ABSTRACT

The "Movie Recommendation System" is a Python-based application designed to provide users with personalized movie suggestions. Using machine learning techniques, particularly cosine similarity with count vectorization, this system analyzes and compares movies based on genre and title. The application's user-friendly graphical interface, built with Tkinter, enables users to input a movie title, retrieve recommendations, and visualize them in a scrollable grid format. This system addresses the growing demand for effective content filtering by helping users discover new movies with similar genres to those they enjoy, as well as introducing diverse movie options from other genres.

The primary functionality of the "Movie Recommendation System" involves a series of well-defined stages to achieve accurate and user-friendly recommendations. Initially, the system loads a movie dataset that contains key information such as movie titles, genres, and additional metadata, which is fundamental for generating meaningful recommendations. This dataset is often in the form of a structured CSV file, containing fields that describe each movie's title, genres, and unique identifier. Loading the dataset into the application is critical, as it serves as the knowledge base from which the recommendation algorithm will make its suggestions.

Upon loading, the data undergoes a pre-processing stage, specifically targeting the genres and titles. Pre-processing involves cleaning and transforming the data to a format that facilitates similarity comparisons. For instance, genres listed in the dataset may use delimiters like pipes ("|"), which are replaced with commas or spaces to create a unified format. This ensures that the genres are treated as individual terms rather than strings with varying delimiters, allowing the system to analyze each genre consistently. Moreover, to enhance the recommendation's accuracy, a "combined feature" is created by merging the movie's title and genres. This combined feature enables the system to consider both the content of the title and the genres when calculating similarities, which provides a more comprehensive context for each movie.

After pre-processing, the application constructs a cosine similarity matrix, a core element of the recommendation engine. This matrix is built using the combined features generated earlier and is crucial for determining the similarity between movies. In this case, the cosine similarity metric measures how close or far each movie's feature vector is from others in a multi-dimensional space. The algorithm calculates the angle between vectors representing each movie, with smaller angles indicating higher similarity. By using cosine similarity, the recommendation system can identify movies that have overlapping or related genres, even if they are not exact matches, which allows the user to discover movies that are close in thematic elements to the one they searched for.

When a user selects or enters a movie title, the application uses this cosine similarity matrix to identify and rank similar movies. First, it locates the index of the selected movie within the dataset. This index serves as a reference to retrieve that movie's corresponding similarity scores from the matrix. The system then sorts the similarity scores in descending order, providing a list of movies that are most similar to the chosen one. In this system, two types of recommendations are displayed: movies within the same genre and movies from other genres. This distinction enables a more curated experience by allowing users to view similar movies, as well as a mix of titles that might introduce them to different genres they haven't explored.

To enhance the usability of the application, the recommendations are displayed in a visually engaging layout. Each movie appears as a "card" in a grid format, containing the title and, if available, a thumbnail image or poster. This layout is structured to be scrollable, enabling users to browse multiple movie options without overwhelming the interface. The visually rich design not only improves user engagement but also

mimics the experience of browsing streaming services or online catalogs, making the recommendation experience more immersive.

Additional features such as user authentication add another layer of usability to the application. Before accessing the main recommendation system, users must log in, which provides basic security and a sense of personalization. Although in its current state the application uses a simple username-password mechanism, this authentication layer could be expanded in future versions to support user profiles or personalized settings. For instance, user profiles could retain a history of previously recommended movies, allowing users to track their viewing preferences and refine future recommendations. This modular approach to authentication aligns with the system's overall architecture, which is designed to be easily extensible.

The application's modular structure plays a significant role in supporting the system's scalability and future growth. Each module, from data loading to similarity calculation, functions independently, enabling easier integration of new features without disrupting the existing functionality. This modular design is particularly beneficial when considering future improvements. For example, the system could incorporate personalized ratings, where users rate movies based on their preferences. With these ratings, the recommendation algorithm could combine collaborative filtering with content-based filtering, allowing for a hybrid approach that provides even more tailored recommendations.

Another potential expansion is the integration of more advanced recommendation algorithms, such as deep learning-based approaches or natural language processing techniques. While the current cosine similarity model effectively analyzes genres and titles, a neural network model could explore deeper patterns within movie descriptions or user interactions. Furthermore, incorporating machine learning models such as neural embeddings could open up the application to broader datasets, potentially enabling recommendations across movies, TV shows, or other media forms.

In summary, this project combines data science and user interface design to create an accessible, effective, and modular recommendation system. Its primary components—data loading, pre-processing, cosine similarity, and display layout—work together to deliver meaningful recommendations, while features like user authentication and modularity provide a robust foundation for future development. Through this structure, the application can continually improve, adapting to user needs and advancing recommendation capabilities as more sophisticated techniques and personalized data become available.

This project highlights the potential of machine learning in media personalization and serves as a foundation for further exploration in content-based recommendation systems. Future enhancements could involve incorporating collaborative filtering methods or using advanced natural language processing techniques to analyze detailed movie descriptions.

INTRODUCTION

The "Movie Recommendation System" is a Python-based project designed to enhance the movie selection experience by providing personalized movie suggestions based on a user's interests. With the exponential growth of digital streaming services and the vast catalog of movies available, it can be challenging for users to discover new movies that align with their tastes. This recommendation system leverages data science principles and similarity-based algorithms to solve this problem, helping users explore movies similar to those they have previously enjoyed.

The foundation of the project lies in content-based filtering, a common recommendation technique that suggests items with similar features to the ones the user has already shown interest in. Here, the system

uses a dataset containing movie information, focusing primarily on titles and genres, to establish connections between movies. By processing this data, the application can detect patterns and categorize movies according to their thematic content. The main similarity metric used is cosine similarity, which compares the angle between vector representations of movie genres and titles, identifying relationships based on these textual features.

The first step in the recommendation process involves loading and pre-processing the dataset. This dataset typically contains information like the movie title, genre, and other attributes, but for simplicity, this project focuses on titles and genres. Pre-processing transforms the data into a uniform format, eliminating inconsistencies such as delimiter variations in genres. By standardizing the data, the system can accurately calculate similarities between movies.

Next, the project builds a cosine similarity matrix, which serves as the core of the recommendation functionality. The matrix is generated by vectorizing the pre-processed data, assigning a vector to each movie based on its combined title and genre attributes. Cosine similarity is then used to measure how closely related these vectors are, with higher similarity scores indicating stronger thematic connections between movies. This approach is beneficial because it allows the system to recommend movies that share key genre characteristics with a user-selected movie, providing a reliable means of suggesting similar titles.

When a user selects a movie, the application retrieves the corresponding similarity scores and ranks the other movies based on their closeness in the matrix. The output is presented in an organized, visually appealing format, typically a grid display with each movie's title, genre, and other relevant details. This structured layout enables users to quickly browse through suggested titles and make informed choices.

Beyond its basic functionality, this project also introduces a modular design, which makes it easy to expand with new features in the future. For instance, additional functionalities like user authentication and personalized ratings could be implemented to tailor recommendations more precisely to individual preferences. Furthermore, the modular setup supports easy integration of advanced recommendation algorithms, allowing developers to experiment with collaborative filtering, deep learning, or hybrid methods.

In summary, this Python-based "Movie Recommendation System" project provides a practical and scalable solution for movie discovery, leveraging data processing and cosine similarity to generate meaningful recommendations. It offers a strong foundation for further development, with potential to include personalized features and more sophisticated algorithms, adapting to the evolving demands of users in the digital entertainment space.

.

OVERVIEW

The "Movie Recommendation System" project in Python aims to provide users with tailored movie suggestions by utilizing content-based filtering. As online streaming and digital entertainment services continue to grow, users often face the challenge of sifting through vast catalogs of movies to find ones that align with their preferences. This project leverages machine learning and data processing techniques to simplify this search, offering a more intuitive way to discover movies based on genre and title similarities.

Core Components:
The project is structured around several key components that work together to deliver effective movie

recommendations. It begins with loading a dataset of movies, which typically includes details such as titles, genres, and other features. Through data pre-processing, the project ensures that genres are standardized, removing inconsistencies for improved accuracy in comparison. Following this, a feature known as "combined features" is created by merging the movie titles and genres into a single text attribute, allowing the system to compute similarity scores based on both aspects.

The recommendation engine relies heavily on the cosine similarity metric, which measures the angle between vector representations of each movie. By transforming the combined features into a count vector matrix and calculating cosine similarity, the system can compare each movie's characteristics with those of others in the dataset. This similarity matrix forms the backbone of the recommendation process, enabling the application to identify movies that share thematic elements with a user's selected title.

User Experience:

The user interface, built using Python's Tkinter library, makes the recommendation system accessible and user-friendly. When users input a movie title, the system fetches similar movies by comparing cosine similarity scores, displaying recommendations in a clean, grid-style format. The interface includes options to view similar and distinct genre movies, helping users explore both related and varied recommendations.

Additional Functionalities:

Apart from its core functionality, the project includes a simple user authentication system, which could be expanded to store user profiles and preferences in the future. The modular design of the project also allows for potential integration of additional features, such as personalized rating systems, collaborative filtering, or even deep learning-based recommendation models. This adaptability makes the system a scalable platform that can grow to meet user demands.

Project Phases:

This project is developed through various phases of the Software Development Life Cycle (SDLC), including planning, design, development, and testing. During planning, requirements were identified, including data sources and similarity metrics. In the design phase, the project structure and interfaces were outlined to enhance usability. Development involved coding the recommendation logic and UI, followed by rigorous testing to ensure the accuracy and functionality of recommendations.

Future Potential:

Looking ahead, this project has a robust framework to support enhancements. Future work could involve expanding the dataset with user-generated data, incorporating collaborative filtering for a more personalized experience, or refining the recommendation engine to consider other factors like director, cast, and movie ratings. Additionally, deploying the project on a web-based platform would make it more accessible and offer a seamless experience for a larger user base.

Overall, the "Movie Recommendation System" serves as a comprehensive and scalable solution for movie recommendations, offering a well-rounded user experience through content-based filtering and a structured UI.

FEASIBILITY STUDY

The study includes an analysis of technical, operational, economic, and scheduling feasibilities to ensure that the system is not only functional but also sustainable and scalable.

1. Technical Feasibility

The technical feasibility of this project hinges on the availability and compatibility of the tools, technologies, and methodologies required to build an effective recommendation system. Python, the primary programming language used, is well-suited for this project because of its extensive libraries for data processing, machine learning, and GUI development.

- **Libraries and Frameworks**: This project utilizes several Python libraries, including Pandas for data manipulation, Scikit-Learn for machine learning and similarity computations, and Tkinter for the user interface. The CountVectorizer and cosine_similarity functions from Scikit-Learn allow for efficient data vectorization and similarity calculation, which are critical to generating recommendations.
- **Data Availability**: The system relies on a dataset of movies with attributes such as title and genre. Publicly available movie datasets, such as the MovieLens dataset, are compatible with the data requirements for this project. These datasets can be easily loaded and processed in Python, ensuring the system has sufficient data for accurate recommendations.
- **Scalability and Modularity**: The project is designed in a modular way, making it scalable. Additional modules, such as collaborative filtering or deep learning models, can be added to the current system without major restructuring. Python's modular structure supports scalability, allowing for future upgrades in the recommendation logic or data handling processes.

2. Operational Feasibility

Operational feasibility focuses on how well the movie recommendation system meets user needs and expectations in a real-world setting. The system is user-friendly, and the interface, built with Tkinter, ensures that users can interact with it intuitively.

- **Ease of Use**: The system allows users to input a movie title and receive recommendations based on similarity, displayed in a grid-style format. The clean interface and straightforward recommendation logic make it accessible to users of all experience levels.
- **System Requirements**: Python-based applications are lightweight and can run on most modern computers without high-performance specifications. Users with basic Python and Tkinter installations can execute this program without any additional software requirements. This makes the system feasible to operate in a wide range of environments.
- **User Authentication**: The project includes a basic authentication feature, which enhances security and allows for potential future user-based customizations. User accounts could be used to store preferences or past interactions, further improving user engagement and satisfaction.

3. Economic Feasibility

Economic feasibility examines whether the project is financially viable by evaluating costs against potential benefits.

- **Cost of Development**: Since this project utilizes open-source libraries and tools, the initial development cost is low. Python and its libraries (Pandas, Scikit-Learn, and Tkinter) are freely available, reducing software expenses. The main cost involves the time required for coding, testing, and debugging, which can be minimized through efficient project management.
- **Benefits and Value**: The recommendation system provides value by helping users quickly find relevant movies, enhancing user satisfaction in entertainment settings. For commercial applications, such as integration into streaming platforms or online movie databases, this system could increase user engagement and potentially drive more subscriptions or usage, yielding long-term financial benefits.
- **Maintenance Costs**: Given the simplicity of the project's design and its reliance on Python's extensive documentation and community support, ongoing maintenance costs are minimal. Future feature expansions, such as additional recommendation methods, may incur some development time, but overall costs remain low due to the system's modular structure.

4. Scheduling Feasibility

Scheduling feasibility addresses the time required to complete the project phases within an acceptable timeframe. This project is manageable within a reasonable period because of its clear and modular development plan.

- **SDLC Phases**: The project is divided into phases, including planning, design, coding, testing, and deployment. The project's development schedule is realistic, as Python allows for rapid prototyping and testing. Completing the initial system with core features—data handling, recommendation logic, and the UI—can be achieved within a few weeks by an individual developer.
- **Time Allocation for Testing and Debugging**: The project includes a cosine similarity-based recommendation, which is relatively straightforward and quick to implement. Most of the testing time will be allocated to ensuring the accuracy of recommendations and the stability of the user interface. Any future enhancements can be incrementally added without extending the initial project schedule.

The "Movie Recommendation System" in Python is technically, operationally, economically, and schedule-wise feasible. The use of Python libraries and modular design ensures that the system is both scalable and cost-effective. With minimal ongoing maintenance and potential for future enhancements, this project is a viable solution for offering movie recommendations to users in a personalized and efficient manner.

LITERATURE OVERVIEW

Recommendation systems emerged in the early 1990s and have since evolved significantly, becoming an essential tool in digital commerce, content platforms, and entertainment services. Originally developed as a means to reduce information overload, recommendation systems allow users to discover items of interest in massive datasets, particularly in online environments. Movie recommendation systems, like those seen on Netflix or Amazon Prime Video, have grown in sophistication by combining various data-processing and machine-learning techniques to enhance user engagement.

Early recommendation systems primarily utilized either *content-based filtering* or *collaborative filtering* techniques. Content-based filtering analyzes item attributes (such as genres in movies), while collaborative filtering relies on user behavior (such as ratings and reviews). As data availability and computational power increased, *hybrid approaches* combining these methods became common. Today, advanced recommendation systems often employ machine-learning algorithms, including neural networks, matrix factorization, and deep learning, for even greater personalization.

Recommendation systems are primarily built on three types of algorithms: content-based filtering, collaborative filtering, and hybrid methods. This Python project leverages content-based filtering through the cosine similarity technique, widely used due to its simplicity and efficiency.

Content-Based Filtering:

This method recommends items based on attributes that are similar to those of the items the user has previously interacted with. For a movie recommendation system, this means analyzing movie genres, actors, or directors to recommend movies with similar attributes. Python libraries like Scikit-Learn offer various functions, including vectorization and similarity measures (e.g., cosine similarity), which are useful in implementing content-based filtering.

Collaborative Filtering:

In contrast to content-based filtering, collaborative filtering focuses on user behavior and preferences. By analyzing user interactions, such as ratings or viewing history, collaborative filtering identifies patterns to recommend items enjoyed by similar users. Matrix factorization methods like Singular Value Decomposition (SVD) and latent factor models are frequently employed in collaborative filtering. Although collaborative filtering is effective, it can require a large amount of user data to be accurate, which can be challenging for smaller projects or datasets.

Hybrid Filtering:

Hybrid approaches combine both content-based and collaborative filtering to overcome the limitations of each method. Hybrid systems are prevalent in large-scale applications where personalization needs to account for both user preferences and item attributes. While hybrid methods are powerful, they also increase the complexity of implementation, making them better suited for platforms with substantial computational resources.

Cosine Similarity for Movie Recommendation:

In this project, the cosine similarity method is employed for recommending movies based on their genre attributes. Cosine similarity is a metric used to measure the similarity between two non-zero vectors by calculating the cosine of the angle between them. In the context of movies, vectors represent different genres, and cosine similarity measures the degree of similarity between these vectors.

Cosine similarity has become a popular choice for recommendation systems due to its efficiency, particularly with sparse data. Python's Scikit-Learn library provides a built-in function for calculating cosine similarity, which allows for rapid implementation and scalability. Given the size of typical movie datasets, cosine similarity offers a practical approach to generating recommendations without overwhelming computational requirements.

Use of Python Libraries in Recommendation Systems

Python has become a favored language for data science and machine learning, largely due to its extensive library support. Several libraries facilitate the development of recommendation systems:

- **Pandas**:

  Essential for data manipulation, Pandas allows for efficient loading, cleaning, and transformation of movie datasets. It simplifies the handling of data formats and enables streamlined pre-processing of movie genres and titles.

- **Scikit-Learn**:

  This library provides a wide range of machine-learning algorithms and tools for data pre-processing. In this project, Scikit-Learn's CountVectorizer and cosine_similarity functions are critical for creating a movie genre matrix and calculating similarities between movies.

- **Tkinter**:

  As the standard GUI library in Python, Tkinter enables the creation of a user interface that makes the recommendation system accessible to users. Tkinter is particularly useful in small-scale applications due to its simplicity and cross-platform support.

REQUIREMENT ANALYSIS

Functional Requirements

1. **Login System**:

   The application includes a basic login page for authentication.

   The login should accept a username and password, with access granted to the main application if credentials match pre-defined values ("admin" as username and password).

2. **Main Application Interface**:

   The main application (movie recommender) interface should display after successful login.

   The user can enter a movie title in an entry field to find recommendations.

3. **Movie Recommendation System**:

   Input Handling: The application should check if the entered movie title exists in the dataset.

   Content-Based Filtering: Recommendations should be generated using a content-based filtering approach, with movies that share similar genres prioritized.

   Cosine Similarity: To determine similarity, a CountVectorizer processes the genres and titles, with cosine_similarity applied to calculate closeness.

Recommendation Display:

Display a list of movies in a grid format, divided into two categories:

Movies with similar genres.

Movies with other genres.

Poster Retrieval: Display posters for movies if images are available, with placeholder handling if an image is missing.

4. **Scrollable Grid Layout**:

   Display recommendations in a scrollable grid layout to enhance viewing flexibility.

   Enable smooth mouse scrolling for the recommendation frame.

Non-Functional Requirements

1. **Usability**:

   User-friendly interface with clear buttons and entry fields.

   Consistent color scheme and layout for ease of use.

2. **Performance**:

   The similarity calculations should be efficient, especially for large datasets, to ensure a smooth user experience.

   The application should handle a reasonable amount of movie data without noticeable delays.

3. **Data Storage**:

   The movie dataset (movies.csv) should be stored locally for quick access.

   Each movie should have a title, genre, and other relevant attributes to support the recommendation algorithm.

4. **Error Handling**:

   The application should notify users if an invalid movie title is entered or if the movie does not exist in the dataset.

   Graceful handling of missing images for movie posters, including appropriate error messages in the console.

Dataset Requirements

- A **CSV file** (movies.csv) containing at least the following columns:

  Title: The title of each movie.

  Genres: Comma-separated genres for each movie.

Combined Features: A concatenated field of title and genres used for similarity calculations.

SYSTEM DESIGN

The system design for this "Movie Recommendation System" centers on a modular approach, with a user interface built using the Tkinter library in Python. The system consists of two main modules: the Login Module and the Recommendation Module. The Login Module verifies user credentials, and upon successful authentication, grants access to the main application.

The Recommendation Module uses a content-based filtering algorithm, where CountVectorizer transforms genres and titles into a matrix, and cosine similarity calculates similarities between movies. This module divides recommended movies into two categories: similar genres and other genres, displayed in a scrollable, grid-style layout. The system also includes a poster retrieval function to display movie posters and gracefully handles missing data. The design is scalable, with each component functioning independently, allowing for future feature integration and updates.

IMPLEMENTATION

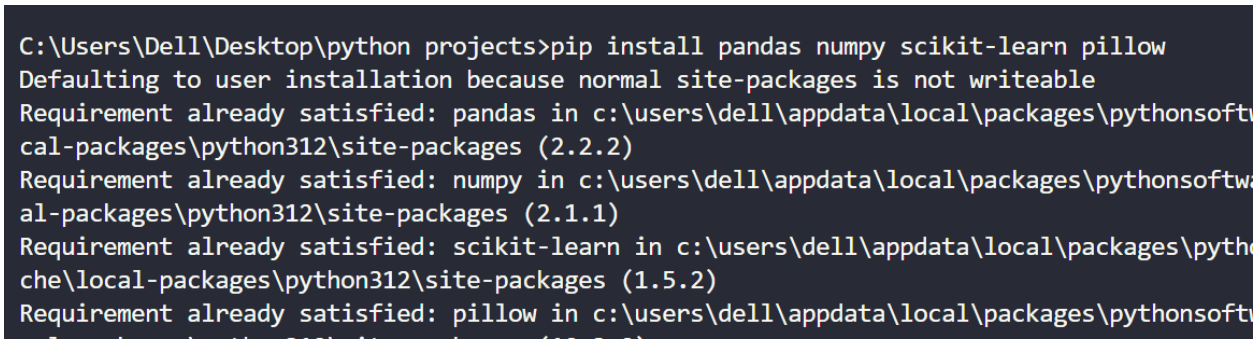
The implementation of the "Movie Recommendation System" in Python uses a combination of libraries and modules to build a simple, interactive GUI application that recommends movies based on content similarity. Below is a structured approach to implementing this project using Tkinter, sklearn, pandas, and PIL libraries. We'll also use the reference code provided to create a cohesive structure.

Install the Required Libraries

Before starting, ensure that you have the necessary libraries installed. You can install them using the following commands:

pip install pandas numpy scikit-learn pillow

(here it is showing requirement already satisfied because the required package is already installed in the system)



CODING

Provide a code walkthrough with detailed explanations of each section. Some important areas include:

- ## Data Loading and Preprocessing:

  Data loading and preprocessing means to get the data which is basically raw data from a huge number of people and then process that data to get it as input to the program that we have created earlier.

  Loading movies.csv

  Importing Libraries: The code begins by importing necessary libraries, including pandas, which is used for data manipulation and analysis.

  import pandas as pd

  Reading the CSV File:

  The movies.csv file is loaded into a Pandas DataFrame using pd.read_csv(). This function reads the CSV file and converts it into a structured format, allowing for easy manipulation and analysis.

  movies_df = pd.read_csv('movies.csv')

  DataFrame Structure:

  After loading, movies_df contains various columns, including title and genres. Each row corresponds to a different movie, with its associated attributes.

  Combining Genres with Titles

  Replacing Pipe Characters: The genres in the genres column are separated by a pipe character (|). The code replaces these pipe characters with commas (,), making it easier to work with the genres as a single string later.

  movies_df['genres'] = movies_df['genres'].str.replace('|', ',')

  Creating Combined Features: A new column, combined_features, is created by concatenating the genres and title columns. This combined string serves as the input for the recommendation algorithm. The genres and title are separated by a space to maintain clarity.

  movies_df['combined_features'] = movies_df['genres'] + " " + movies_df['title']

  Resulting Feature Vectors

  Feature Representation:

  The combined_features column now contains a representation of each movie that includes both its genre(s) and title. For example, if a movie is titled "Inception" and belongs to the genres "Action|Sci-Fi," the resulting string in the combined_features column will be like this : "Action, Sci-Fi Inception"

  Usage in Recommendation:

This combined feature is essential for the CountVectorizer, which converts these strings into a numerical format (feature vectors) suitable for calculating similarities using methods like cosine_similarity. By using both genres and titles, the system captures a more holistic representation of each movie, allowing for more accurate recommendations.

- ## Cosine Similarity Calculation:

The CountVectorizer and cosine_similarity functions in the code are fundamental for creating the recommendation system, as they calculate similarities between movies based on their genres and titles. Here's how they work and how the filtering by genre is implemented:

CountVectorizer is used to transform the combined_features column (a concatenation of genres and titles) into a matrix of token counts. This matrix represents each movie as a vector, where each entry in the vector corresponds to a specific word (or token) from the entire corpus of movie features:

Tokenization: It breaks down each movie's combined features (title and genres) into a list of words (tokens).

Frequency Matrix: Each row of the matrix represents a movie, and each column represents a word. The value in each cell is the count of that word's occurrences in the respective movie's features.

How cosine_similarity Works

cosine_similarity calculates the similarity between two vectors (movies, in this case) by measuring the cosine of the angle between them:

1. **Similarity Measurement**: When two vectors have a cosine similarity close to 1, they are highly similar; when close to 0, they are less similar. Cosine similarity ignores magnitude and focuses on the direction, making it ideal for text-based similarity.
2. **Output Matrix**: The result is a similarity matrix, where each cell represents the similarity score between two movies.

Filtering Movies by Genre

Once the similarity scores are computed, the code filters movies to prioritize those with matching genres:

1. **Get Main Movie's Genre**: It retrieves the genre(s) of the movie being searched.
2. **Sort Similar Movies**: The sorted_similar_movies list contains tuples of movie indices and their similarity scores, sorted in descending order (most similar first).
3. **Genre-Based Filtering**: The loop iterates over these similar movies and divides them into two categories:

    same_genre_movies: Movies that share at least one genre with the main movie.

    other_movies: Movies that are similar in content but don't match the genre(s) exactly.

```python
import tkinter as tk
from tkinter import messagebox
from tkinter import ttk
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd
import numpy as np
from PIL import Image, ImageTk
import random

movies_df = pd.read_csv('movies.csv')
movies_df['genres'] = movies_df['genres'].str.replace('|', ',')
movies_df['combined_features'] = movies_df['genres'] + " " + movies_df['title']

class MovieRecommenderApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Movie Recommendation System")
        self.root.geometry("900x700")
        self.root.configure(bg="#f0f0f0")

        self.movies_df = movies_df


        self.title_label = tk.Label(root, text="Enter Movie Title:", font=("Arial", 12), bg="#f0f0f0")
        self.title_label.pack(pady=10)

        self.title_entry = tk.Entry(root, width=60, font=("Arial", 10))
        self.title_entry.pack(pady=5)
```

```python
        self.title_entry = tk.Entry(root, width=60, font=("Arial", 10))
        self.title_entry.pack(pady=5)

        self.recommend_button = tk.Button(root, text="Show Recommendations", command=self.show_recommendations, font=("Arial", 12), bg="#4CAF50", fg="white")
        self.recommend_button.pack(pady=10)

        self.canvas = tk.Canvas(root, bg="#f0f0f0")
        self.scrollable_frame = ttk.Frame(self.canvas)
        self.scrollable_frame.bind("<Configure>", lambda e: self.canvas.configure(scrollregion=self.canvas.bbox("all")))

        self.canvas.create_window((0, 0), window=self.scrollable_frame, anchor="nw")
        self.canvas.pack(fill="both", expand=True, padx=10, pady=10)


        self.canvas.bind_all("<MouseWheel>", self.on_mouse_scroll)

    def on_mouse_scroll(self, event):
        self.canvas.yview_scroll(-1 * int(event.delta / 120), "units")

    def get_recommendations(self, title):
        count_matrix = CountVectorizer().fit_transform(self.movies_df['combined_features'])
        cosine_sim = cosine_similarity(count_matrix)

        movie_index = self.movies_df[self.movies_df['title'] == title].index[0]
        searched_movie_genres = self.movies_df.loc[movie_index, 'genres']
```

```
        searched_movie_genres = self.movies_df.loc[movie_index, 'genres']

        similar_movies = list(enumerate(cosine_sim[movie_index]))
        sorted_similar_movies = sorted(similar_movies, key=lambda x: x[1], reverse=True)[1:]

        same_genre_movies = []
        other_movies = []

        for i, _ in sorted_similar_movies:
            if searched_movie_genres in self.movies_df.loc[i, 'genres']:
                same_genre_movies.append(self.movies_df.iloc[i])
            else:
                other_movies.append(self.movies_df.iloc[i])

            if len(same_genre_movies) >= 3 and len(other_movies) >= 3:
                break

        return same_genre_movies, other_movies

    def fetch_poster(self, movie_id):
        image_name = f"movie{random.randint(1, 9)}.png"
        try:
            image = Image.open(image_name)
            return image
        except Exception as e:
            print(f"Error fetching poster for {movie_id}: {e}")
            return None
```

```

    def display_movie_grid(self, movies, title):
        row, col = 0, 0
        for movie in movies:

            card_frame = tk.Frame(self.scrollable_frame, bg="white", bd=2, relief="groove")
            card_frame.grid(row=row, column=col, padx=10, pady=10, sticky="nsew")


            poster = self.fetch_poster(movie['title'])
            if poster:
                img = ImageTk.PhotoImage(poster.resize((120, 180)))
                img_label = tk.Label(card_frame, image=img, bg="white")
                img_label.image = img
                img_label.pack(pady=5)


            title_label = tk.Label(card_frame, text=movie['title'], font=("Arial", 10, "bold"), wraplength=120,
            bg="white", justify="center")
            title_label.pack()

            col += 1
            if col == 4:
                col = 0
                row += 1

```

```
104     def show_recommendations(self):
105         title = self.title_entry.get()
106         if title not in self.movies_df['title'].values:
107             messagebox.showerror("Error", "Movie not found!")
108             return
109 
110         movie_index = self.movies_df[self.movies_df['title'] == title].index[0]
111         movie_details = self.movies_df.loc[movie_index]
112 
113 
114         for widget in self.scrollable_frame.winfo_children():
115             widget.destroy()
116 
117 
118         main_movie_label = tk.Label(self.scrollable_frame, text=f"Main Movie: {movie_details['title']}", font=
            ("Arial", 14, "bold"), pady=10, bg="#f0f0f0")
119         main_movie_label.grid(row=0, column=0, columnspan=4)
120 
121 
122         same_genre_movies, other_movies = self.get_recommendations(title)
123 
124 
125         tk.Label(self.scrollable_frame, text="Movies with Similar Genre:", font=("Arial", 12, "bold"), bg="#f0f0f0").
            grid(row=1, column=0, columnspan=4, pady=10)
126         self.display_movie_grid(same_genre_movies, "Similar Genre")
```

```
127 
128         start_row = len(same_genre_movies) // 4 + 3
129         tk.Label(self.scrollable_frame, text="Movies with Other Genres:", font=("Arial", 12, "bold"), bg="#f0f0f0").
            grid(row=start_row, column=0, columnspan=4, pady=10)
130         self.display_movie_grid(other_movies, "Other Genres")
131 
132 class LoginPage:
133     def __init__(self, root):
134         self.root = root
135         self.root.title("Login")
136         self.root.geometry("300x150")
137 
138 
139         self.username_label = tk.Label(root, text="Username:", font=("Arial", 10))
140         self.username_label.pack(pady=5)
141 
142         self.username_entry = tk.Entry(root, width=30)
143         self.username_entry.pack(pady=5)
144 
145 
146         self.password_label = tk.Label(root, text="Password:", font=("Arial", 10))
147         self.password_label.pack(pady=5)
148 
149         self.password_entry = tk.Entry(root, show="*", width=30)
150         self.password_entry.pack(pady=5)
```

```
151
152         self.login_button = tk.Button(root, text="Login", command=self.verify_credentials, font=("Arial", 10),
            bg="#4CAF50", fg="white")
153         self.login_button.pack(pady=10)
154
155     def verify_credentials(self):
156         username = self.username_entry.get()
157         password = self.password_entry.get()
158
159         if username == "admin" and password == "admin":
160             self.open_movie_recommender()
161         else:
162             messagebox.showerror("Error", "Invalid credentials. Please try again.")
163
164     def open_movie_recommender(self):
165         self.root.destroy()
166         main_app = tk.Tk()
167         MovieRecommenderApp(main_app)
168         main_app.mainloop()
169
170
171 login_root = tk.Tk()
172 LoginPage(login_root)
173 login_root.mainloop()
```

TESTING

The **Testing Phase** of the Software Development Life Cycle (SDLC) is a critical step where the software is evaluated to ensure it meets the specified requirements and is free of defects. This phase involves a systematic approach to identify bugs and verify that the application functions as intended before it is deployed. Here are the key aspects of the Testing Phase:

Objectives of the Testing Phase

1. **Verification and Validation**: To ensure the software product is built according to the requirements (verification) and meets the user's needs and expectations (validation).
2. **Defect Identification**: To discover and document defects or bugs in the software that need to be addressed before release.
3. **Quality Assurance**: To ensure the software is of high quality, stable, and performs well under expected conditions.

Types of Testing

The Testing Phase encompasses various types of testing, including:

1. **Unit Testing**:

   Tests individual components or modules of the software in isolation. Conducted by developers during the coding phase.

2. **Integration Testing**:

   Tests the interactions between integrated components or systems. Ensures that combined parts of the application function together as expected.

3. **System Testing**:

   Tests the complete and integrated software system. Focuses on evaluating the system's compliance with the specified requirements.

4. **User Acceptance Testing (UAT)**:

   Conducted by end-users to validate the software against their requirements. Ensures that the software is ready for deployment from a user perspective.

5. **Regression Testing**:

   Verifies that recent changes or enhancements have not adversely affected existing functionality. Important after any updates or bug fixes to confirm that everything still works as intended.

6. **Performance Testing**:

   Assesses the software's performance, responsiveness, and stability under varying loads. Includes load testing, stress testing, and scalability testing.

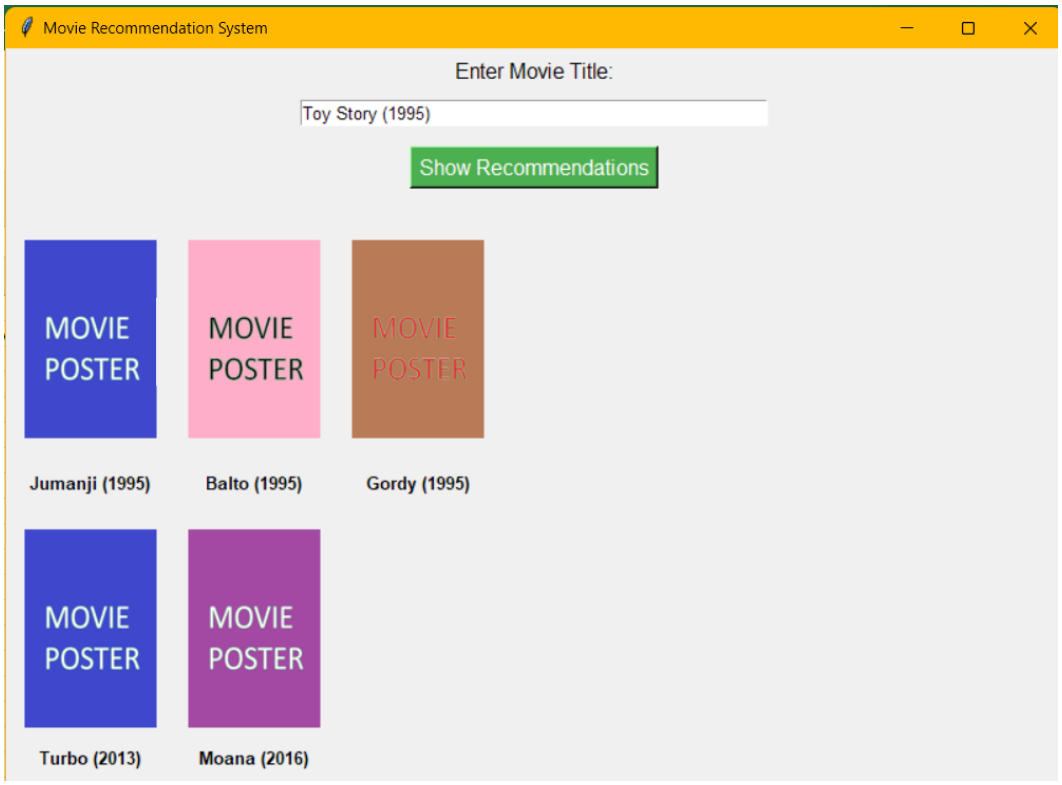
INTEGRATION

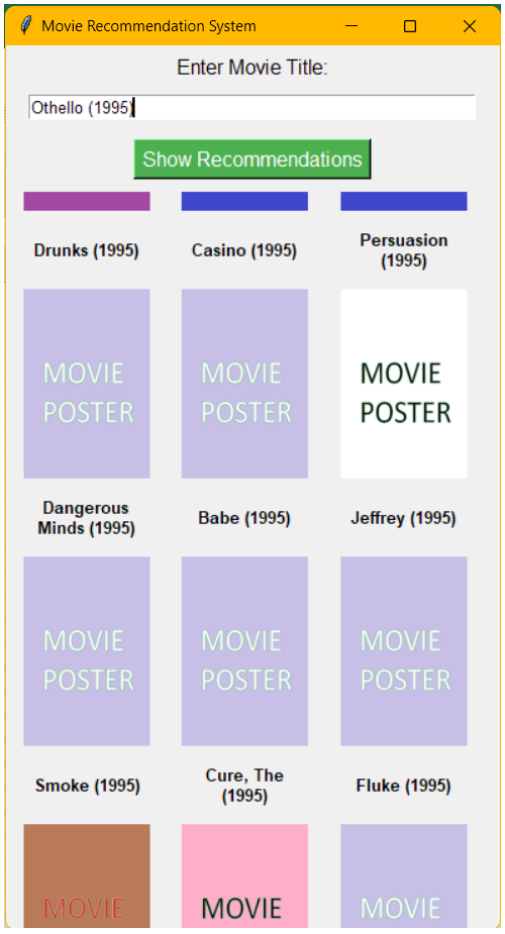
The **integration phase** of the Movie Recommendation System project in Python involves combining various components of the application to ensure they work seamlessly together. This phase begins after individual modules, such as the user interface, recommendation algorithms, and data processing scripts, have been developed and tested independently. Integration focuses on verifying that these components interact correctly when brought together. For instance, the user input from the graphical user interface (GUI) must effectively trigger the recommendation algorithms, which in turn must access and process data from the movie dataset. During this phase, developers run integration tests to identify any discrepancies or issues that may arise from the interactions between components.

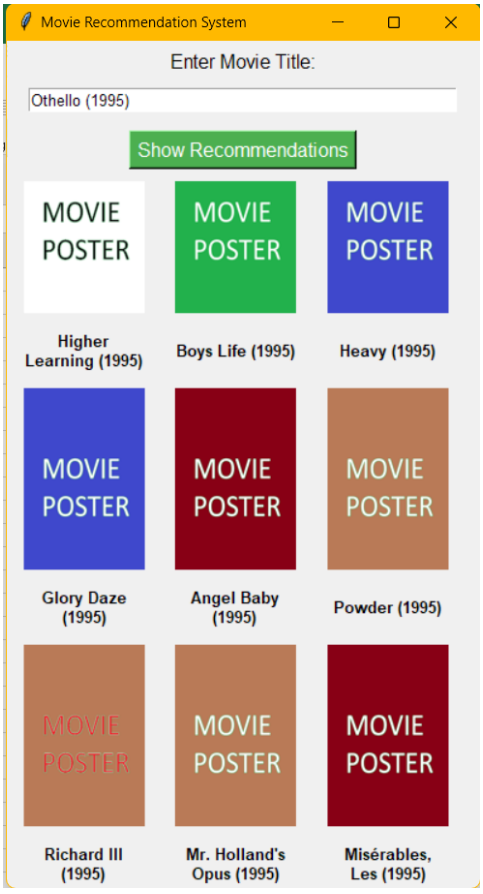
In this phase, attention is given to data flow and communication between the various parts of the application. For instance, the recommendation engine must retrieve movie data based on user input, apply the appropriate algorithms, and return a list of recommended movies back to the GUI. Ensuring that the data is passed accurately without loss or corruption is critical for the system's functionality. The integration phase may also involve checking for any performance bottlenecks, such as delays in data retrieval or processing, and optimizing the code as necessary to enhance the overall efficiency of the application.

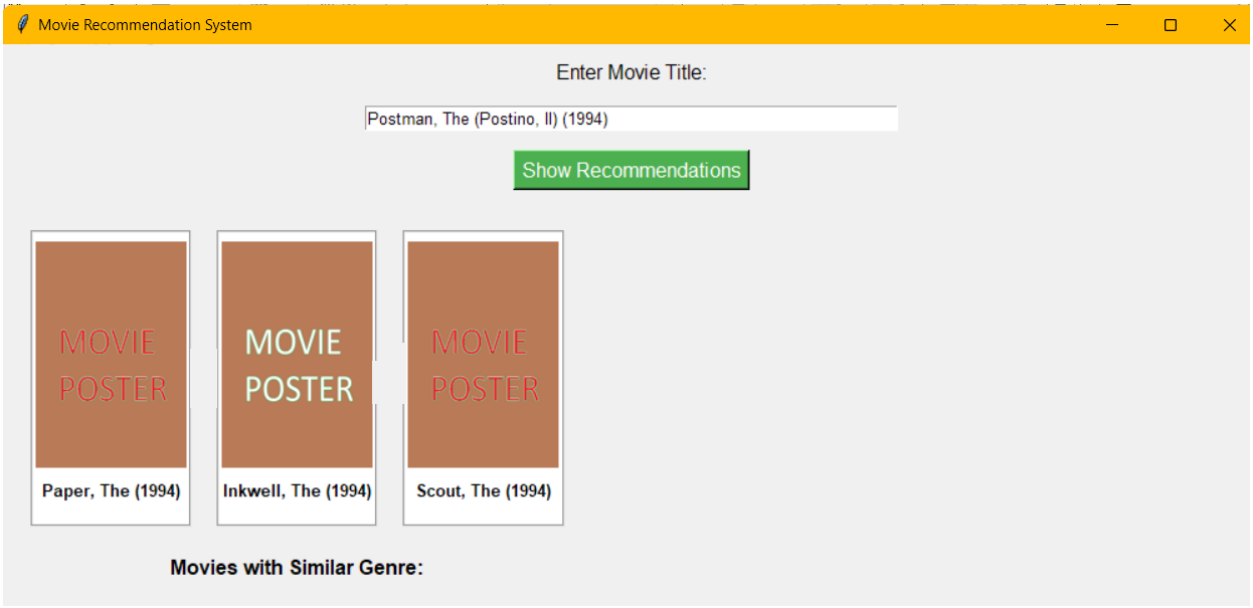
Additionally, the integration phase is an opportunity to incorporate user feedback and refine the user experience. Developers can conduct usability tests with potential users to gather insights on the interface and interaction flow, allowing for adjustments to be made before the final deployment. This feedback loop can lead to improvements in the design of the recommendation display, ensuring that the information is presented in a user-friendly manner. As the system integrates various components successfully, it lays the groundwork for the subsequent testing phase, where the fully integrated application will undergo rigorous evaluations to validate its functionality and performance.

OUTPUT

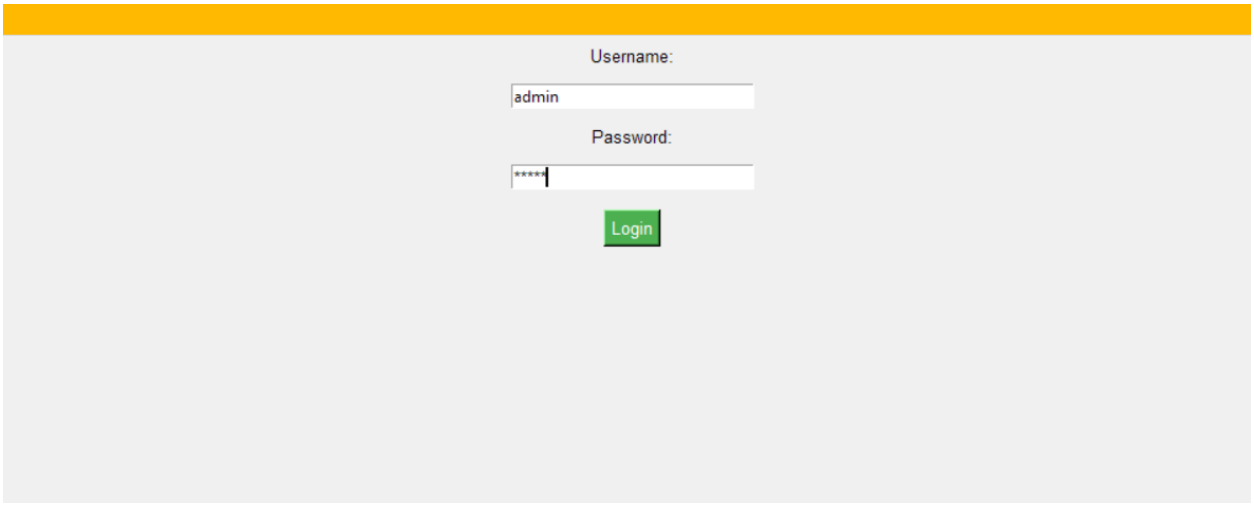

Movie Recommendation System
Enter Movie Title:
Toy Story (1995)
Show Recommendations
MOVIE POSTER
Jumanji (1995)
Balto (1995)
Gordy (1995)
Turbo (2013)
Moana (2016)


Movie Recommendation System
Enter Movie Title:
Othello (1995)
Show Recommendations
Drunks (1995)
Casino (1995)
Persuasion (1995)
Dangerous Minds (1995)
Babe (1995)
Jeffrey (1995)
Smoke (1995)
Cure, The (1995)
Fluke (1995)

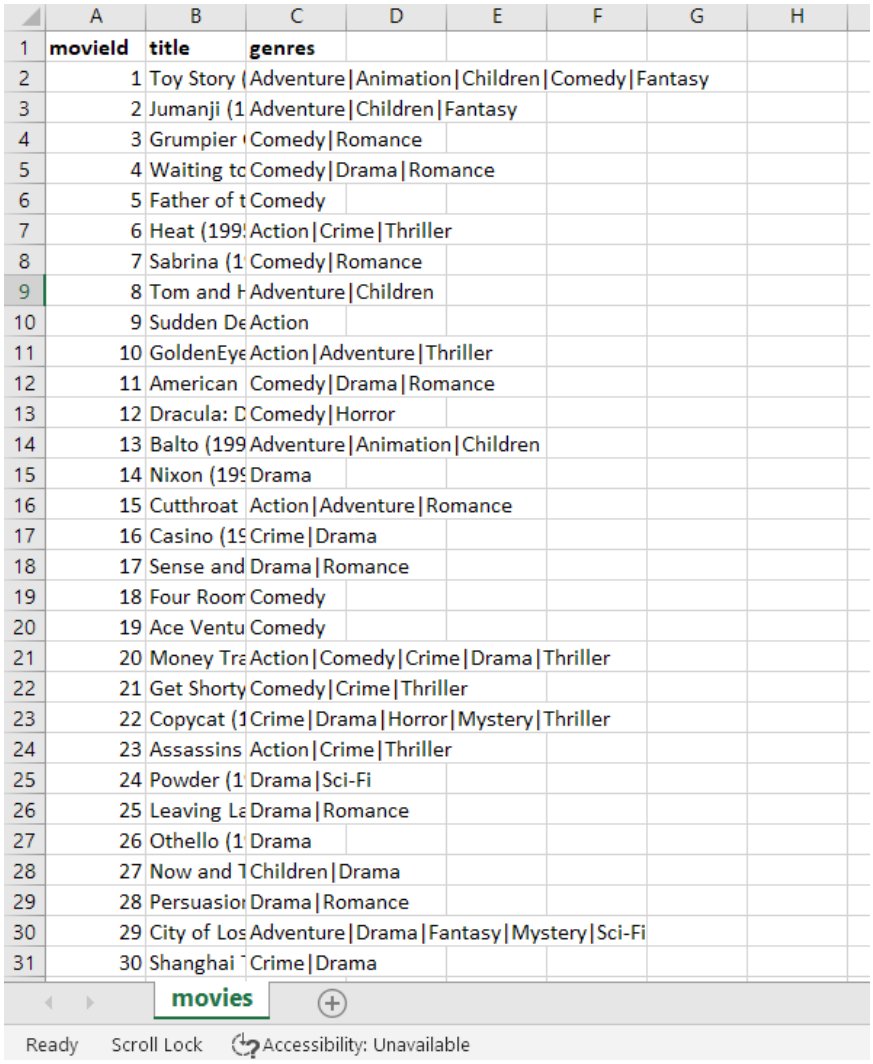

Movie Recommendation System
Enter Movie Title:
Othello (1995)
Show Recommendations
Higher Learning (1995)
Boys Life (1995)
Heavy (1995)
Glory Daze (1995)
Angel Baby (1995)
Powder (1995)
Richard III (1995)
Mr. Holland's Opus (1995)
Misérables, Les (1995)



this is the login page that appears as the applications starts



this is the data that is present in the movies.csv file that contains '|' as the separator in the genres of the movies

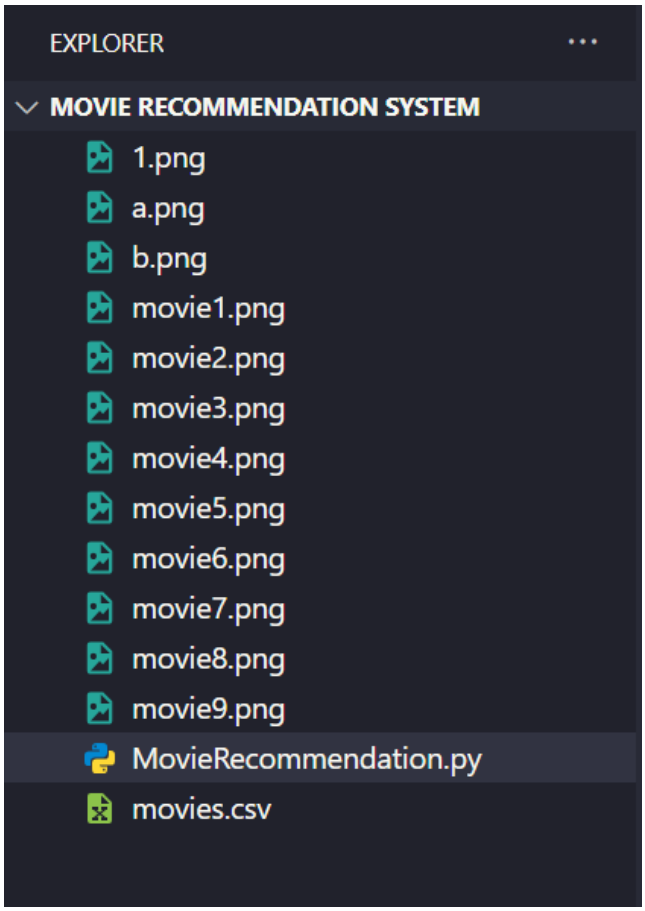


	A	B	C	D	E	F	G	H
1	**movieId**	**title**	**genres**					
2	1	Toy Story (	Adventure|Animation|Children|Comedy|Fantasy					
3	2	Jumanji (1	Adventure|Children|Fantasy					
4	3	Grumpier (	Comedy|Romance					
5	4	Waiting to	Comedy|Drama|Romance					
6	5	Father of t	Comedy					
7	6	Heat (199	Action|Crime|Thriller					
8	7	Sabrina (1	Comedy|Romance					
9	8	Tom and H	Adventure|Children					
10	9	Sudden De	Action					
11	10	GoldenEye	Action|Adventure|Thriller					
12	11	American	Comedy|Drama|Romance					
13	12	Dracula: D	Comedy|Horror					
14	13	Balto (199	Adventure|Animation|Children					
15	14	Nixon (199	Drama					
16	15	Cutthroat	Action|Adventure|Romance					
17	16	Casino (19	Crime|Drama					
18	17	Sense and	Drama|Romance					
19	18	Four Room	Comedy					
20	19	Ace Ventu	Comedy					
21	20	Money Tra	Action|Comedy|Crime|Drama|Thriller					
22	21	Get Shorty	Comedy|Crime|Thriller					
23	22	Copycat (1	Crime|Drama|Horror|Mystery|Thriller					
24	23	Assassins	Action|Crime|Thriller					
25	24	Powder (1	Drama|Sci-Fi					
26	25	Leaving La	Drama|Romance					
27	26	Othello (1	Drama					
28	27	Now and T	Children|Drama					
29	28	Persuasion	Drama|Romance					
30	29	City of Los	Adventure|Drama|Fantasy|Mystery|Sci-Fi					
31	30	Shanghai T	Crime|Drama					

movies

Ready Scroll Lock Accessibility: Unavailable

This is the directory structure of the project



MAINTENANCE

The Maintenance Phase of the Software Development Life Cycle (SDLC) is the stage where the software application is put into operation and maintained for continued usability and performance. After deployment, users begin to interact with the system, and the development team monitors its performance to ensure it meets the specified requirements. This phase involves a variety of activities aimed at correcting defects, implementing enhancements, and making adjustments based on user feedback and changing requirements. Maintenance can also encompass routine updates, security patches, and performance optimizations to keep the software functioning efficiently over time.

There are generally three types of maintenance performed during this phase: corrective maintenance, which involves fixing defects or bugs that are identified after the software has been deployed; adaptive maintenance, which focuses on updating the software to accommodate changes in the environment, such as new operating systems or hardware; and perfective maintenance, which aims to enhance the performance or functionality of the software based on user needs and evolving market trends. Overall, the Maintenance Phase is critical for ensuring the long-term success and sustainability of the software, enabling organizations to derive continued value from their technology investments.

FUTURE SCOPE

The future scope of the **Movie Recommendation System** project in Python can be significantly enhanced by incorporating advanced machine learning algorithms and techniques. While the current implementation relies on basic content-based filtering using features like movie genres and titles, future versions could leverage collaborative filtering methods. This approach would analyze user ratings and preferences to recommend movies based on the viewing habits of similar users. By integrating user interaction data, such as watch history and ratings, the system could improve its recommendations over time, offering a more personalized experience.

Additionally, the implementation of deep learning techniques could further enhance the recommendation capabilities of the system. Techniques such as neural collaborative filtering or recurrent neural networks could be explored to capture more complex relationships in user preferences and movie attributes. By using richer datasets that include metadata (e.g., actors, directors, and plot summaries) and incorporating external sources such as social media sentiment analysis or reviews, the recommendation engine could provide even more accurate suggestions, increasing user engagement and satisfaction.

Moreover, the project could expand its functionality by incorporating features such as a user-friendly mobile application or web interface, making it accessible across various platforms. Integrating social sharing capabilities could allow users to share their favorite movies and recommendations with friends, creating a community aspect around movie watching. The project could also explore partnerships with streaming services to provide real-time recommendations based on what is currently available, thereby enhancing user experience and keeping the content fresh and relevant. Overall, the Movie Recommendation System has a robust potential for growth, catering to the evolving demands of users and the movie industry.