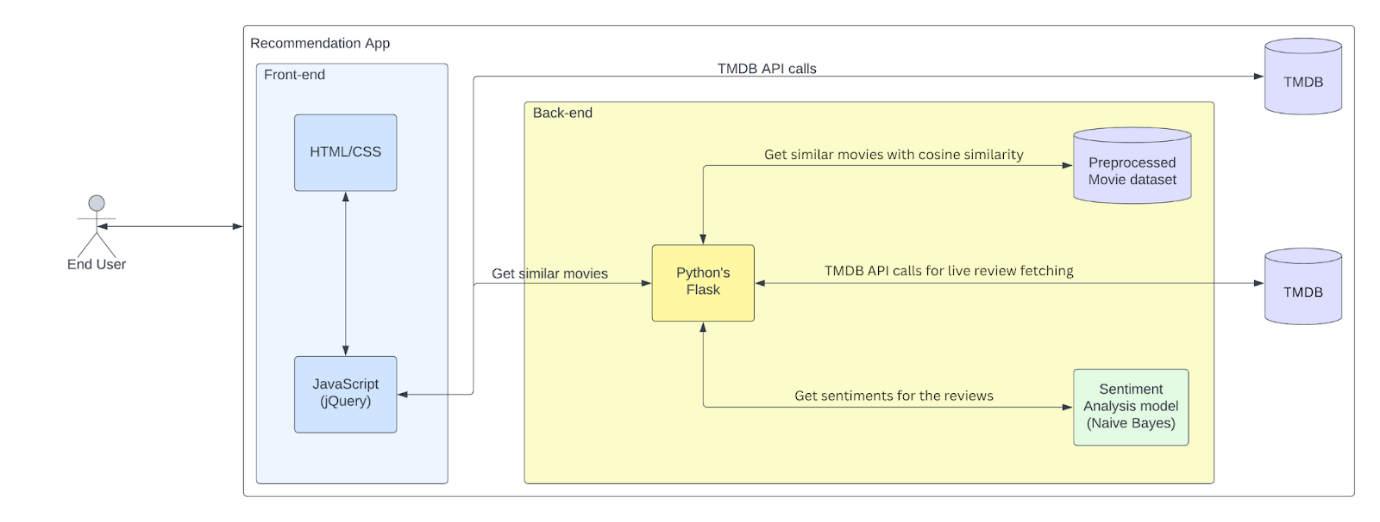


Figure A‑2 System Architecture

# **Appendix B: Screenshots**

## **Homepage / Landing Page**

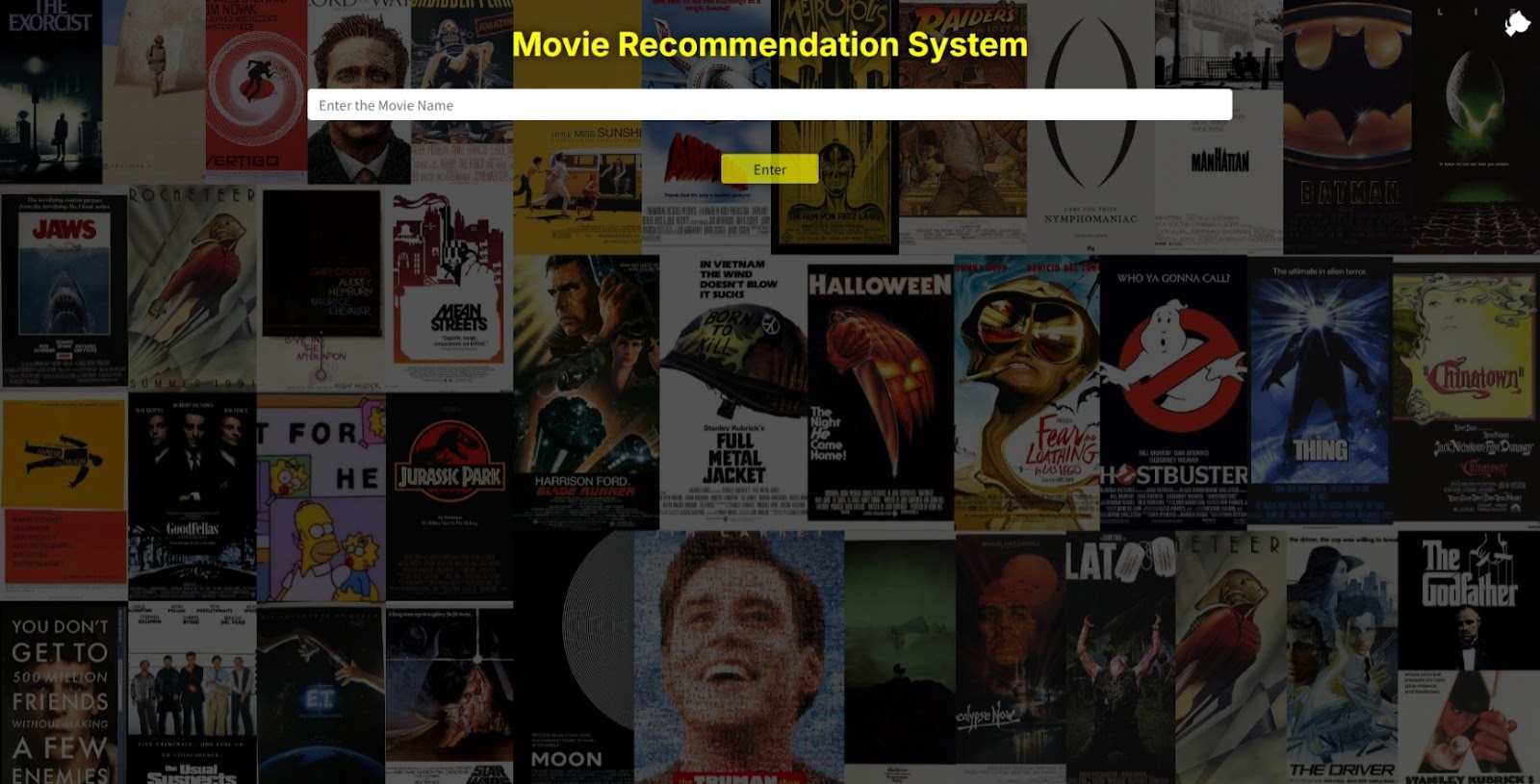


Figure B‑1 Homepage with Search Bar

## **Movie Search with Autocomplete/Suggestions**

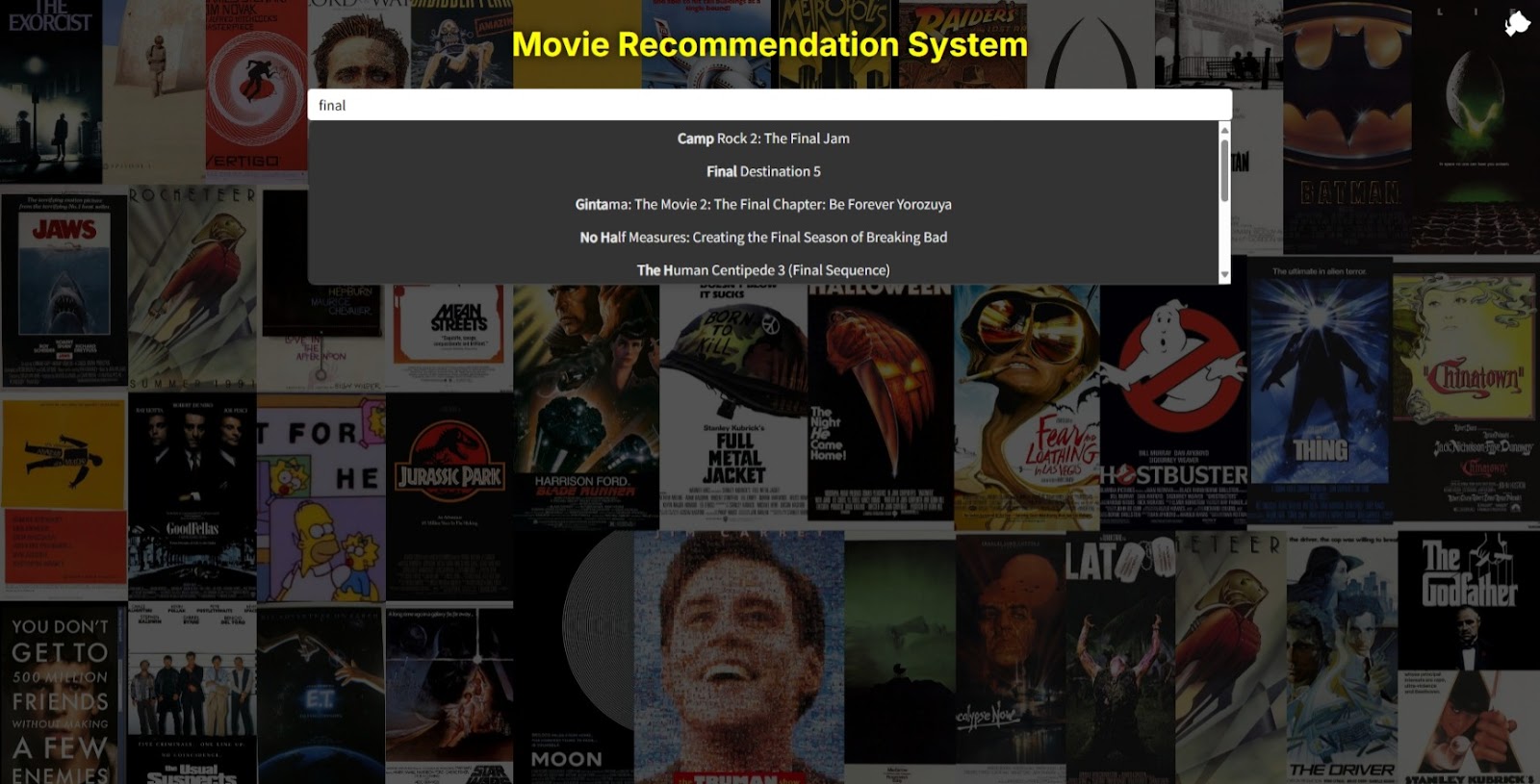


Figure B‑2 Homepage with Autocomplete/Suggestions

## **Movie Details Page (Final Destination Bloodlines)**

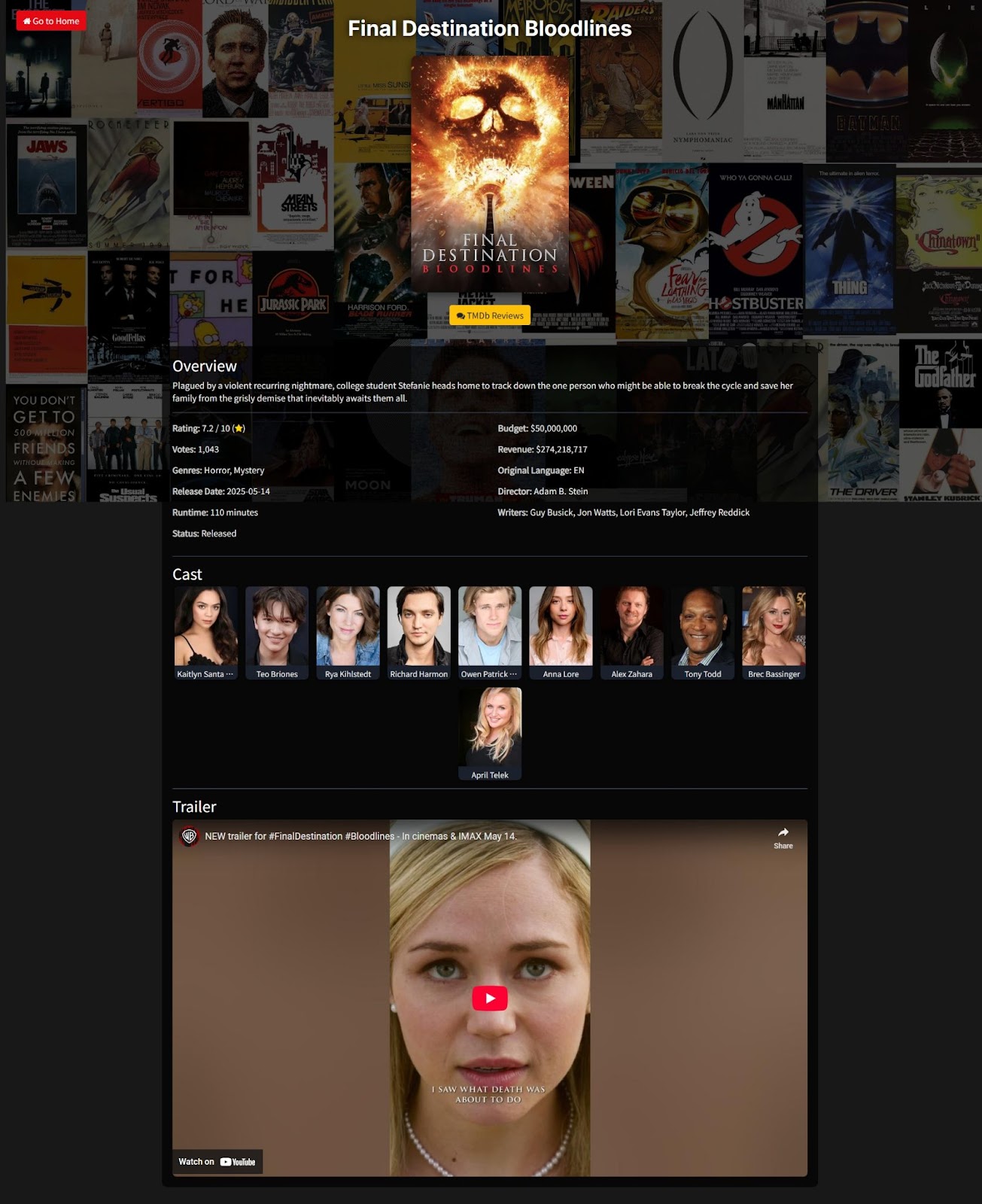


Figure B‑3 Final Destination Bloodline Movie Details Page

## **Movie Recommendations Display**

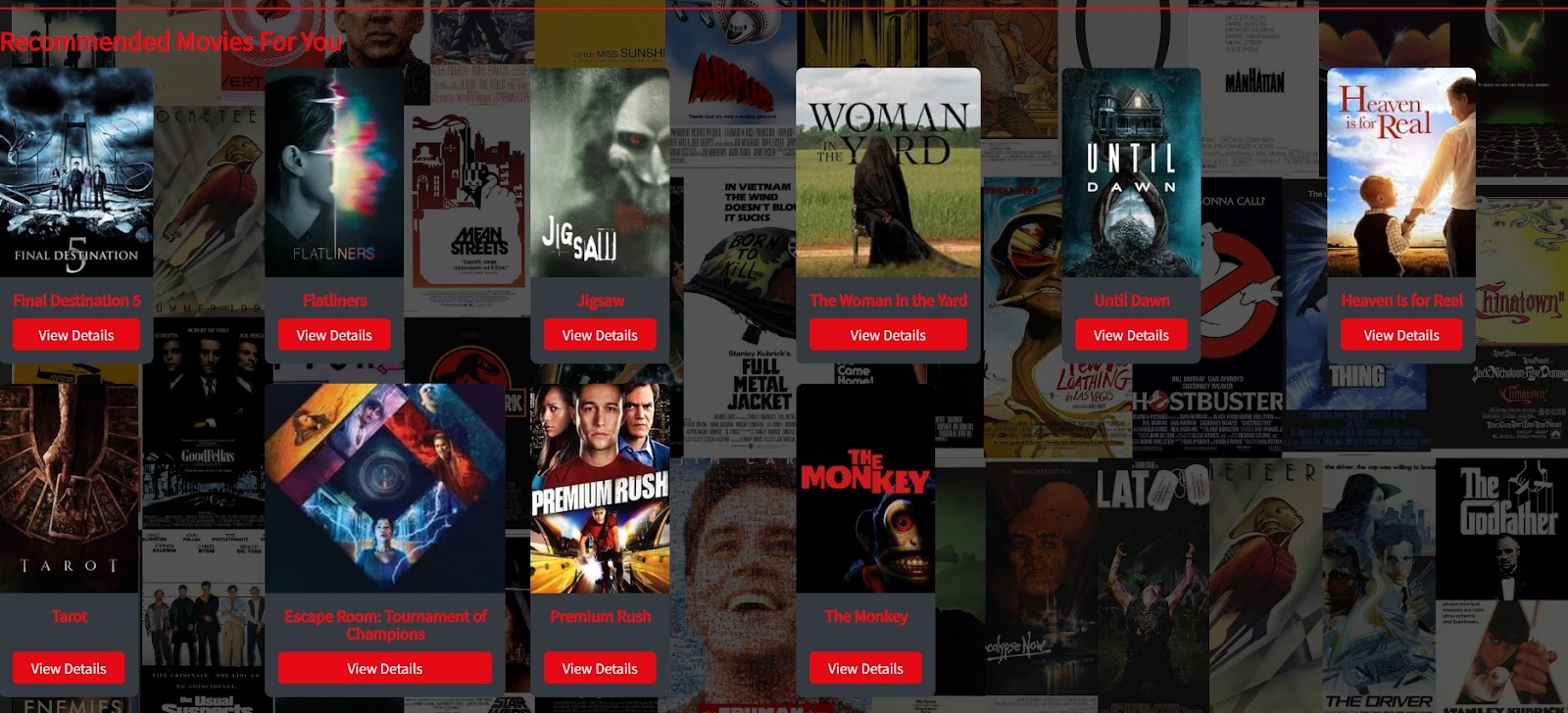


Figure B‑4 Movie Recommendations

## **Sentiment Analysis Results (Reviews)**

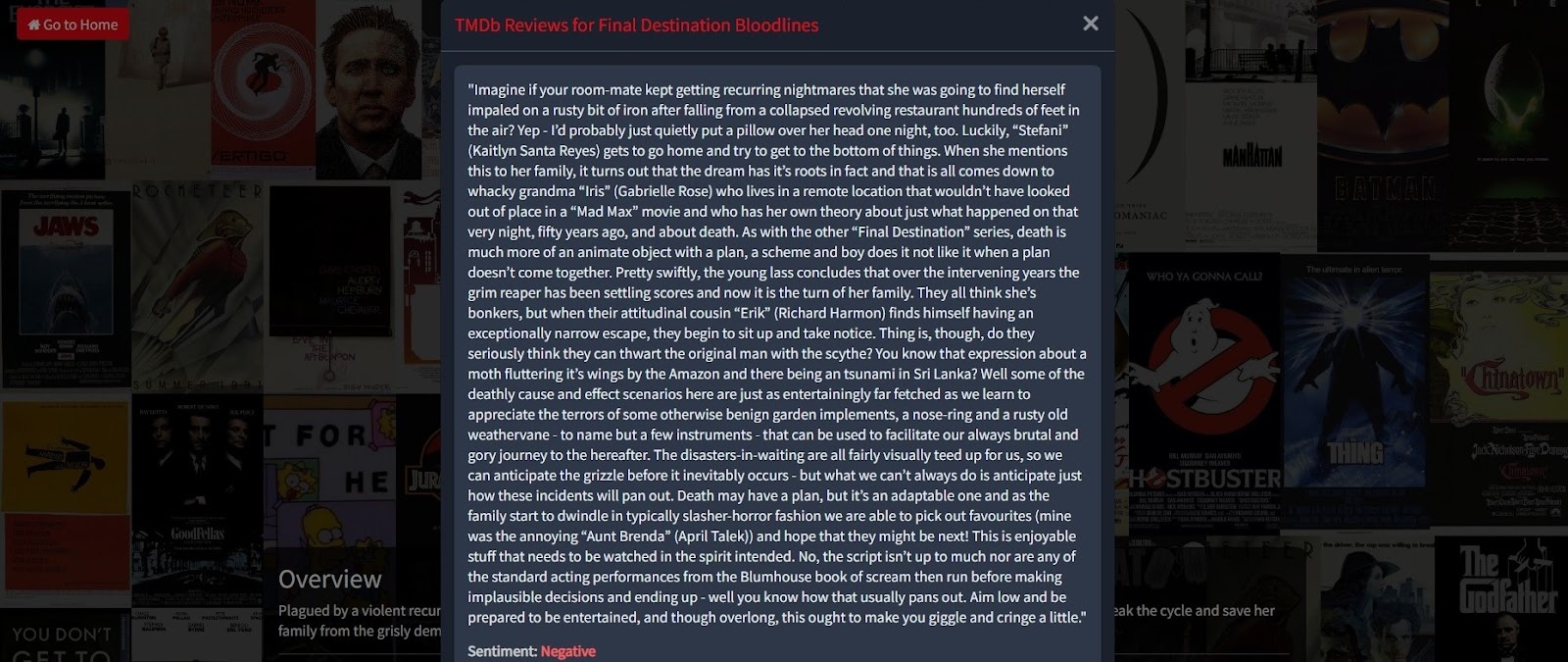


Figure B‑5 Negative Review

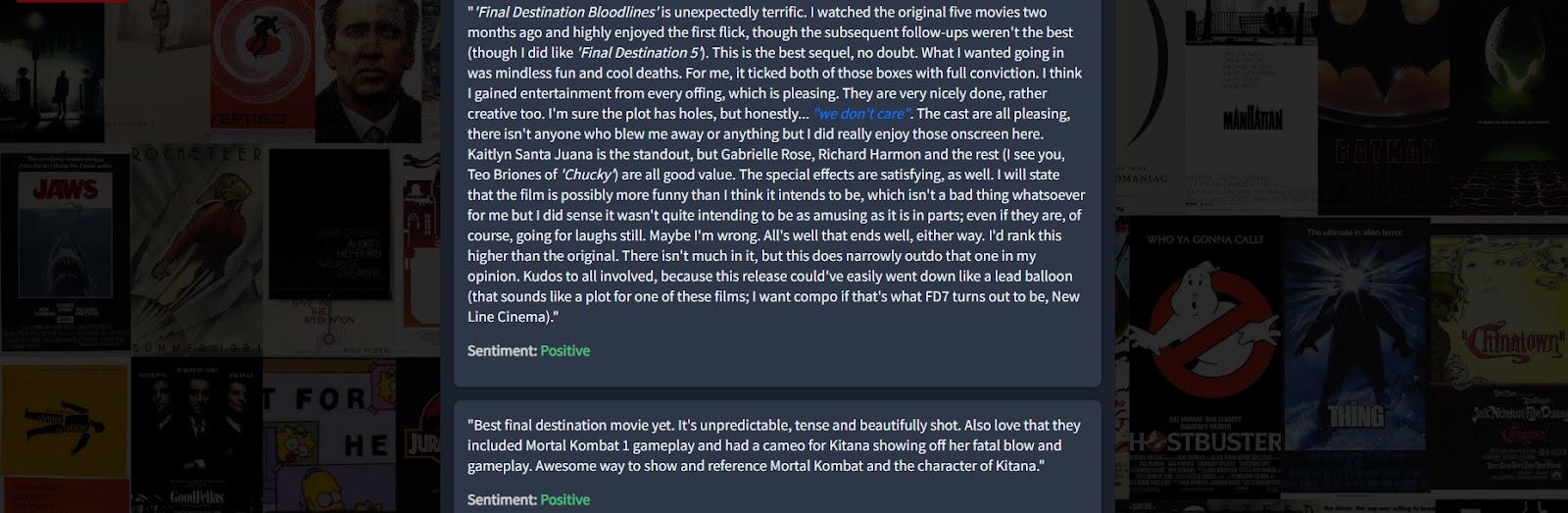


Figure B‑6 Positive Review

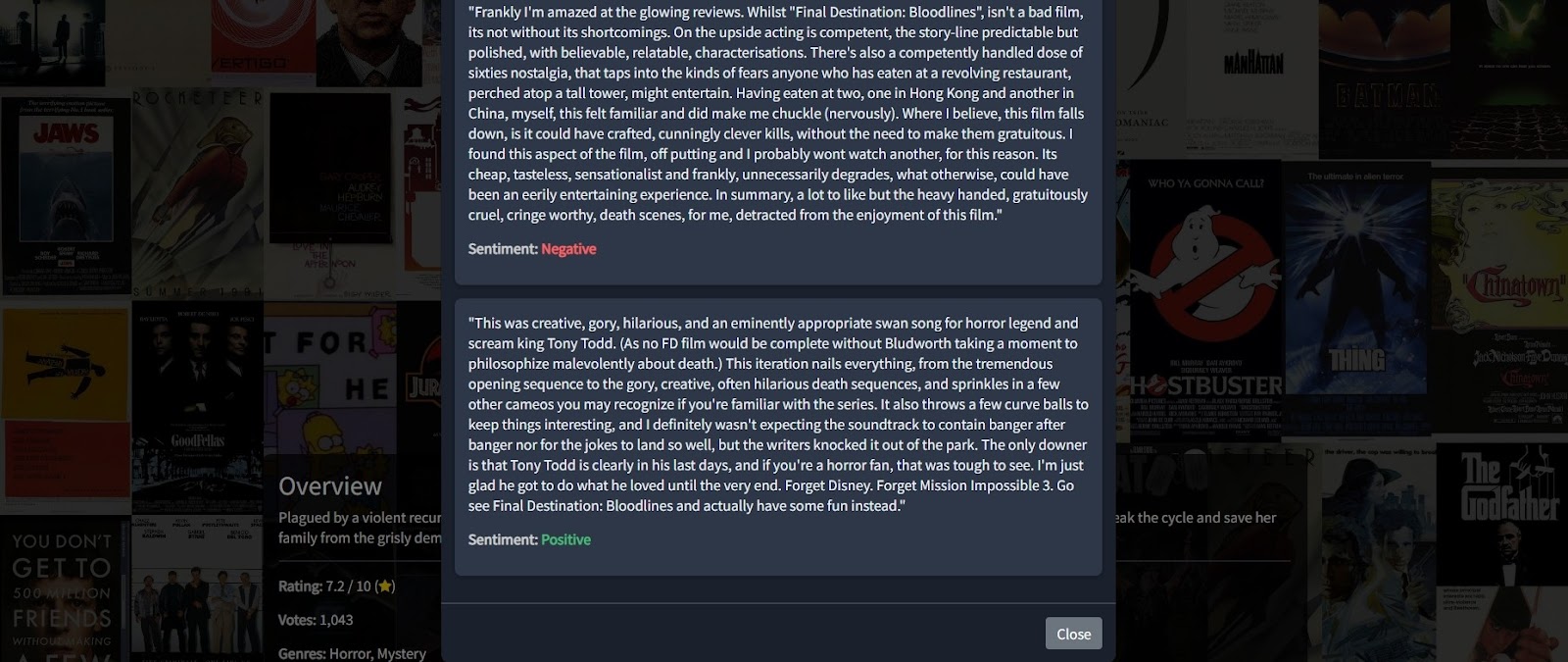


Figure B‑7 Negative & Positive Review

## **Actor Bio**

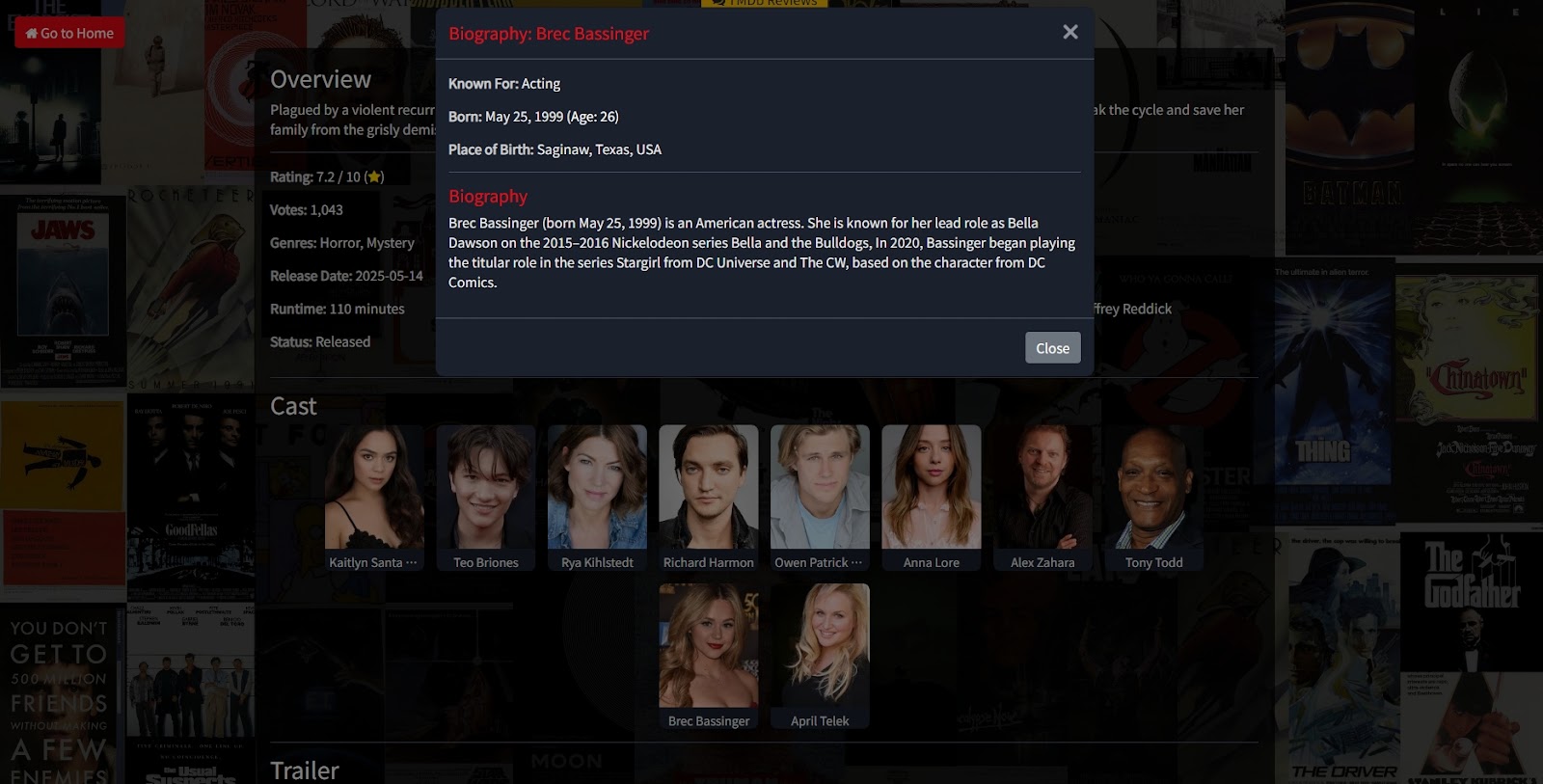


Figure B‑8 Bio of Actor

# **Appendix C: Sample Data**

## **Sample of main\_data.csv**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| movie\_title | director\_name | actor\_1\_name | actor\_2\_name | actor\_3\_name | genres | keywords | id | poster\_path | comb |
| Shrek Forever After | Mike Mitchell | Mike Myers | Eddie Murphy | Cameron Diaz | Comedy Adventure Fantasy Animation Family | witch sequel ogre | 10192 | /6HrfPZtKcGmX2tUWW3cnciZTaSD.jpg | mike mitchell mike myers eddie murphy cameron diaz comedy adventure fantasy animation family witch sequel ogre |
| Khaleja | Trivikram Srinivas | Mahesh Babu | Shafi | Prakash Raj | Action |  | 56986 | /iTwJRCbiMaIaA0gjozXeznfZelk.jpg | trivikram srinivas mahesh babu shafi prakash raj action |
| Naruto Shippuden the Movie: The Lost Tower | Masahiko Murata | Junko Takeuchi | Kazuhiko Inoue | Toshiyuki Morikawa | Adventure Action Animation | ninja shounen anime adventure | 50723 | /6e2YvN1tQK4xQHlmy7GJTuXOt2u.jpg | masahiko murata junko takeuchi kazuhiko inoue toshiyuki morikawa adventure action animation ninja shounen anime adventure |
| Housefull | Sajid Khan | Akshay Kumar | Arjun Rampal | Ritesh Deshmukh | Comedy Romance | love game | 58051 | /koPiIQ7evC5rer6modUDxoZjYV.jpg | sajid khan akshay kumar arjun rampal ritesh deshmukh comedy romance love game |
| Detective Conan: The Lost Ship in the Sky | Yasuichiro Yamamoto | Minami Takayama | Kappei Yamaguchi | Wakana Yamazaki | Animation Crime Mystery Action | detective anime | 97375 | /kthV4ZwXO37pfCFTbX1cHOYie4r.jpg | yasuichiro yamamoto minami takayama kappei yamaguchi wakana yamazaki animation crime mystery action detective anime |
| Nagavalli | P. Vasu | Venkatesh | Richa Gangopadhyay | Anushka Shetty | Action Fantasy Thriller Drama |  | 83154 | /5Z4RvsQAPSyx5ItW1UcpTjsRgae.jpg | p. vasu venkatesh richa gangopadhyay anushka shetty action fantasy thriller drama |
| Phineas and Ferb The Movie: Across the 2nd Dimension | Robert Hughes | Vincent Martella | Thomas Brodie-Sangster | Dee Bradley Baker | Animation Comedy Family TV Movie Science Fiction Adventure Action Music | monster sibling relationship one-sided love secret agent mad scientist alternate | 71689 | /updMFSOZwEftfkJRpozwBlRlYI2.jpg | robert hughes vincent martella thomas brodie-sangster dee bradley baker animation |

Table C‑1 Sample of main\_data.csv

## **Source of reviews.csv Dataset**

|  |  |
| --- | --- |
| review | sentiment |
| One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be hooked. They are right, as this is exactly what happened with me.<br /><br />The first thing that struck me about Oz was its brutality and unflinching scenes of violence, which set in right from the word GO. Trust me, this is not a show for the faint hearted or timid. This show pulls no punches with regards to drugs, sex or violence. Its is hardcore, in the classic use of the word.<br /><br />It is called OZ as that is the nickname given to the Oswald Maximum Security State Penitentary. It focuses mainly on Emerald City, an experimental section of the prison where all the cells have glass fronts and face inwards, so privacy is not high on the agenda. Em City is home to many..Aryans, Muslims, gangstas, Latinos, Christians, Italians, Irish and more....so scuffles, death stares, dodgy dealings and shady agreements are never far away.<br /><br />I would say the main appeal of the show is due to the fact that it goes where other shows wouldn't dare. Forget pretty pictures painted for mainstream audiences, forget charm, forget romance...OZ doesn't mess around. The first episode I ever saw struck me as so nasty it was surreal, I couldn't say I was ready for it, but as I watched more, I developed a taste for Oz, and got accustomed to the high levels of graphic violence. Not just violence, but injustice (crooked guards who'll be sold out for a nickel, inmates who'll kill on order and get away with it, well mannered, middle class inmates being turned into prison bitches due to their lack of street skills . | positive |
| If you like original gut wrenching laughter you will like this movie. If you are young or old then you will love this movie, hell even my mom liked it.<br /><br />Great Camp!!! | positive |
| Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his parents are fighting all the time.<br /><br />This movie is slower than a soap opera... and suddenly, Jake decides to become Rambo and kill the zombie.<br /><br />OK, first of all when you're going to make a film you must Decide if its a thriller or a drama! As a drama the movie is watchable. Parents are divorcing & arguing like in real life. And then we have Jake with his closet which totally ruins all the film! I expected to see a BOOGEYMAN similar movie, and instead i watched a drama with some meaningless thriller spots.<br /><br />3 out of 10 just for the well playing parents & descent dialogs. As for the shots with Jake: just ignore them. | negative |
| Kind of drawn in by the erotic scenes, only to realize this was one of the most amateurish and unbelievable bits of film I've ever seen. Sort of like a high school film project. What was Rosanna Arquette thinking?? And what was with all those stock characters in that bizarre supposed Midwest town? Pretty hard to get involved with this one. No lessons to be learned from it, no brilliant insights, just stilted and quite ridiculous (but lots of skin, if that intrigues you) videotaped nonsense....What was with the bisexual relationship, out of nowhere, after all the heterosexual encounters. And what was with that absurd dance, with everybody playing their stereotyped roles? Give this one a pass, it's like a million other miles of bad, wasted film, money that could have been spent on starving children or Aids in Africa..... | negative |

Table C‑2 Sample of reviews.csv

The 'reviews.csv' dataset, consisting of approximately 50,000 movie reviews used for training the sentiment analysis model, was obtained from the following source: \*\*Dataset Name:\*\* IMDb 50K Movie Reviews Dataset \*\*Source:\*\* Kaggle \*\*URL:\*\* <https://www.kaggle.com/datasets/lakshmi25npathi/imdb-50k-movie-reviews-dataset>

# **Appendix D: User Manual**

## **Introduction**

This manual provides step-by-step instructions for setting up, running, and interacting with the "Movie Recommendation System with Sentiment Analysis" project. It is designed for users who wish to test the application locally and understand its basic functionalities.

## **System Requirements**

To run this application, ensure your system meets the following requirements:

* **Operating System:** Windows 10/11, macOS, or Linux
* **Python:** Version 3.8 or higher (recommended)
* **pip:** Python package installer (usually comes with Python)
* **Git:** (Optional, for cloning the repository)
* **Web Browser:** Any modern browser (e.g., Google Chrome, Mozilla Firefox, Microsoft Edge, Apple Safari)

## **Setup Instructions**

Follow these steps to prepare your environment and get the project ready to run:

1. **Obtain Project Files:**

If using Git: Clone the project repository:  
 Bash  
git clone <https://github.com/samminga-akshay/Movie-Recommendation-System-with-Sentiment-Analysis.git>

Navigate into the project directory:  
 Bash  
cd Movie-Recommendation-System-with-Sentiment-Analysis

* *Alternatively, if not using Git:* Download the project as a ZIP file, extract it, and navigate into the extracted project directory using your command line.

1. **Create and Activate a Virtual Environment (Recommended):** A virtual environment isolates project dependencies.

Create the environment:  
python -m venv venv

* Activate the environment:

On Windows:  
.\venv\Scripts\activate

On macOS/Linux:  
source venv/bin/activate

**Install Project Dependencies:** With your virtual environment active, install all required

Python libraries:  
pip install -r requirements.txt

1. **Obtain and Configure TMDb API Key:** The application relies on The Movie Database (TMDb) API for movie information and reviews.
   * Go to [The Movie Database (TMDb) website](https://www.themoviedb.org/) and sign up for a free account.
   * Log in, go to your account settings, and request a developer API key.
   * **Crucial Step:** Open the files main.py and static/js/recommend.js in a text editor (like VS Code).
   * Locate the line TMDB\_API\_KEY = 'YOUR\_TMDB\_API\_KEY\_HERE' (in main.py) and const TMDB\_API\_KEY = 'YOUR\_TMDB\_API\_KEY\_HERE'; (in recommend.js).
   * **Replace 'YOUR\_TMDB\_API\_KEY\_HERE' with your actual TMDb API key in both files.**
2. **Prepare Data and Models:** The system requires preprocessed movie data and a trained sentiment analysis model.

**Run Data Preprocessing:** This script fetches and cleans initial movie data:  
python fetch\_and\_preprocess\_data.py

**Train Sentiment Model:** This script trains and saves the sentiment analysis model.  
python sentiment.py

* *(Note: The sentiment model training requires the 'reviews.csv' dataset. If not present, the script will prompt you or you can download it from* [*IMDb 50K Movie Reviews Dataset on Kaggle*](https://www.google.com/search?q=https://www.kaggle.com/datasets/lakshmi25npathi/imdb-50k-movie-reviews-dataset&authuser=2)*.)*

## **Running the Application**

Once all setup steps are complete, you can start the web application:

1. Ensure your virtual environment is activated (from step 3 of Setup Instructions).

Run the Flask application:  
python main.py

1. The application will typically start on http://127.0.0.1:5000/. Open this URL in your web browser.

## **How to Use the Application**

Once the application is running in your browser:

1. **Searching for Movies:**
   * On the homepage, use the search bar to find movies. As you type, an autocomplete feature will suggest movie titles.
   * Select a movie from the suggestions or press Enter to search.
2. **Viewing Movie Details:**
   * After selecting a movie, you will be directed to its details page. This page displays the movie's title, poster, plot summary, genres, release date, cast, and director.
3. **Getting Recommendations:**
   * On the movie details page, click the "Get Recommendations" button (or similar) to view a list of movies that are similar based on content-based filtering.
4. **Viewing Sentiment Analysis:**
   * On the movie details page, scroll down to the section displaying user reviews. The system will show individual reviews and their predicted sentiment (e.g., "Positive" or "Negative").

# **Appendix E: Glossary**

This glossary provides definitions for key technical terms and concepts used in this thesis, particularly those related to the Movie Recommendation System with Sentiment Analysis.

* **AJAX (Asynchronous JavaScript and XML):** A set of web development techniques that allows a web page to update asynchronously by exchanging small amounts of data with the server behind the scenes, without reloading the entire page.
* **API (Application Programming Interface):** A set of rules and protocols that allows different software applications to communicate with each other. In this project, the TMDb API is used to retrieve movie data.
* **Backend:** The server-side of a web application that handles data storage, business logic, and communication with databases and external APIs. In this project, Flask serves as the backend.
* **Content-Based Filtering:** A recommendation system technique that recommends items similar to those a user has liked in the past, based on the items' attributes or content (e.g., genres, cast, keywords).
* **Cosine Similarity:** A metric used to measure the similarity between two non-zero vectors in an inner product space. It is often used in information retrieval to compare document similarity or, in this case, movie similarity based on their feature vectors.
* **Dataset:** A collection of related data, typically organized in a structured format, used for analysis, training models, or other computational tasks.
* **Flask:** A lightweight Python web framework used for building web applications quickly and with minimal boilerplate code.
* **Frontend:** The client-side of a web application, comprising the user interface and user experience elements with which users directly interact (e.g., HTML, CSS, JavaScript).
* **Machine Learning (ML):** A subfield of artificial intelligence that focuses on developing algorithms allowing computers to learn from data without being explicitly programmed.
* **Model (Machine Learning Model):** A mathematical algorithm or set of algorithms trained on a dataset to recognize patterns or make predictions. In this project, a sentiment analysis model is used.
* **Natural Language Processing (NLP):** A field of artificial intelligence that enables computers to understand, interpret, and generate human language. It's crucial for text processing in sentiment analysis.
* **NLTK (Natural Language Toolkit):** A popular Python library for working with human language data, widely used for tasks such as tokenization, stemming, lemmatization, and sentiment analysis.
* **Pickle (.pkl file):** A Python module used for serializing and deserializing Python objects. It's used here to save and load trained machine learning models and transformers.
* **Preprocessing (Data Preprocessing):** The process of transforming raw data into an understandable format suitable for analysis or machine learning model training. This includes cleaning, normalization, and feature engineering.
* **Recommendation System:** A software system that predicts user preferences for items and recommends items that a user might like.
* **Sentiment Analysis:** The process of computationally identifying and categorizing opinions expressed in a piece of text, especially to determine whether the writer's attitude towards a particular topic, product, etc., is positive, negative, or neutral.
* **TF-IDF (Term Frequency-Inverse Document Frequency):** A numerical statistic that reflects how important a word is to a document in a collection or corpus. It is commonly used as a weighting factor in information retrieval and text mining.
* **TMDb (The Movie Database):** A community-built movie and TV show database that provides a rich API for developers to access comprehensive film information.
* **User Interface (UI):** The visual part of a computer application or website that allows users to interact with it.
* **Virtual Environment:** An isolated Python environment that allows you to manage dependencies for a specific project without interfering with other projects or the global Python installation.