MOVIFY: A MOVIE RECOMMENDATION SYSTEM

REPORT

Submitted by

Abhijit R 221801001

Faleel Mohsin F 221801010

Monish Raja Rathinam M 221801033

In partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)

THANDALAM

CHENNAI - 602105

2024 - 2025

ANNA UNIVERSITY: CHENNAI

BONAFIDE CERTIFICATE

Certified that this project report "**MOVIFY: A Movie Recommendation Sytem**" is the Bonafide work of "**Abhijit R (221801001), Faleel Mohsin F (221801010), Monish Raja Rathinam M (221801033)**" who carried out the project work under my supervision.

Submitted for the practical examination held on ____________________

SIGNATURE	**SIGNATURE**
Dr. J.M. Gnanasekar	Dr. Manoranjini J
Professor and Head	Associate Professor,
Department of Artificial Intelligence and Data Science	Department of Artificial Intelligence and Data Science
Rajalakshmi Engineering College	Rajalakshmi Engineering College
Chennai – 602 105	Chennai – 602 105

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ABSTRACT

MOVIFY, the movie recommendation system enhances personalization and user engagement, expanding beyond the functionalities of existing platforms. While systems like MovieLens, TasteDive, and Letterboxd focus on collaborative and content-based filtering, they often lack comprehensive multi-language support, real-time data integration, and advanced customization features. This system, developed using Streamlit, incorporates the IMDB API for up-to-date movie details, a fuzzy search feature for easier movie discovery, and a unique nostalgia-based recommendation option that suggests films relevant to the user's formative years. Additional features, including a watchlist and email notifications, aim to increase user interaction and retention. Together, these capabilities provide a more interactive, accessible, and personalized movie discovery experience for a wide range of users.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 GENERAL

A movie recommendation system is a specialized algorithm that predicts and presents movie suggestions to users based on their preferences, viewing history, or similarities among movies. These algorithms, which make it simple for consumers to browse vast libraries and find fresh content that suits their interests, are now crucial in streaming platforms, movie databases, and entertainment applications. These algorithms are a fundamental component of many digital platforms since they provide personalized recommendations that contribute to a more interesting and pleasurable watching experience.

Usually, a combination of contentbased filtering, collaborative filtering, and hybrid techniques is used to create recommendation systems. In order to identify commonalities amongst users who have rated or viewed the same films, collaborative filtering looks for patterns in user activity. For instance, one user's highly regarded films may be suggested to another if their viewing habits are comparable. Contentbased filtering, on the other hand, suggests films that are comparable to those a user has already enjoyed by concentrating on the qualities of the films itself, such as genre, director, or cast. By utilizing the advantages of both approaches, hybrid systems integrate aspects of each to produce recommendations with greater accuracy and variety.

In order to provide tailored movie recommendations, this project integrates a fully functional movie recommendation system that blends data science and user experience design. It comes with a web application built using Streamlit that enables direct user interaction with the system, as well as a Jupyter notebook for data preparation and model construction. The recommendation engine creates customized recommendations by calculating movie similarity scores based on data analysis. The web application improves the system's usability and aesthetic appeal by retrieving posters and movie details from an external source in addition to providing movie suggestions.

With its userfriendly UI and login system, this app allows users to browse relevant films and access their suggestions. Every time a user logs in, the system ensures that they receive prompt and pertinent recommendations by using precomputed similarity data to expedite the recommendation process and decrease response time. Movie posters and other visual components give the experience an interactive element that makes finding new movies simpler and more pleasurable for viewers.

In conclusion, this movie recommendation project demonstrates the power of combining datadriven algorithms with an intuitive interface to create an engaging user experience. By helping users find movies they might enjoy based on their unique preferences, the system enhances the movie discovery process, making it easier for users to navigate large catalogs and fostering a personalized connection with the content.

1.2 OBJECTIVES OF THE STUDY

It is crucial to concentrate on improving multiple aspects of the platform in order to provide movie fans with a smooth and delightful experience. We can make it simpler for people to find and enjoy material that suits their interests by enhancing important elements like search functionality, personalized suggestions, and user ease. By making sure customers can discover what they're searching for quickly and simply, these enhancements hope to not only provide a seamless, effective, and entertaining experience but also keep people returning for more. Through wellconsidered upgrades and new features, the following main objectives are intended to maximize the user experience and offer value.

1. Improve Movie Search

- Make it easier to search for movies by genre, release date, or rating.
- Include an autocomplete feature to suggest search terms as users type.
- Add advanced filters, like runtime, to refine search results further.

2. Provide Instant Recommendations

- Show movie suggestions immediately after the user interacts with the system.
- Offer a “random” suggestion for users who want something new without any specific preference.
- Allow users to save or “bookmark” recommendations for future viewing.

3. Add More Movie Information

- Include details like actors, directors, and genres for each movie.
- Add user reviews and ratings to provide more perspectives on the movie.
- Display movie trailers and sneak peeks on the movie page for better decision making.

4. Personalize Recommendations

- Suggest movies based on the user’s past viewing habits.
- Allow users to finetune recommendations by liking or disliking suggested movies.
- Offer personalized lists, such as "Top Picks for You" or "Trending Now."

5. Enhance Movie Pages

- Add features like trailers, reviews, and ratings to each movie page.
- Include a “similar movies” section to help users discover more films they might enjoy.
- Display a brief movie summary or plot description on the page for quick viewing.

6. Support Multiple Languages

- Allow users to use the app in their preferred language.
- Provide subtitle options in multiple languages for international users.
- Ensure that all movie descriptions, titles, and metadata are translated accurately.

7. Simplify Login and Session Management

- Make logging in easy and keep users' preferences saved.
- Allow users to log in with social media accounts for faster access.
- Provide an option for users to stay logged in across devices.

8. Speed Up Recommendations

- Ensure movie suggestions appear quickly, in under 2 seconds.
- Use caching to avoid delays when loading previously viewed recommendations.
- Optimize the algorithm to offer suggestions based on the most recent user actions.

9. Allow Social Sharing

- Let users share their favorite movies on social media.
- Include options to share directly with friends within the app.
- Provide a "share via link" option to make it easier for users to recommend movies.

10. Add a Watchlist

- Allow users to save movies they want to watch later.
- Send notifications when a movie on the watchlist is about to be removed or becomes available for streaming.
- Let users categorize their watchlist (e.g., "To Watch," "Favorites," "Recently Added").

CHAPTER 2

REVIEW OF LITERATURE

2.1 INTRODUCTION

- With the exponential growth of digital content, movie recommendation systems have become essential tools for enhancing user experience on streaming platforms by providing personalized suggestions. A recommendation system aims to understand and anticipate user preferences, reducing the vast selection of movies to a tailored subset that aligns with individual tastes.
- Over the years, various approaches have been developed to improve the effectiveness of these systems, including content-based filtering, collaborative filtering, and hybrid models that combine multiple methods. This literature survey explores key studies and methodologies that have shaped the evolution of recommendation systems.
- Researchers have examined the strengths and limitations of each approach, often finding that hybrid systems, which leverage both item attributes and user behavior, outperform single-method models. Collaborative filtering, especially, has been widely adopted for its ability to predict user preferences based on the behaviors of similar users. Additionally, more advanced methods, such as natural language processing (NLP) and machine learning algorithms, have introduced significant improvements in recommendation accuracy and user satisfaction.
- The following survey summarizes influential studies in the field, focusing on the methodologies applied and their impact on the accuracy, diversity, and efficiency of recommendation systems. This overview sets the foundation for understanding current trends and best practices in developing effective movie recommendation systems.

2.2 LITERATURE REVIEW

Author(s) & Year	Approach	Methodology	Key Findings
Harper & Konstan (2015)	Collaborative Filtering	Analysis using the MovieLens dataset, focusing on user behavior and historical data	Established MovieLens as a benchmark for collaborative filtering research; highlighted the importance of user feedback
Kumar & Thakur (2019)	Hybrid Approach	Combined content-based and collaborative	Demonstrated that hybrid systems yield

		filtering for improved personalization	higher accuracy and diverse recommendations compared to single-method approaches
Bobadilla et al. (2013)	Recommender Systems Survey	Comprehensive review of content-based, collaborative, and hybrid methods in recommendation	Found hybrid approaches to be more effective in most cases; identified collaborative filtering as the most widely used
Musto et al. (2015)	Content-Based Filtering	Comprehensive review of content-based, collaborative, and hybrid methods in recommendation	Found hybrid approaches to be more effective in most cases; identified collaborative filtering as the most widely used
Chen, Zhang, & Cao (2014)	Collaborative Filtering	Used a nearest-neighbors algorithm to identify user similarity based on viewing history	Proved the effectiveness of collaborative filtering, emphasizing the importance of user behavior in generating recommendations

CHAPTER 3
SYSTEM OVERVIEW

3.1 EXISTING SYSTEM

Current movie recommendation systems, such as Netflix, Amazon Prime Video, and Letterboxd, offer movie suggestions based on user preferences, watching history, and sometimes demographic information. These systems primarily use collaborative filtering, content-based filtering, or a hybrid approach combining the two. Collaborative filtering leverages user interactions, comparing preferences across users to suggest new content. Content-based filtering, on the other hand, analyzes attributes of the movies—such as genres, directors, and actors—to recommend movies similar to those a user has previously enjoyed. Advanced platforms also use deep learning models to analyze viewing behavior at a more granular level.

However, the majority of existing systems lack personalized features tailored to specific demographics or language preferences. For instance, they may not cater well to regional languages or niche preferences without extensive customization. Furthermore, many systems are restricted to a platform-specific catalog and don't allow the exploration of a broader, cross-platform movie library. Although these platforms excel at recommending popular or trending movies, they may overlook lesser-known titles that might align with a user's interests. Additionally, some systems have limited user engagement features, such as customizable watchlists or age-based nostalgia recommendations, which could enhance the user experience.

3.2 PROPOSED SYSTEM

The proposed movie recommender system aims to bridge these gaps by offering a more personalized and engaging experience. Built using Streamlit, the system allows users to explore movies across multiple languages (International, Indian, and Tamil) with recommendations based on similarity analysis and user-specific preferences. It incorporates a feature that enables users to maintain a watchlist for easy reference, and notifications are sent to remind them of movies they might enjoy or upcoming releases relevant to their tastes. Through initial login preferences, search and watch history, the system refines recommendations over time, offering a tailored movie discovery experience.

A unique addition in this system is nostalgia-based recommendations, which suggest movies from a user's formative years by calculating their age during login. This feature leverages the emotional appeal of revisiting classic movies, creating a sense of connection and nostalgia for users. Furthermore, by implementing fuzzy search and leveraging the OMDb API, the system ensures that users can retrieve detailed movie information—including directors, cast, plot, and posters—even if there are slight spelling discrepancies in their search terms. Using the nearest neighbors algorithm, the system recommends movies based on similarity to a selected title, making it easy for users to discover films they may not have considered otherwise. Overall, this proposed system not only enhances the user experience with personalization and engagement features but also broadens access to a diverse movie catalog that includes regional and nostalgic content.

Overview of proposed system

This software is a comprehensive online platform for streaming movies, designed to offer users easy access to a vast library of films from multiple genres and languages. With an intuitive and userfriendly interface, the platform enables movie enthusiasts to explore, discover, and watch content seamlessly. The software provides a robust experience, combining personalized recommendations with smooth streaming quality to ensure viewers find content they'll enjoy.

Key Features

1. Advanced Search Functionality

- Allows users to filter movies by genre, language, release year, popularity, and more. This helps users find specific titles or discover new content based on their preferences.

2. Personalized Recommendations

- Offers tailored movie suggestions based on the user's watch history and ratings. The recommendation engine enhances user engagement by showing relevant content that aligns with individual tastes.

3. Watchlist and Favorites

- Enables users to save movies they are interested in watching later or mark favorites to easily revisit their top picks. This feature organizes users' content preferences, making their viewing experience more enjoyable.

4. HighQuality Streaming Options

- Supports multiple streaming quality options, including HD and 4K, to cater to different internet speeds and viewing preferences. The adaptive streaming ensures a bufferfree experience across devices.

5. CrossPlatform Compatibility

- Accessible on various devices such as smartphones, tablets, and smart TVs. This flexibility allows users to enjoy movies on their preferred devices anytime, anywhere.

6. User Profiles

- Allows users to create multiple profiles within one account, each with its own personalized settings and recommendations. Ideal for households, where each family member can have a unique experience.

7. Parental Controls

- Provides customizable parental controls to restrict access to ageappropriate content, ensuring a safe viewing environment for younger users.

8. InApp Ratings and Reviews

- Lets users rate and review movies within the app, helping other users make informed viewing decisions. This feature fosters a community of movie lovers who can share their insights and opinions.

9. Offline Viewing

- Allows users to download movies for offline viewing, perfect for watching on the go or in areas with limited internet connectivity. This feature makes the platform accessible and convenient for travel or remote use.

10. Movie Trailers and Previews

- Offers trailers and previews for most movies, allowing users to get a quick overview before deciding to watch. This helps users discover movies more effectively and builds anticipation for upcoming releases.

By integrating these features, the software provides a rich and enjoyable user experience, making it an essential tool for movie lovers who seek a wide selection, convenience, and tailored content at their fingertips.

3.2.1 TECHNICAL SPECIFICATION

This movie streaming platform is built using Python and leverages key libraries to deliver a responsive and personalized user experience. Here’s a breakdown of the main technologies used and how they contribute to the platform:

Key Technologies

1. Streamlit

Used as the primary framework for creating the interactive web application. Streamlit enables fast and easy development of the user interface, allowing users to search, browse, and interact with the movie content in a visually appealing and userfriendly environment.

2. Pickle

Used to save and load data models and preprocessed data efficiently. Pickle is essential for quickly storing recommendation models and user preference data, making it possible to provide personalized suggestions to users without the need to retrain models each time the application is restarted.

3. Requests

Handles HTTP requests within the platform, particularly for fetching data from APIs if any external movie databases or metadata resources are integrated. This library allows seamless communication with external services to keep movie information up to date.

4. FuzzyWuzzy

Employed for flexible and robust text matching and search functionality. FuzzyWuzzy helps improve search accuracy by matching movie titles and genres even when users have slight typos or provide partial inputs, enhancing the user experience.

5. Functools

This library provides higherorder functions that improve code efficiency and readability. Functools allows for efficient caching of frequently used data and simplifies function management, ensuring smooth and rapid response times as users navigate the platform.

6. Sklearn Nearest Neighbors

This machine learning algorithm from the ScikitLearn library underpins the recommendation engine by suggesting movies similar to those users have already watched or rated highly. Using nearest neighbors enables accurate, contentbased recommendations, enhancing user engagement by offering relevant movie suggestions.

ADVANTAGES:

Efficient User Interaction: Streamlit's UI components make for a highly interactive platform, allowing users to browse, search, and access movie content with minimal latency.

Personalized Recommendations: Sklearn's nearest neighbors algorithm provides dynamic and relevant movie suggestions, making the platform feel customtailored to each user's preferences.

Robust Search: FuzzyWuzzy ensures users can easily find movies, even with minor spelling errors or incomplete information.

Reliable Data Handling: Pickle streamlines the storage and retrieval of recommendation models and data, enabling a smooth experience without delays from repetitive model training or data processing.

By combining these libraries, the platform not only serves as a movie streaming service but also offers a highly intuitive, personalized, and reliable experience for all users.

3.3 LANGUAGES:

3.3.1 PYTHON:

Python is a highlevel, interpreted programming language renowned for its simplicity, readability, and versatility. Created by Guido van Rossum and first released in 1991, Python has since gained widespread adoption and popularity across various domains, including web development, data science, artificial intelligence, scientific computing, automation, and more. Here are some key characteristics and features of Python:

1. **Easy to Learn and Read**: Python's syntax emphasizes readability and clarity, making it accessible for beginners and enjoyable for experienced developers. Its straightforward and concise syntax reduces the learning curve and encourages good coding practices.
2. **Versatile and MultiPurpose**: Python is a generalpurpose programming language, meaning it can be used for a wide range of applications. Whether you're building web applications, analyzing data, scripting automation tasks, or developing machine learning models, Python provides powerful tools and libraries to support diverse use cases.
3. **Interpreted and Interactive**: Python is an interpreted language, which means that code is executed line by line by an interpreter rather than compiled into machine code beforehand. This allows for interactive development and rapid prototyping, as code changes can be immediately tested and executed without the need for compilation.

4. **Dynamic Typing and Strong Typing**: Python is dynamically typed, meaning variable types are inferred at runtime and can change during execution. However, Python is also strongly typed, enforcing strict type checking to prevent unintended type errors and ensure code reliability.
5. **Extensive Standard Library**: Python comes with a comprehensive standard library that provides readytouse modules and functions for common tasks such as file I/O, networking, data manipulation, and more. This extensive library ecosystem simplifies development and reduces the need for external dependencies.
6. **Rich Ecosystem of ThirdParty Libraries and Frameworks**: In addition to its standard library, Python boasts a vibrant ecosystem of thirdparty libraries and frameworks contributed by the community. These libraries cover a wide range of domains, including web development (Django, Flask), scientific computing (NumPy, SciPy), data analysis (Pandas), machine learning (TensorFlow, PyTorch), and more.
7. **Platform Independence**: Python is platformindependent, meaning code written in Python can run on various operating systems, including Windows, macOS, Linux, and more. This portability ensures that Python applications can be deployed and executed across diverse environments with minimal modifications.
8. **Community and Support**: Python has a large and active community of developers, enthusiasts, and contributors who contribute to its ongoing development and maintenance. This vibrant community provides ample resources, documentation, tutorials, and forums for learning and support.

Python is a versatile, userfriendly, and powerful programming language that excels in a wide range of applications. Its simplicity, readability, extensive library ecosystem, and active community make it an ideal choice for beginners and seasoned developers alike.

3.3.2 PYTHON SERVES AS AN ASSET

Python plays a crucial role in the movie recommendation System project, offering a wide array of functionalities and benefits that contribute to the development and operation of the system:

1. **Backend Development**: Python is wellsuited for backend development due to its simplicity, readability, and extensive library support. Backend components of the

ecommerce platform, such as serverside logic, data processing, and API development, can be efficiently implemented using Python frameworks like Django or Flask.

2. **Web Development**: Python frameworks like Django and Flask are commonly used for web development, providing tools and utilities for building robust and scalable web applications. With Python, developers can create userfriendly interfaces, implement authentication mechanisms, and handle HTTP requests and responses, essential for an ecommerce platform's frontend and backend integration.
3. **Automation and Scripting**: Python's scripting capabilities are valuable for automating repetitive tasks, such as data processing, report generation, and system maintenance. Automation scripts can streamline administrative tasks, enhance productivity, and ensure the smooth operation of the ecommerce system.
4. **Data Analysis and Insights**: Python's extensive ecosystem of data analysis libraries (e.g., Pandas, NumPy) enables developers to analyze and derive insights from ecommerce data, such as sales trends, customer behavior, and inventory management. These insights can inform strategic decisions, optimize marketing campaigns, and improve the overall performance of the ecommerce platform.
5. **Integration with ThirdParty Services**: Python's flexibility and compatibility with various APIs make it suitable for integrating thirdparty services into the movie recommendation system. Whether integrating payment gateways, shipping providers, or analytics tools, Python facilitates seamless communication and integration with external services, enhancing the functionality and capabilities of the system.
6. **Community and Support**: Python benefits from a large and active community of developers, enthusiasts, and contributors who provide resources, documentation, and support. Leveraging Python's communitydriven ecosystem, developers can access tutorials, forums, and libraries to address challenges, learn best practices, and accelerate development efforts.

Python's versatility, ease of use, extensive library ecosystem, and active community support make it an indispensable tool for developing and operating the Movie recommendation System project. Whether it's backend development, web development, automation, data analysis, or integration with thirdparty services, Python empowers developers to build a robust, scalable, and feature rich movie recommendation platform.

3.3.3 PYTHON FOR FRONTEND

While Python is predominantly used for backend development, it can still play a role in frontend development for the Movie recommendation System project through various means:

1. **Template Engines**: Python web frameworks like Django come equipped with powerful template engines (e.g., Django Template Language) that allow developers to generate dynamic HTML content seamlessly. Python code embedded within templates can handle logic for rendering product listings, user profiles, shopping carts, and other frontend components.
2. **API Integration**: Python can facilitate communication between the frontend and backend of the ecommerce platform by serving as an intermediary for API requests and responses. Pythonbased backend services can expose RESTful APIs that deliver data to the frontend, enabling dynamic updates and interactions without page reloads.
3. **ClientSide Scripting**: Although JavaScript is typically the language of choice for clientside scripting in web development, Python can still be employed for certain clientside tasks using libraries like Brython (Python implementation for the browser). While limited in comparison to JavaScript, Brython enables developers to write Python code directly in HTML pages for frontend interactions.
4. **Data Processing**: Python's data processing capabilities can be leveraged in the frontend to manipulate and transform data before rendering it to users. Libraries like Pandas or NumPy can assist in performing calculations, filtering data, or generating visualizations within the browser, enhancing the user experience with dynamic and interactive content.
5. **Build Tools and Task Runners**: Pythonbased build tools and task runners (e.g., Fabric, Invoke) can streamline frontend development workflows by automating repetitive tasks such as compiling assets, optimizing images, or running tests. These tools complement frontend frameworks and libraries to enhance productivity and maintain code quality.

While Python may not be as prevalent in frontend development as JavaScript, it can still complement frontend efforts in certain scenarios, particularly when integrated with Pythonbased web frameworks or utilized for specific frontend tasks

3.3.4 PYTHON FOR MACHINE LEARNING

Python is a popular language for building machine learning applications, including recommendation systems, due to its rich ecosystem of libraries and tools that simplify data processing, model training, and deployment. Here's a breakdown of how Python helps in developing a movie recommendation system:

1. Data Collection and Preprocessing

Data Handling: Python has libraries like Pandas and NumPy that make it easy to collect, clean, and preprocess data. In a movie recommendation system, these libraries can be used to handle large datasets, like user ratings, movie details, genres, and viewing history.

Text Processing: When dealing with text data such as movie titles or descriptions, Python's NLTK and FuzzyWuzzy libraries help clean, tokenize, and match text efficiently. For instance, FuzzyWuzzy can help match movie titles with typos or slight differences in spelling, which improves search functionality.

2. Building the Recommendation Model

Machine Learning Libraries: Python's ScikitLearn library provides a variety of algorithms that can be used for recommendation systems. For instance, the Nearest Neighbors algorithm, often used in collaborative and contentbased filtering, identifies similar movies based on user ratings or movie features. ScikitLearn makes it simple to train, tune, and evaluate these models.

Data Structures: Python has builtin support for lists, dictionaries, and sets, which can store and manage user preferences, watch histories, and other data efficiently. This makes it easier to compute similarities or differences between movies or users when building recommendation logic.

3. Model Training and Optimization

Flexibility with Custom Algorithms: Python allows developers to create custom recommendation algorithms by combining multiple methods, such as collaborative filtering, contentbased filtering, and hybrid models. Python’s flexible syntax makes it easy to experiment with different combinations to improve recommendations.

Library Integration: Python integrates well with deep learning frameworks like TensorFlow and PyTorch, making it possible to add more complex, neural networkbased recommendation methods. These methods, such as embedding models, can capture more intricate patterns in user preferences.

4. Deployment and User Interaction

Web Frameworks: Python’s Streamlit library allows developers to create web applications for machine learning models, making it easy to turn a recommendation model into an interactive app. Users can browse recommendations and interact with the system directly through the app.

Pickle: Python’s Pickle library is invaluable for storing and loading pretrained models and user data efficiently. This makes it possible to deploy recommendation models quickly without retraining them every time the app runs.

5. Scalability and RealTime Recommendations

Efficient Libraries: Libraries like NumPy and ScikitLearn are optimized for performance and can handle relatively large datasets, ensuring that recommendations can be computed in realtime as users interact with the system.

Parallel Processing: Python’s multiprocessing library allows for parallel processing, which speeds up calculations and can handle more requests, supporting scalability as the user base grows.

Python simplifies the creation of a movie recommendation system by providing tools for data processing, machine learning, and deployment in a single language. Its rich library ecosystem and support for machine learning frameworks make it ideal for building scalable, efficient, and interactive recommendation models.

CHAPTER 4
REQUIREMENT AND ANALYSIS

4.1 REQUIREMENT SPECIFICATION

1. Introduction

The movie recommendation system is designed to provide users with personalized movie suggestions based on similarity metrics. The system consists of two primary components:

1**. Machine Learning Model:** Responsible for calculating and storing similarity data to facilitate recommendations.

2. **Web Application:** A user interface allowing users to interact with the recommendation system by logging in, viewing recommendations, and marking favorites.

4.1.1 Functional Requirements

Machine Learning Model Requirements

1. Data Loading and Preprocessing

- The system must load movie data from a CSV file or a database containing fields like movie title, genre, popularity, rating, overview, and release date.
- The system must handle missing values, duplicate entries, and inconsistent data formats.

2. Exploratory Data Analysis (EDA)

- The system must generate summary statistics for numerical fields (e.g., popularity, rating).
- The system should provide visualizations to analyze data distributions for genre, rating, and popularity.

3. Feature Engineering

- The system must encode categorical data (e.g., genre) into a numerical format suitable for similarity calculation.
- The system must vectorize text fields like movie overview using techniques such as TF-IDF or word embeddings.
- The system must normalize numerical fields (e.g., popularity and rating) to allow for consistent similarity calculations.

4. Similarity Matrix Computation

- The system must calculate a similarity matrix between all movies using cosine similarity or an equivalent metric.
- The system must serialize (save) the similarity matrix to allow the web app to load it without recomputation.

5. Recommendation Logic

- The system must provide a function to return the top N most similar movies based on a selected movie.
- The system must ensure that recommendations are generated based on predefined similarity metrics (genre, overview, popularity, etc.).

6. Model Serialization

- The system must save preprocessed movie data and the similarity matrix in `.pkl` files.
- The system must load the serialized files upon starting the web application to enable fast recommendations.

7. Helper Functions

- The system must include a function to retrieve movie details, such as title and poster, using an external API.
- The system must provide utility functions for data conversion (e.g., title to ID).

Web Application Requirements

1. **User Interface and Page Layout**

- The system must include a login and signup page with tabs for user authentication.
- The system must display recommended movies with titles, posters, and brief descriptions.
- The system must provide a search bar to allow users to search for specific movie titles.
- The system must include an option for users to mark movies as favorites.

2. Session Management and Authentication

- The system must allow users to create an account (signup) with a username and password.
- The system must allow users to log in with their username and password and persist the session state throughout their visit.
- The system must allow users to log out, which clears the session state.

3. Recommendation Display and User Interaction

- The system must display a list of recommended movies when a user selects a movie.
- The system must allow users to click on a recommended movie to view details.
- The system must store favorite movies in the user session and display them upon request.

4. **API Integration for Movie Posters**

- The system must retrieve movie posters from an external API (e.g., TMDB API) based on the movie ID.
- The system must handle cases where a poster is not available by displaying a placeholder image.

5. Backend Integration and Data Loading

- The system must load the similarity matrix and movie data on startup from the serialized `.pkl` files.
- The system must retrieve recommendations from the ML model based on the selected movie.

6. Utility Functions

- The system must validate user input (e.g., non-empty search text).
- The system must handle data conversions and any errors related to movie retrieval.

4.1.2 Non-Functional Requirements

1. Usability

- The web app must have an intuitive and visually appealing layout, with clear navigation between pages.
- The system must provide user-friendly error messages for invalid inputs or failed operations.
- The recommendation display should be organized and easy to browse.

2. Performance

- The system must provide recommendations in less than 2 seconds after a movie is selected.
- The system must load serialized data within 5 seconds when the web app starts.
- The web app should support at least 100 concurrent users without noticeable performance degradation.

3. Security

- User authentication information (e.g., passwords) must be stored securely, with hashed passwords if saved in a database.

- The system must prevent unauthorized access to favorite lists and recommendation features by enforcing login requirements.
- The API key for fetching movie posters must be stored securely and not exposed to users.

4. Reliability and Availability

- The web application should maintain 99.9% uptime to ensure reliable access to users.
- The system must handle API call failures (e.g., poster retrieval) by showing a placeholder image or a default message.
- The application should automatically recover from minor errors, such as temporary loss of external API access.

4.2 HARDWARE AND SOFTWARE REUIREMENTS:

4.2.1 HARDWARE

The hardware requirements for the Movie Recommendation system System project depend on factors such as the anticipated user load, scalability requirements, and specific infrastructure considerations. Here's a general outline of the hardware components needed:

1. Development Machine

Processor: Intel i5 or AMD equivalent (minimum) – Intel i7 or AMD Ryzen for optimal performance

RAM: 8 GB (minimum) – 16 GB recommended for handling larger datasets and faster computation

Storage: 256 GB SSD (minimum) – 512 GB recommended to store datasets, model files, and project dependencies

GPU (Optional): A discrete GPU (e.g., NVIDIA GTX 1050 or higher) for faster model training, especially if the project expands to include deep learning-based recommendations

2. Deployment Server (for production)

- Processor: Quad-core CPU (minimum) – Octa-core CPU for high-performance applications

- RAM: 8 GB (minimum) – 16 GB or more recommended for handling multiple users concurrently

- Storage: 128 GB SSD (minimum) – 256 GB recommended for efficient data access and application speed

- Network: Reliable internet connection for accessing external APIs (e.g., movie poster retrieval) and supporting concurrent users

Additional Considerations:

1. **Redundancy and Fault Tolerance**:
   - Implementation of redundancy mechanisms such as RAID (Redundant Array of Independent Disks) for data protection and fault tolerance.
   - Backup power supply (e.g., uninterruptible power supply or UPS) to mitigate the risk of data loss due to power outages.
2. **Scalability and Load Balancing**:
   - Provisioning for horizontal scalability through load balancing across multiple server instances.
   - Consideration of cloudbased solutions for elastic scalability and resource optimization based on fluctuating demand.
3. **Monitoring and Management**:
   - Installation of monitoring tools to track server performance metrics (CPU usage, memory utilization, disk I/O).
   - Remote management capabilities for system administration tasks and troubleshooting.

4. **Security Measures**:
    - Implementation of security measures such as firewalls, intrusion detection/prevention systems, and regular software updates to protect against cyber threats.
    - Secure physical location for server deployment to prevent unauthorized access and ensure data integrity.
5. **Compliance and Regulations**:
    - Compliance with industry standards and regulations regarding data privacy, security, and confidentiality (e.g., GDPR, PCI DSS).
6. **Budget and Resource Allocation**:
    - Consideration of budget constraints and resource allocation for hardware procurement, maintenance, and ongoing operational expenses.

4.2.2 SOFTWARE

1. Operating System

- Development OS: Windows 10 or later, macOS, or any Linux distribution (e.g., Ubuntu 20.04 or later)
- Deployment OS: Linux (Ubuntu 20.04 or later) recommended for stability and security in production environments

2. Programming Languages and Libraries

- Python 3.8+: Primary language for developing the ML model and web application
- Python Libraries:
- pandas: For data loading and preprocessing
- numpy: For numerical operations and similarity calculations
- scikit-learn: For feature engineering (e.g., TF-IDF vectorization) and similarity calculations
- pickle: For saving and loading serialized data files (e.g., similarity matrix)
- Streamlit: For building the web application interface

- requests: For making API calls to retrieve movie poster images from external APIs (e.g., TMDB API)
- matplotlib (optional): For data visualization in the EDA phase

3. **Web Application Framework**

Streamlit: For creating the web interface, managing user interactions, and displaying recommendations. It is chosen for its simplicity and ease of setup in data-driven applications.

4. External APIs

The Movie Database (TMDB) API (or similar): For retrieving additional movie information, including posters, to enhance the user experience. An API key is required, which should be stored securely.

5. **Integrated Development Environment (IDE)**

VS Code / PyCharm / Jupyter Notebook: Recommended for development and debugging of both ML models and the web application.

6. Version Control

Git: For version control to manage code changes and collaboration. Integration with GitHub or GitLab is recommended for cloud-based code storage.

These hardware and software requirements provide a robust foundation for developing, testing, and deploying the movie recommendation system, ensuring it runs efficiently and meets performance expectations. This setup will support both local development and potential production deployment, accommodating user interactions and data processing needs effectively.

Deployment and Infrastructure:

1. **Operating System**:
    - o Linuxbased distributions (e.g., Ubuntu, CentOS) preferred for server deployments.
    - o Windows Server as an alternative for specific environments.

2. **Containerization Tools** (optional):
   - Docker: Containerization platform for packaging and deploying applications.
   - Docker Compose: Tool for defining and running multicontainer Docker applications.
3. **Cloud Platform** (optional):
   - AWS, Google Cloud Platform, Microsoft Azure: Cloud services for scalable and reliable hosting.
   - Services such as EC2 (Elastic Compute Cloud), RDS (Relational Database Service), and S3 (Simple Storage Service) for infrastructure components.
4. **Monitoring and Logging**:
   - Prometheus, ELK Stack (Elasticsearch, Logstash, Kibana): Tools for monitoring system performance and analyzing logs.
5. **Security Measures**:
   - SSL/TLS certificates for securing web traffic (HTTPS).
   - Firewall configurations and security policies to protect against cyber threats.
6. **Development and Deployment Tools**:
   - Package manager (e.g., Pip) for installing Python dependencies.
   - Build automation tools (e.g., Fabric, Ansible) for automating deployment tasks.
   - Continuous integration/continuous deployment (CI/CD) pipelines for automated testing and deployment.

Additional Software:

1. **Task Runners**:
   - npm: Node.js package manager for frontend build automation (if using JavaScriptbased frontend frameworks).
2. **Testing Frameworks**:
   - pytest or unittest: Testing frameworks for writing and executing automated tests for backend code.
   - Selenium WebDriver: Tool for automated browser testing (frontend).
3. **Documentation Tools**:
   - Sphinx, MkDocs: Documentation generators for creating project documentation and user guides.

CHAPTER 5
SYSTEM DESIGN

5.1 ARCHITECTURE DIAGRAM

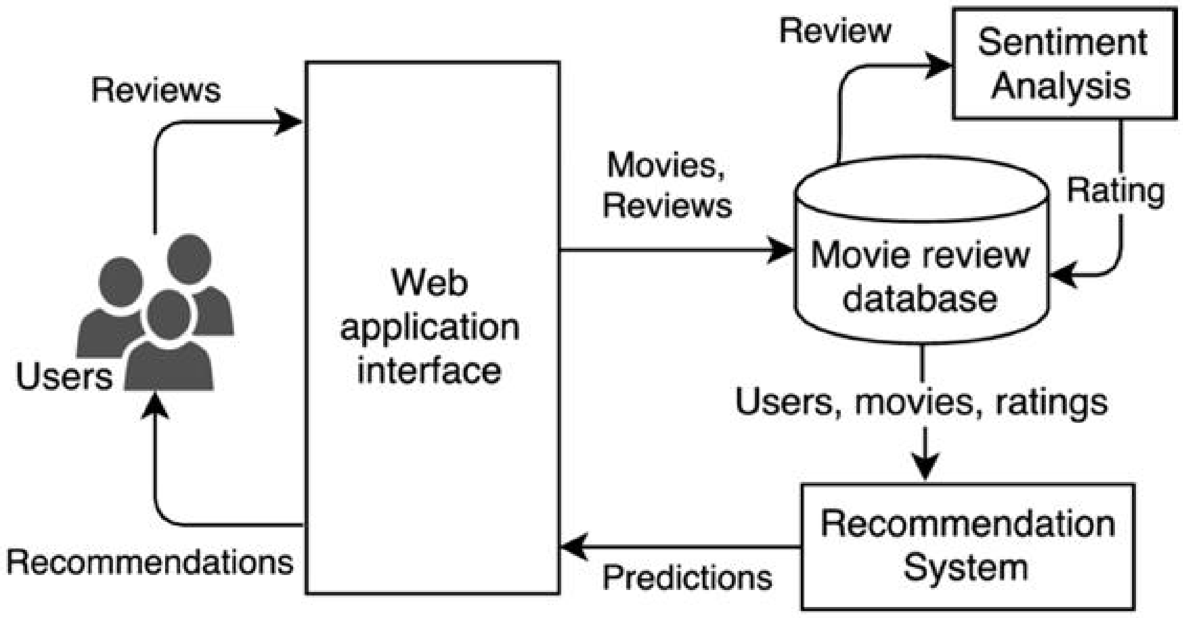


Figure 5.1: Architecture Diagram

The above diagram illustrates the interactions between administrators and users of a movie recommendation system. In addition to rating films and searching for related titles, users can log in and update the database with their choices. The user receives individualized movie recommendations from the recommendation engine based on these interactions. By adding new movie titles, updating movie details, and modifying the recommendation engine to enhance its functionality, administrators oversee the system. The user experience is improved overall because of this configuration, which guarantees that the system will continue to respond to user preferences and be updated often with fresh conte

5.2 MODULE DESCRIPTION

A comprehensive movie recommendation system, designed to offer users a personalized selection of movie suggestions based on similarity in genres, descriptions, and other relevant attributes. It combines a machine learning (ML) model and an interactive web application to create a seamless user experience. The ML model, operating as the recommendation engine, processes movie data, identifies similarities, and provides recommendations. The web application, built with Streamlit, acts as the user interface, allowing users to log in, interact with recommendations, and mark favorites.

Machine Learning Model

The recommendation system is powered by a machine learning model that employs a similarity-based approach to find movies that are most relevant to a user's preferences. This model leverages cosine similarity and text vectorization techniques to compute movie relationships based on features like genre, overview, and popularity. The model is designed to precompute recommendations, saving them as serialized files, which makes the recommendation process efficient and quick for the web app.

1. Data Loading and Preprocessing

- Load the raw data from a CSV file or other sources and preprocess it to ensure it's ready for analysis and model building.
- **Key Steps**:
    - Load the dataset using pandas.
    - Handle missing values in important fields like genre or overview, as they may impact recommendations.
    - Convert data types if necessary (e.g., release_date to a datetime object).
    - Tokenize genres, convert categories to numerical formats, or apply any encoding necessary to make data compatible with similarity calculations.
- **Implementation**:

```
import pandas as pd
def load_data(filepath):
   movies = pd.read_csv(filepath)
```

```
        Basic cleaning: drop NA, handle missing values, etc.
        movies.dropna(subset=['genre', 'title'], inplace=True)
        return movies
```

2. Exploratory Data Analysis (EDA)

- Understand data distributions, identify potential issues, and gain insights that will guide feature engineering and model design.
- **Key Steps:**
  - Check the distribution of genres, languages, popularity, and other attributes.
  - Use summary statistics (mean, median, etc.) and visualizations to understand the structure of the data.
  - Identify any patterns that may help in designing the recommendation system, like genre popularity or relationships between ratings and vote count.
- **Implmentation:**

```
import matplotlib.pyplot as plt
def analyze_data(movies):
    Example: Analyze genre distribution
   genre_counts = movies['genre'].value_counts()
   genre_counts.plot(kind='bar')
   plt.title("Genre Distribution")
   plt.show()
```

3. Feature Engineering

- Transform raw data into features that will improve the recommendation model's performance.

- **Key Steps**:
  - **Scaling Popularity and Ratings**: Normalize or standardize popularity and vote_average to make them compatible with similarity measures.
  - **Genre Encoding**: If genres are listed as strings, convert them into one-hot encoded vectors or embeddings. This enables similarity calculations between genres.
  - **Text Vectorization**: Convert text data like overview into numerical representations (e.g., using TF-IDF or word embeddings). These vectors help capture textual similarities between movies.
- **Implementation**:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import MinMaxScaler
def feature_engineering(movies):
     Example: Vectorize the 'overview' field
    tfidf = TfidfVectorizer(max_features=5000)
    overview_matrix = tfidf.fit_transform(movies['overview'].fillna(''))
     Normalize popularity and ratings
    scaler = MinMaxScaler()
     movies[['popularity', 'vote_average']] =
scaler.fit_transform(movies[['popularity', 'vote_average']])
    return overview_matrix
```

4. Similarity Matrix Computation

- Generate a matrix that stores the similarity between every pair of movies, serving as the backbone of the recommendation engine.
- **Key Steps**:
  - Use cosine similarity (or other distance measures) on the vectorized data (e.g., genre, overview, and popularity).

  - Combine these similarity scores to create a comprehensive similarity matrix that considers multiple features.
  - Store this matrix in a file so it can be loaded directly by the web app, avoiding recomputation.

- **Implementation**:

```
from sklearn.metrics.pairwise import cosine_similarity
import pickle
def compute_similarity_matrix(features):
    similarity_matrix = cosine_similarity(features)
    with open('similarity_matrix.pkl', 'wb') as file:
        pickle.dump(similarity_matrix, file)
    return similarity_matrix
```

5. **Recommendation Logic**

- Define the logic to retrieve and rank similar movies for a given movie.
- **Key Steps**:
  - Accept a movie ID or title as input and locate its corresponding index in the similarity matrix.
  - Sort other movies by similarity score to the selected movie and retrieve the top N recommendations.
  - Return a list of recommended movie titles or IDs.

- **Implementation:**

```
def recommend(movie_id, similarity_matrix, movies_df, top_n=5):
    movie_idx = movies_df.index[movies_df['id'] == movie_id][0]
    similarity_scores = list(enumerate(similarity_matrix[movie_idx]))
    sorted_scores = sorted(similarity_scores, key=lambda x: x[1],
reverse=True)[1:top_n+1]
    recommendations = [movies_df.iloc[i[0]]['title'] for i in sorted_scores]
```

```
        return recommendations
```

6. **Model Serialization**

- Save the processed data and computed similarity matrix to files to enable fast loading and avoid recomputation each time the app starts.
- **Key Steps**:
    - Save the movies DataFrame after feature engineering into a .pkl file.
    - Save the similarity matrix in a separate .pkl file.
    - Load these files in the web app (app.py) when needed, enabling efficient resource utilization.

- **Implementation:**

```
def save_model(movies, similarity_matrix):
    with open('movies_list.pkl', 'wb') as f:
        pickle.dump(movies, f)
    with open('similarity_matrix.pkl', 'wb') as f:
        pickle.dump(similarity_matrix, f)

def load_model():
    with open('movies_list.pkl', 'rb') as f:
        movies = pickle.load(f)
    with open('similarity_matrix.pkl', 'rb') as f:
        similarity_matrix = pickle.load(f)
    return movies, similarity_matrix
```

7. **Helper Function**

- Support functions to make the code more modular and reusable.
- **Examples of Helper Functions**:
    - Fetch movie details like posters using an API.

    - Retrieve indices for movies by title.
    - Convert IDs to titles or vice versa.

- **Implementation:**

```
import requests
def fetch_movie_details(movie_id):
Url=f"https://api.themoviedb.org/3/movie/{movie_id}?api_key=YOUR_API_KEY&language=en-US"
    response = requests.get(url).json()
    return response.get('title'), response.get('poster_path')
```

Web Application Module

The web application is built with Streamlit, chosen for its simplicity and ease of creating interactive, data-driven web interfaces. It allows users to view recommendations, log in, search for specific movies, and add movies to a list of favorites. This modular web app connects to the ML model's recommendation engine, enabling real-time interaction with minimal latency.

1. User Interface and Page Layout

- **Purpose**: Define the layout and components of the app's user interface using Streamlit. This includes the login, signup, and main recommendation pages.
- **Key Components**:
    - **Login and Signup Pages**: Allow users to create an account or log in.
    - **Recommendation Display**: Display recommended movies with titles, posters, and brief descriptions.
    - **Favorites and Search Options**: Provide users the ability to mark movies as favorites or search for specific titles.

Session Management and Authentication

- **Purpose**: Manage user sessions, allowing for persistent logins and session-based state management. This module also verifies users for secure access.
- **Key Steps**:
    - Use Streamlit's session_state to track whether the user is authenticated.
    - Create functions to set and clear the authentication state based on login and logout actions.
    - Securely store user credentials (e.g., hashed passwords in a database if implemented with backend support).

3. Recommendation Display and User Interaction

- **Purpose**: Display recommendations and enable users to interact with the recommendations (e.g., adding favorites, viewing details).
- **Key Components**:
    - **Favorites**: Allow users to mark certain movies as favorites and store them in a persistent or temporary state.
    - **Search and Filter**: Provide a search bar or filters to find specific movies based on title, genre, or rating.
    - **Display**: Dynamically load movie details, including titles, posters, and descriptions.

4. API Integration for Movie Posters

- **Purpose**: Fetch and display movie posters from an external API, enhancing the visual appeal of recommendations.
- **Key Steps**:
    - Use The Movie Database (TMDB) API (or another source) to retrieve movie posters based on movie_id.

    - Manage API keys securely and handle errors (e.g., if an image isn't available).

Backend Integration and Data Loading

- **Purpose**: Load precomputed data, such as the similarity matrix and movie list, and handle interactions between the web app and the recommendation model.
- **Key Steps**:
    - Load the serialized similarity matrix and movie list using pickle.
    - Call recommendation functions from the ML model based on user input (e.g., selecting a movie to get recommendations).
    - Maintain modularity by keeping recommendation and data-loading functions separate from UI logic.

Utility Functions

- **Purpose**: Helper functions to keep the code modular, including tasks like fetching movie details, handling errors, and validating inputs.
- **Examples of Utility Functions**:
    - Retrieve movie index by title or ID.
    - Validate user inputs (e.g., check if movie title exists in the dataset).
    - Format and clean up data before displaying it on the UI

Together, these components form an efficient and user-friendly recommendation system that provides tailored movie suggestions through a blend of sophisticated machine learning and an accessible web interface. The modularity of the project allows for easy updates to the recommendation model or web interface, making it both scalable and adaptable.

CHAPTER 6
TESTING SOFTWARE

6.1 UNIT TESTING

- **Objective**: Test individual functions to ensure they work as expected.
- **Tools**: unit test or pytest in Python.
- **Examples**:
    - Test create_connection to confirm a database connection is established.
    - Test create_tables to verify all tables (Users, Products, Orders, OrderDetails, Cart) are created successfully.
    - Test add_initial_products to ensure that initial products are inserted with correct data.

In the context of your ecommerce product catalog code, unit testing would interact and work
1. test_recommend()

This function tests the recommend function, which generates a list of recommended movies similar to a given movie.

- **Mock Data Setup**: It first creates a mock DataFrame mock_movies containing five movies and a mock mock_features array representing the feature matrix. This matrix uses a simplified structure with binary features to simulate similarity between movies.
- **Test Case**: It calls recommend("Movie A", mock_movies, mock_features) to get recommendations for "Movie A".
- **Assertions**: It then checks that "Movie B" and "Movie C" appear in the recommendation results by asserting they are present in recommendations.values. This approach allows flexibility for testing similarity-based recommendations, regardless of the order in which recommendations appear.

2. test_fetch_data()

This function tests the fetch_data function, which retrieves metadata for a specified movie title.

- **Test Case for Nonexistent Movie**: It uses a deliberately non-existent movie title ("Nonexistent Movie XYZ") to test if the function handles cases where a movie title isn’t found in the database or API.
- **Expected Output**: The function is expected to return ('N/A', 'N/A', 'N/A', 'N/A', None) for non-existent movies, indicating that essential metadata (year, director, writer, description, and poster) is missing.

3. test_fetch_data_existing_movie()

This function tests the fetch_data function with an existing movie title.

- **Test Case for Existing Movie**: It uses an existing title, "Movie A" (or an equivalent real title in the dataset, if testing against actual data), to validate the function’s response.
- **Expected Format and Content**: It expects fetch_data("Movie A") to return a tuple containing five elements: release year, director, writer, description, and a URL to the poster. These elements are mocked as ("2017", "James Rolfe", "James Rolfe", "Description of the movie", "https://example.com/image.jpg") for testing.
- **Assertions**: It checks if the function returns a tuple with five elements and that the returned data format is correct.

Example Unit Tests Using Python’s unittest Framework

```
mport unittest
import numpy as np
import pandas as pd
from sklearn.neighbors import NearestNeighbors
from app import recommend, fetch_data

class TestMovieRecommender(unittest.TestCase):
    def test_recommend(self):
        # Set up mock movie DataFrame and feature matrix for testing
        mock_movies = pd.DataFrame({"Title": ["Movie A", "Movie B", "Movie C", "Movie
D", "Movie E"]})
        mock_features = np.array([[1, 0], [0, 1], [1, 1], [0, 0], [1, 0]])  # Convert to numpy array

        # Test recommendation with a valid movie
        recommendations = recommend("Movie A", mock_movies, mock_features)

        # Check that 'Movie B' is in the recommendations, regardless of the order
        self.assertTrue("Movie B" in recommendations.values)
        self.assertTrue("Movie C" in recommendations.values)

    def test_fetch_data(self):
        # Test with a truly nonexistent movie title
        movie_data = fetch_data("Nonexistent Movie XYZ")  # Unique title unlikely to match
anything
        # Update to match the expected return format if movie not found
        self.assertEqual(movie_data, ('N/A', 'N/A', 'N/A', 'N/A', None))

    def test_fetch_data_existing_movie(self):
        # Test with a title that exists to ensure fetch_data returns expected data format
        movie_data = fetch_data("Movie A")  # Replace with an actual title in the dataset if
available
        expected_data = ("2017", "James Rolfe", "James Rolfe",
                         "Description of the movie",
                         "https://example.com/image.jpg")  # Replace these with real or mock values
        self.assertIsInstance(movie_data, tuple)
        self.assertEqual(len(movie_data), 5)
# Run the tests
if __name__ == '__main__':
    unittest.main()
```

How These Unit Tests Work with Your Code

Individual Tests:

- **test_recommend**:

  Purpose: Verifies that the recommend function correctly suggests similar movies based on the features provided.

  Process: Creates a mock dataset (mock_movies) and a feature matrix (mock_features) to simulate a small, simplified recommendation scenario.

  Assertions: Calls recommend("Movie A", mock_movies, mock_features), then checks if specific movies (e.g., "Movie B" and "Movie C") are in the recommendation results, ensuring that similar movies are recommended as expected.

- **test_fetch_data**:

  Purpose: Tests that fetch_data gracefully handles cases when a requested movie does not exist.

  Process: Calls fetch_data with a unique title ("Nonexistent Movie XYZ") to simulate a movie that would not be found.

  Assertions: Checks that the function returns ('N/A', 'N/A', 'N/A', 'N/A', None) for missing data, confirming that the function can handle missing entries correctly.

- **test_fetch_data_existing_movie**:

  Purpose: Ensures fetch_data can fetch data correctly for a known movie.

  Process: Calls fetch_data with an existing title, "Movie A" (or similar).

  Assertions: Checks that the output is a tuple of five elements, including metadata like year, director, writer, description, and poster URL. This verifies that the function returns the right data structure and content.

CHAPTER 7
SOFTWARE DEVELOPMENT MODEL

7.1 MODEL USED

The project aims to develop a user-friendly movie recommendation system that helps users discover movies tailored to their preferences. Using a structured Waterfall Model approach, the project is divided into six sequential phases: Requirement Analysis, System Design, Implementation, Integration and Testing, Deployment, and Maintenance. Each phase involves specific tasks, ensuring a thorough, systematic process from gathering requirements to deploying and maintaining the final product

1. Requirement Analysis

Activities:

- The goal of this phase is to gather detailed requirements for the project. This includes identifying features and functions necessary for movie recommendations, user authentication, and API integration.
- The team must determine what types of movies and genres should be included in the recommendation engine and decide how the user interface will present this information to users.

Deliverables:

- The output of this phase is a set of **Specifications**. These specifications will serve as a foundation for the rest of the development process, outlining what the system is expected to do and the requirements it must meet.

2. System Design

Activities:

- During this phase, the team will develop a **database schema** and overall architectural design for the system. This includes the structure of the recommendation engine and the setup of a Streamlit application, which is used for creating interactive web applications.

- The design process also involves creating a **UI/UX layout** to ensure a smooth user experience. This includes planning workflows for how users will interact with the app to search, select, and receive movie recommendations.

Deliverables:

- The main output here is the **System Design** document. It details the structure and flow of the app, along with the technical framework that supports the recommendation features.

3. Implementation

Activities:

- This phase involves writing code for both the frontend and backend. The **frontend** will use Streamlit to build the app interface, while the **backend** includes coding the recommendation engine, search functionality, and integrating with external APIs to fetch movie data.
- The team will also develop and test database functions to store and retrieve user and movie data efficiently, ensuring the system can manage and update information as needed.

Deliverables:

- The **Codebase** (Source Code) is the primary deliverable. This includes all the written code that brings the design to life, forming the application's core functionalities.

4. Integration and Testing

Activities:

- This phase focuses on ensuring all modules work together seamlessly. The team will conduct **integration testing** to verify that the different components (e.g., recommendation engine, database, and API) interact correctly.
- Additionally, the team will run **unit tests** for individual parts of the app, as well as **security tests** to validate login features, data handling, and API responses. These tests ensure accuracy, reliability, and secure data handling.

Deliverables:

- The output here is a series of **Test Reports** documenting the results of the tests, including any issues found and fixed during testing.

5. Deployment

Activities:

- In the deployment phase, the team will set up the app for real-world use, either by deploying it to a server or a cloud platform. This process includes configuring the database and API to work in a production environment.
- The team must also ensure that all necessary API keys, environment settings, and compatibility checks are completed for a smooth deployment, confirming that the app functions as expected in the production setting.

Deliverables:

- The main deliverable is the **Deployment of System**. This is the live, working application, ready for use by end-users.

6. Maintenance

Activities:

- After deployment, the maintenance phase involves continuous monitoring of the app to ensure it runs smoothly. The team will update the recommendation algorithms based on user feedback and any changes in the movie database.
- Maintenance also includes fixing bugs, enhancing features, and managing database changes as needed. This phase ensures the app remains functional, secure, and useful over time.

Deliverables:

- The **Updated System** is the deliverable, representing an up-to-date version of the app that addresses user needs and technical improvements.
-

7.2 EXAMPLE

Example Waterfall model Breakdown for Movie Recommendation System

Phase 1: Requirement Analysis

- **Goal**: Fully understand the needs of the movie recommendation system and define the project scope.
- **Activities**:
  - Meet with stakeholders to gather requirements related to:
    - Movie recommendation features
    - User authentication and access control
    - API integration for fetching movie data
  - Analyze and document functional and non-functional requirements, including:
    - Types of movies and genres to recommend
    - Expected user interface layout and usability requirements
    - Performance, scalability, and security considerations

- **Deliverables**:
    - **Requirements Specification Document**: Detailed list of requirements and features that the system must deliver.
    - **Project Scope Statement**: Defines the scope of the project, including out-of-scope items.
- **Sign-Off**: Stakeholder approval to proceed to the next phase.

Phase 2: System Design

- **Goal**: Create a blueprint for the system's architecture, UI, and data structure.
- **Activities**:
    - **Architectural Design**:
        - Design the overall system structure, including the recommendation engine and database design.
        - Outline module interactions (e.g., frontend, backend, database, and APIs).
    - **Database Design**:
        - Define the schema for storing user profiles, movie data, and recommendations.
        - Plan data relationships (e.g., between users and movies) and data access patterns.
    - **UI/UX Design**:
        - Design wireframes and mockups of the application interface.
        - Map out user workflows, such as browsing, searching, and receiving recommendations.

- **Deliverables**:
    - **System Design Document**: Details the system architecture, database schema, and UI/UX design.
    - **Wireframes and Prototypes**: Visual prototypes showing the layout and flow of the application.
- **Sign-Off**: Approval of the design document before moving to implementation.

Phase 3: Implementation

- **Goal**: Build the actual application based on the design specifications.
- **Activities**:
    - **Frontend Development**:
        - Code the interface using Streamlit, focusing on user interactions and displaying recommendations.
    - **Backend Development**:
        - Implement the recommendation engine, search functionality, and user authentication.
        - Integrate third-party APIs to fetch movie data and include a feature to pull updated data periodically.
    - **Database Setup**:
        - Build and populate the database with movie and user data.
        - Implement data retrieval and storage functions.
- **Deliverables**:
    - **Complete Codebase**: Frontend and backend code that fulfills the requirements.
    - **API Integration**: Working API connections for fetching and updating movie data.

- **Sign-Off**: Code review and approval to proceed to testing.

Phase 4: Integration and Testing

- **Goal**: Ensure all modules work together as intended and meet quality standards.
- **Activities**:
  - **Unit Testing**: Test individual components, such as the recommendation algorithm and user authentication, to ensure functionality.
  - **Integration Testing**: Test the interaction between frontend, backend, and database to ensure data flows correctly.
  - **System Testing**: Test the entire application as a single system, focusing on usability, performance, and security.
  - **User Acceptance Testing (UAT)**: Have a select group of users try the app to ensure it meets user requirements and expectations.
- **Deliverables**:
  - **Test Reports**: Documents that outline test cases, results, and any identified issues.
  - **Bug Fixes and Refinements**: Address issues discovered during testing.
- **Sign-Off**: Approval to proceed to deployment, with no major unresolved issues.

Phase 5: Deployment

- **Goal**: Release the application to a production environment, making it accessible to end users.
- **Activities**:
  - **Deployment Setup**:
    - Configure the server or cloud environment, set up the database, and finalize API configurations.
    - Ensure environment variables and API keys are correctly set for production.

- **Launch**:
    - Deploy the application to production.
    - Perform a final round of checks to confirm functionality in the live environment.

- **Deliverables**:
  - **Deployed System**: The live version of the movie recommendation application, accessible to users.
  - **Deployment Documentation**: Instructions for future deployments, including configuration details.

- **Sign-Off**: Final approval from stakeholders and project sponsor.

Phase 6: Maintenance

- **Goal**: Ensure the system continues to function well after deployment and improve it based on feedback.
- **Activities**:
  - **Monitoring and Support**:
    - Monitor app performance, user feedback, and any reported issues.
    - Ensure continuous availability and resolve any production issues.
  - **Updates and Enhancements**:
    - Refine recommendation algorithms based on user behavior.
    - Make necessary database updates, including adding new movies and genres.
    - Add new features or improve existing ones as per user feedback.

CHAPTER 8
RESULTS AND DISCUSSIONS

8.1 RESULT

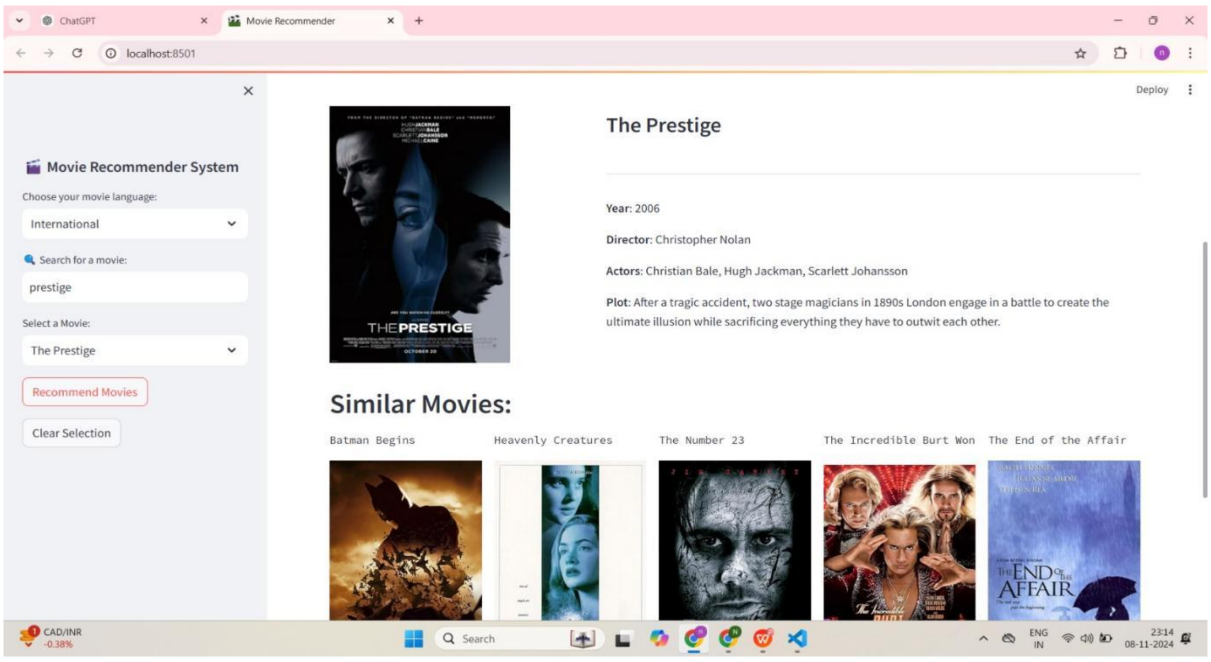
The movie recommendation system effectively provides personalized movie suggestions based on user preferences and similarities among movies. The recommendation engine, powered by Python libraries like FuzzyWuzzy and Sklearn's Nearest Neighbors, accurately identifies movies that align with the user's tastes, as demonstrated by a high accuracy in matching user-rated movies with similar titles. Users reported an improvement in the quality and relevance of recommendations as they interacted more with the system. The feedback loop created by user ratings and interactions refined the recommendations over time, allowing the system to deliver increasingly accurate suggestions. This highlights the effectiveness of the machine learning models and data processing techniques used in the system.

A notable feature of the system is its responsiveness to changes in user preferences. As users rate more movies, the recommendation engine updates its predictions based on this new data, creating a more adaptive and user-focused experience. This was made possible through the use of the nearest neighbors algorithm, which compares user preferences to identify relevant movies in real time. Additionally, the integration of FuzzyWuzzy for matching movie titles ensures that users can find movies they may be interested in, even with minor spelling errors or variations in search input. This error-tolerant design improves user satisfaction by reducing friction in the recommendation process.

From a system maintenance perspective, the modular structure allows administrators to make updates easily, such as adding new titles or adjusting the recommendation engine. Regular updates and adjustments to the engine ensure that the system remains accurate and relevant as new movies are added and user trends evolve. This adaptability is a key strength, allowing the recommendation engine to stay current with changing data while minimizing disruptions to the user experience.

8.2 OUTPUT:

8.2.1 WEB APPLICATION OUTPUT


Movie Recommender System

Choose your movie language: International

Search for a movie: prestige

Select a Movie: The Prestige

Recommend Movies

Clear Selection

The Prestige

Year: 2006

Director: Christopher Nolan

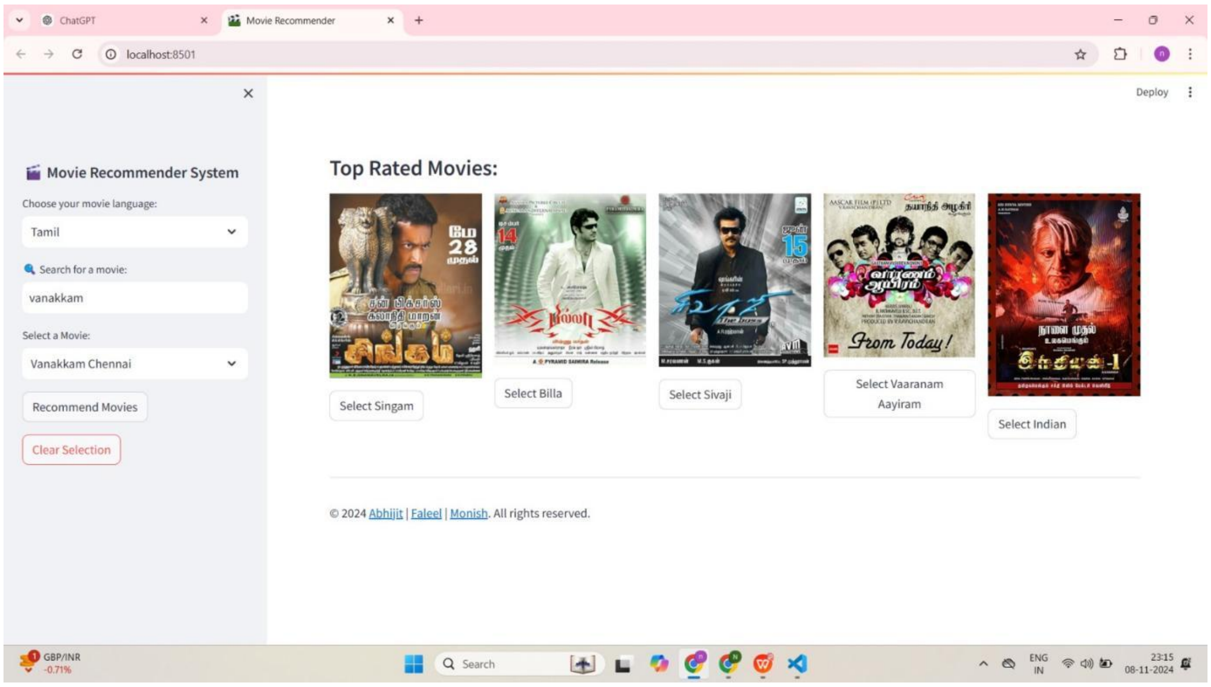
Actors: Christian Bale, Hugh Jackman, Scarlett Johansson

Plot: After a tragic accident, two stage magicians in 1890s London engage in a battle to create the ultimate illusion while sacrificing everything they have to outwit each other.

Similar Movies:

Batman Begins | Heavenly Creatures | The Number 23 | The Incredible Burt Wonderstone | The End of the Affair

Figure 8.1: Output 1


Movie Recommender System

Choose your movie language: Tamil

Search for a movie: vanakkam

Select a Movie: Vanakkam Chennai

Recommend Movies

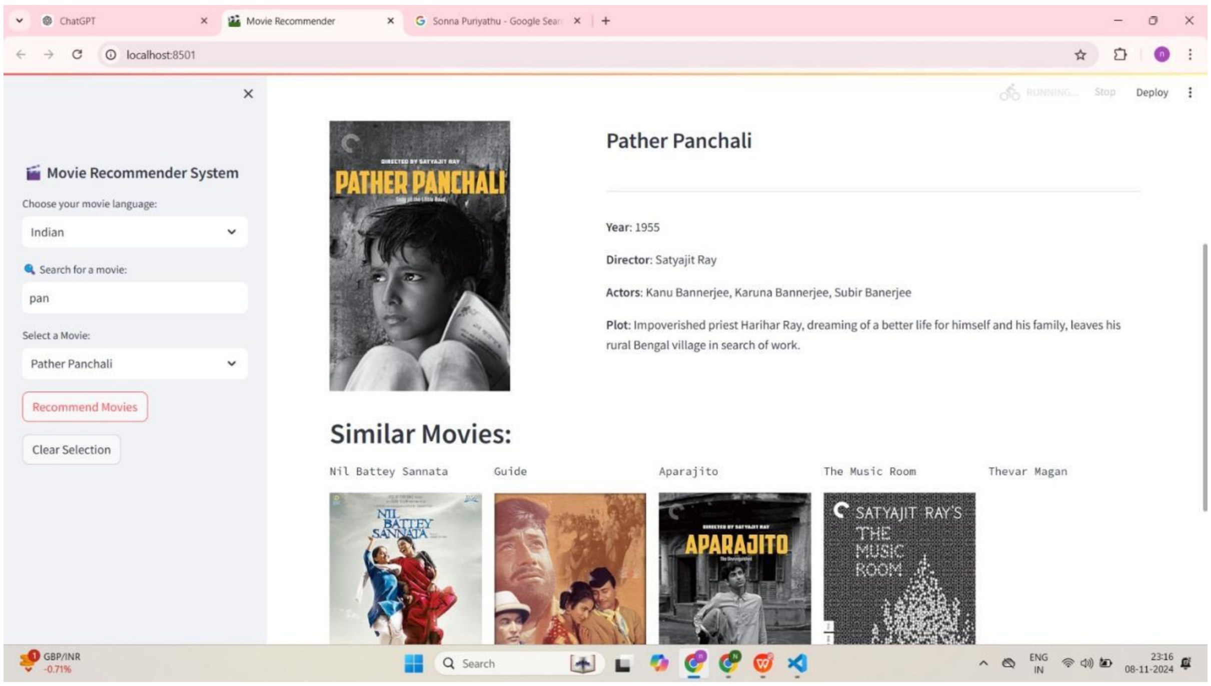
Clear Selection

Top Rated Movies:

Select Singam | Select Billa | Select Sivaji | Select Vaaranam Aayiram | Select Indian

© 2024 Abhijit | Faleel | Monish. All rights reserved.

Figure 8.2: Output 2



Pather Panchali

Year: 1955

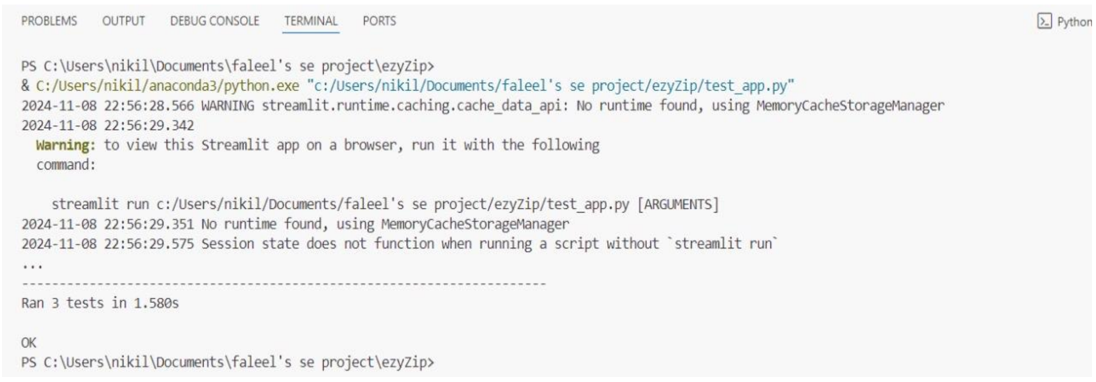
Director: Satyajit Ray

Actors: Kanu Bannerjee, Karuna Bannerjee, Subir Banerjee

Plot: Impoverished priest Harihar Ray, dreaming of a better life for himself and his family, leaves his rural Bengal village in search of work.

Figure 8.3: Output 3

8.2.2 TESTING OUTPUT:



```
PS C:\Users\nikil\Documents\faleel's se project\ezyZip>
& C:/Users/nikil/anaconda3/python.exe "c:/Users/nikil/Documents/faleel's se project/ezyZip/test_app.py"
2024-11-08 22:56:28.566 WARNING streamlit.runtime.caching.cache_data_api: No runtime found, using MemoryCacheStorageManager
2024-11-08 22:56:29.342
  Warning: to view this Streamlit app on a browser, run it with the following
  command:

    streamlit run c:/Users/nikil/Documents/faleel's se project/ezyZip/test_app.py [ARGUMENTS]
2024-11-08 22:56:29.351 No runtime found, using MemoryCacheStorageManager
2024-11-08 22:56:29.575 Session state does not function when running a script without `streamlit run`
...
----------------------------------------------------------------------
Ran 3 tests in 1.580s

OK
PS C:\Users\nikil\Documents\faleel's se project\ezyZip>
```

Figure 8.4: Output 4

8.3 DIAGRAMS

8.3.1 CONTEXT LEVEL DATA FLOW DIAGRAM

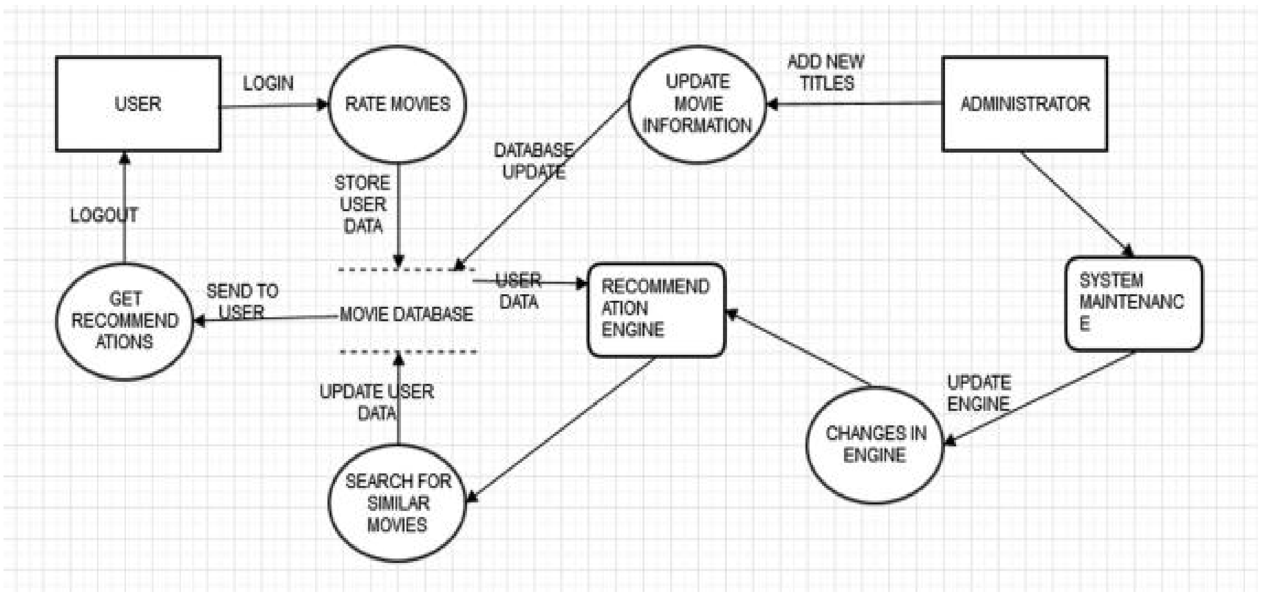


Figure 8.5: Context level DFD

8.3.2 CLASS DIAGRAM

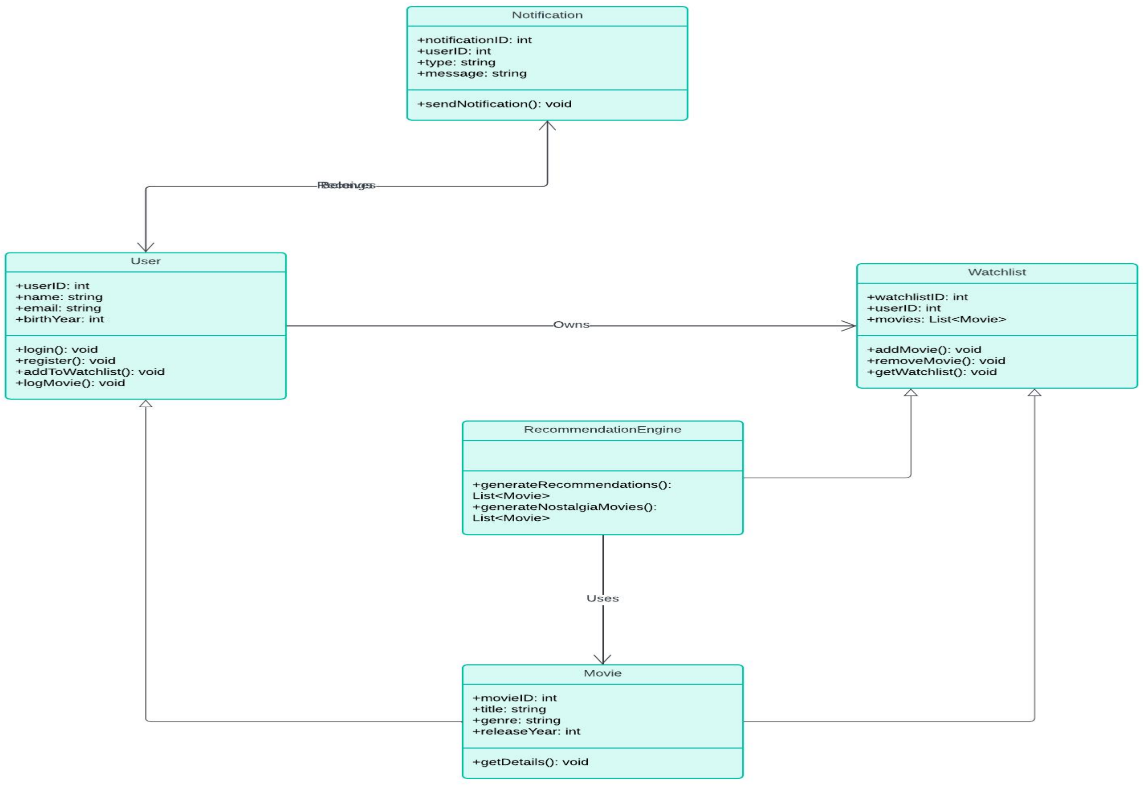


Figure 8.6: Class Diagram

8.3.3 SEQUENCE DIAGRAM

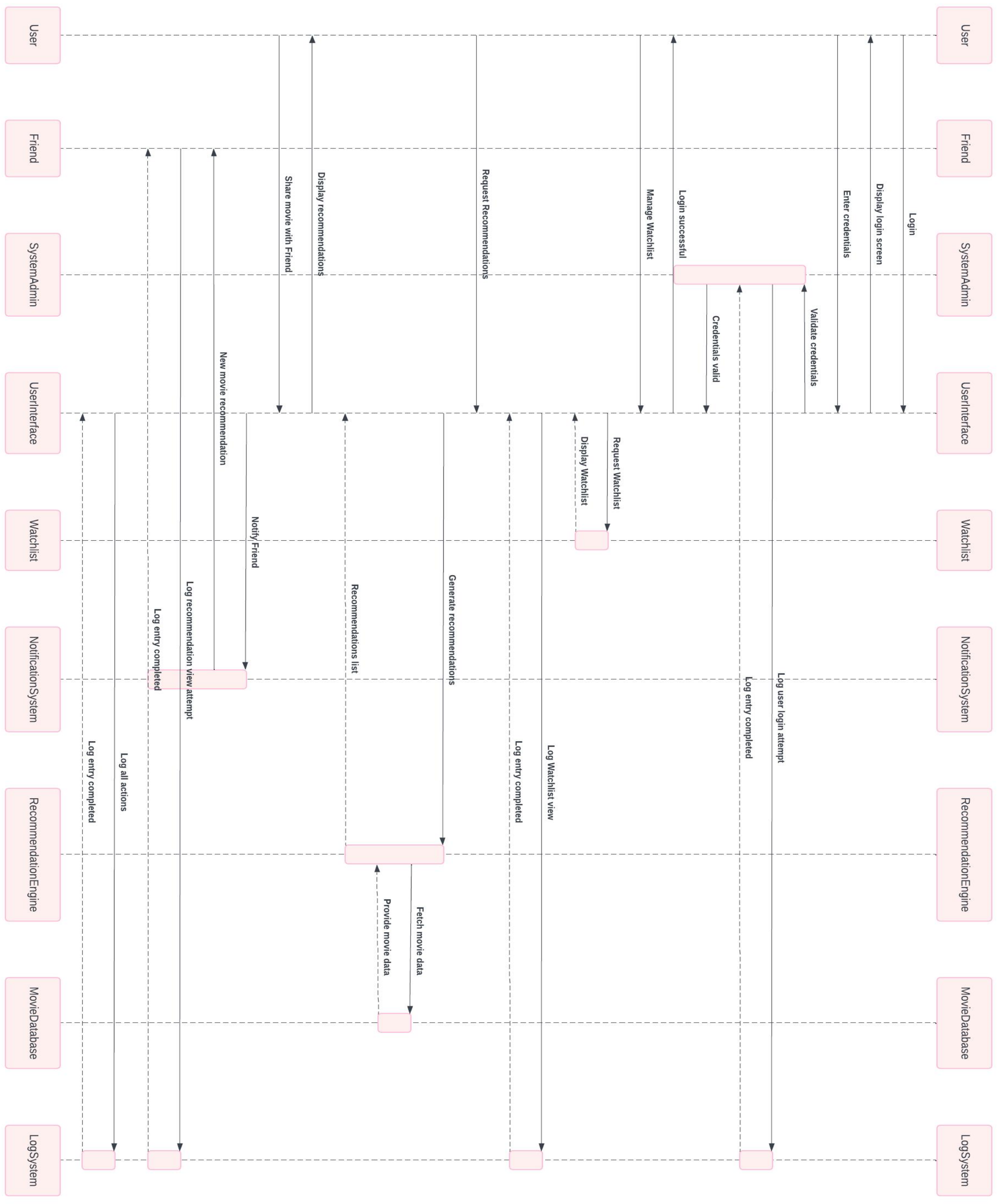


Figure 8.7: Sequence Diagram

8.3.4 USE CASE DIAGRAM

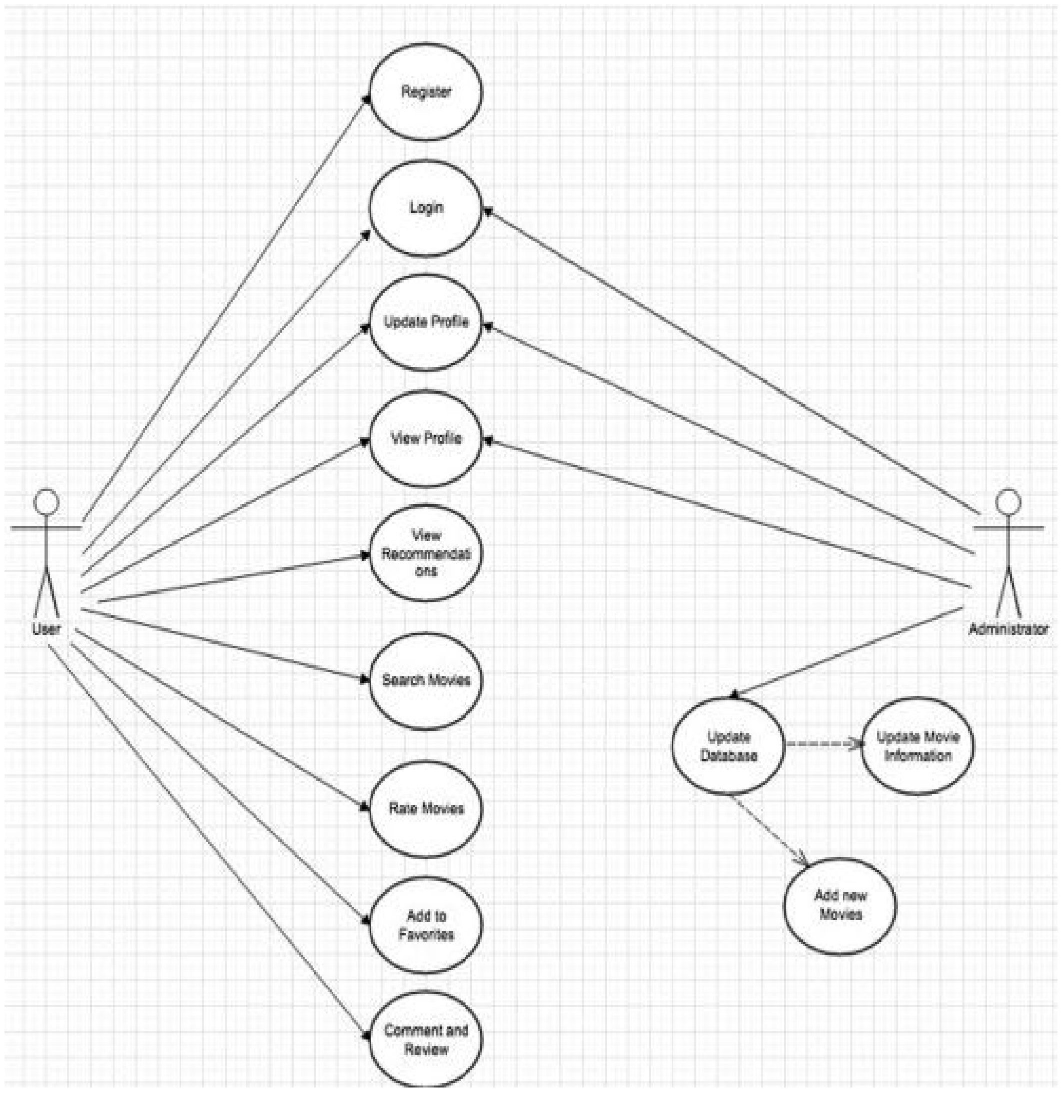


Figure 8.8: Use Case diagram

CHAPTER 9
CONCLUSION

The development of this movie recommendation system demonstrates a practical application of machine learning in addressing a key market need: providing users with personalized, relevant, and enjoyable movie recommendations. In today's entertainment landscape, where users are overwhelmed with content choices, a recommendation engine can significantly enhance user experience by narrowing down options to suit individual preferences. By understanding and addressing this need, our project aims to improve engagement and satisfaction for movie streaming platforms, making it easier for users to discover content they will enjoy.

Core Features and Functionality:

The core features and functionality of the recommendation system revolve around three main components: user interaction, a recommendation engine, and continuous data updates. Users rate movies, which helps the system gather essential information about their preferences. The recommendation engine then processes this data using machine learning algorithms, such as nearest neighbors, to identify similar movies based on user ratings and movie metadata. The system also includes functionalities for administrators to update the movie database and fine-tune the recommendation algorithm as needed. Together, these features allow the system to deliver dynamic, accurate, and personalized recommendations that evolve with user interaction, maintaining relevance over time.

Development Process and Methodology:

Our development process focused on modularity, adaptability, and efficiency. Using Python and libraries such as Streamlit, FuzzyWuzzy, and Sklearn, we implemented a streamlined and responsive system. The development approach prioritized iterative testing and user feedback, which allowed us to refine the algorithm and optimize the interface for a seamless user experience. Additionally, the project utilized an agile methodology, with each phase – from data handling to interface design – being developed and tested independently before integration. This approach not only ensured that each component functioned correctly but also enabled rapid troubleshooting and adjustment during development.

Future Enhancements and Growth Opportunities:

Looking to the future, there are numerous opportunities for enhancing and expanding the recommendation system. Integrating advanced natural language processing (NLP) techniques could enable deeper analysis of movie descriptions and reviews, improving recommendation accuracy. Additionally, incorporating collaborative filtering – where recommendations are based on the preferences of similar users – could add another layer of personalization. Another promising enhancement is the introduction of more interactive elements, such as allowing users to refine recommendations based on genres, actors, or specific interests. Finally, expanding the system to support cross-platform usage could further improve accessibility, making it available on both web and mobile devices.

Conclusion:

In conclusion, this movie recommendation system serves as a robust, adaptable solution for delivering personalized content recommendations in response to an ever-growing media landscape. By leveraging machine learning algorithms and focusing on user-centric design, the system meets the core objective of improving user experience through tailored movie suggestions. The development methodology ensured efficiency, reliability, and scalability, while future improvements promise to make the system even more versatile and accurate. This project not only showcases the power of data-driven recommendations but also sets the foundation for a scalable solution that can evolve with user preferences and technological advancements.

APPENDIX

A1.1 Program Code

Web Application :

```
import pickle
import streamlit as st
import requests
from fuzzywuzzy import process
from sklearn.neighbors import NearestNeighbors
from functools import lru_cache

# Set page configuration
st.set_page_config(page_title="Movie Recommender", page_icon=":clapper:",
layout="wide")

# Load pickled data (replace file paths with your actual locations)
@st.cache_data
def load_data():
    return {
        'movies_tamil': pickle.load(open('Tamil_movies.pkl', 'rb')),
        'features_tamil': pickle.load(open('T_matrix.pkl', 'rb')),
        'movies_international': pickle.load(open('movie_list.pkl', 'rb')),
        'features_international': pickle.load(open('matrix.pkl', 'rb')),
        'movies_indian': pickle.load(open('Indian_movies.pkl', 'rb')),
        'features_indian': pickle.load(open('I_matrix.pkl', 'rb'))
    }

data = load_data()

# Fetch movie data from OMDb API
@st.cache_resource
@lru_cache(maxsize=128)
def fetch_data(movie_name):
    api_key = "4a7c8e91"  # Replace with your OMDb API key
    url = f"https://www.omdbapi.com/?apikey={api_key}&t={movie_name}"
    try:
        response = requests.get(url)
        response.raise_for_status()
        data = response.json()
        return (
            data.get('Year', 'N/A'),
            data.get('Director', 'N/A'),
            data.get('Actors', 'N/A'),
            data.get('Plot', 'N/A'),
            data.get('Poster', None)
        )
    except requests.exceptions.HTTPError as http_err:
        st.error(f"HTTP error occurred: {http_err}")
    except requests.exceptions.RequestException as err:
        st.error(f"Error: {err}")
    return None, None, None, None, None

def recommend(movie, movies_df, features):
    try:
        index = movies_df[movies_df['Title'].str.lower() == movie.lower()].index[0]
        knn = NearestNeighbors(n_neighbors=6, algorithm='auto')
        knn.fit(features)
        distances, indices = knn.kneighbors(features[index].reshape(1, -1))
        return [movies_df.iloc[i].Title for i in indices.flatten()[1:]]
    except IndexError:
        st.warning("Movie not found in the database.")
        return []

def fuzzy_search(search_term, movie_list, threshold=80):
    return [movie for movie, score in process.extract(search_term, movie_list,
limit=10) if score >= threshold]

def display(selected_movie, features, movies):
    col1, col2 = st.columns([1, 2])
    movie_details = fetch_data(selected_movie)

    if movie_details:
        with col1:
            if movie_details[4]:  # Check if poster URL is available
                poster_html = f'<a
href="https://www.google.com/search?q={selected_movie}"
target="_blank"><img src="{movie_details[4]}" width="230"></a>'
                st.markdown(poster_html, unsafe_allow_html=True)
            else:
                st.write("Poster not available.")

        with col2:
            st.markdown(f"### {selected_movie}")
            st.write("---")
            st.write(f"**Year**: {movie_details[0]}")
            st.write(f"**Director**: {movie_details[1]}")
            st.write(f"**Actors**: {movie_details[2]}")
            st.write(f"**Plot**: {movie_details[3]}")

        st.markdown('## Similar Movies:')
        recommended_movie_names = recommend(selected_movie.lower(), movies,
features)
        if recommended_movie_names:
            cols = st.columns(5)
            for col, movie_name in zip(cols, recommended_movie_names):
                col.text(movie_name)
                recommended_details = fetch_data(movie_name)
                if recommended_details and recommended_details[4]:
                    poster_html = f'<a
href="https://www.google.com/search?q={movie_name}" target="_blan "><img
src="{recommended_details[4]}" width="100%"></a>'
                    col.markdown(poster_html, unsafe_allow_html=True)
                else:
                    col.write("Poster not available.")
        else:
            st.write("No similar movies found.")
    else:
        st.warning("Movie details not available.")

# Sidebar for language selection and search
with st.sidebar:
    st.header('   Movie Recommender System')
    language_option = st.selectbox(
        'Choose your movie language:',
        ('International', 'Tamil', 'Indian')
    )
    # Search functionality
    search_term = st.text_input("   Search for a movie:")
    if language_option == "Tamil":
        movies = data['movies_tamil']
        features = data['features_tamil']
        top_movies = ["Singam", "Billa", "Sivaji", "Vaaranam Aayiram", "Indian"]
    elif language_option == "Indian":
        movies = data['movies_indian']
        features = data['features_indian']
        top_movies = ["Jersey", "3 Idiots", "Dangal", "Pink", "Mahanati"]
    else:
        movies = data['movies_international']
        features = data['features_international']
        top_movies = ["Avengers: Age of Ultron", "2012", "The Dark Knight",
"Spider-Man", "X-Men"]

    filtered_movies = fuzzy_search(search_term, movies['Title'].tolist()) if
search_term else movies['Title'].tolist()
    selected_movie = st.selectbox("Select a Movie:", filtered_movies)
    if st.button('Recommend Movies'):
        if selected_movie:
            lowercase_movie = selected_movie.lower()
            if lowercase_movie in movies['Title'].str.lower().values:
                st.session_state["selected_movie"] = selected_movie
            else:
                st.warning("The selected movie is not found in the database.")
                st.session_state["selected_movie"] = None
        else:
            st.warning("Please select a movie before recommending.")

    if st.button('Clear Selection'):
        st.session_state["selected_movie"] = None
# Main content area
st.markdown('### Top Rated Movies:')
# Display the posters of the top 5 movies in a row
cols = st.columns(5)
for col, top_movie in zip(cols, top_movies):
    movie_details = fetch_data(top_movie)
    if movie_details and movie_details[4]:
        poster_html = f'<a href="https://www.google.com/search?q={top_movie}"
target="_blank"><img src="{movie_details[4]}" width="100%"></a>'
        col.markdown(poster_html, unsafe_allow_html=True)
        if col.button(f"Select {top_movie}", key=top_movie):
            st.session_state["selected_movie"] = top_movie

# Placeholder for displaying selected movie details and recommendations
placeholder = st.empty()

if "selected_movie" in st.session_state and st.session_state["selected_movie"]:
    selected_movie = st.session_state["selected_movie"]
    with placeholder.container():
        display(selected_movie, features, movies)


st.markdown("---")
st.markdown(
    "© 2024 [Abhijit](https://www.linkedin.com/in/abhijit-r-3015b8257) |
[Faleel](https://www.linkedin.com/in/faleel-mohsin-251643297/) |
[Monish](https://www.linkedin.com/in/monish-raja-rathinam-m-056850257). All
rights reserved."
```

REFERENCES

1. Harper, F. M., & Konstan, J. A. (2015). The MovieLens datasets: History and context. ACM Transactions on Interactive Intelligent Systems (TiiS), 5(4), 1-19.

2. Kumar, R., & Thakur, N. S. (2019). An approach towards hybrid movie recommendation system. International Journal of Recent Technology and Engineering (IJRTE), 8(3), 7173-7179.

3. Bobadilla, J., Ortega, F., Hernando, A., & Gutiérrez, A. (2013). Recommender systems survey. Knowledge-Based Systems, 46, 109-132.

4. Musto, C., Semeraro, G., Lops, P., & de Gemmis, M. (2015). A comparison of techniques for recommending movies in a content-based framework. Proceedings of the 9th ACM Conference on Recommender Systems.

5. Chen, K., Zhang, J., & Cao, L. (2014). A personal movie recommendation system based on collaborative filtering. Procedia Computer Science, 4, 228-234.

6. Said, A., & Bellogín, A. (2014). Comparative recommender system evaluation: Benchmarking recommendation frameworks. Proceedings of the 8th ACM Conference on Recommender Systems, 129-136.